Theory of Computing Semester Project

Nicole Thoen

nicolethoen@comcast.net

Spring 2016

CSCI 460

Table of Contents

Personal Statement

My study of the Theory of Computing has, first and foremost, widened my understanding of problem solving and computation. I have never engaged in the process of evaluating the computing power of different models of computation. I found that writing proofs of construction to demonstrate the computing power of different models to be a stimulating challenge each time. I also enjoyed learning about and building Push-Down Automata and Turing Machines.

While most of this class was fairly theoretical, I do see that being able to describe a problem or a process well and construct a machine which reliably models that process is extremely practical and relevant to software engineering. At my job, there is a concentrated effort to make all business processes paperless. Many of these processes involve rules, such as approval needed from certain people at certain times or extra steps being added in certain circumstances. The challenge of modeling these processes so that they can be coded and done completely without any paper mirrors the method of defining and constructing state machine and corresponding transition functions.

When presented with the task of converting to a paperless workflow, we began by modeling the workflow with a workflow diagram and a list of rules. This sort of maps to a finite state machine and a language (such as a regular or context free language). In this case, we had to craft a number of test cases to be sure that anyone entering our workflow could eventually arrive at a valid terminal state. Such exercises at work are improved by an understanding of how these models of computation work and knowing how to think in this framework.

Definitions

Predicate logic

Propositional logic (propositional calculus) is a language with values (*True* and *False*) and operations ($\neg$, $\wedge$, $\vee$, $\rightarrow$). The language can also use variables to represent *True* or *False* values, or formulas what evaluate to true or false.

It is a language of assertions that each evaluate to true or false, including assertions of the negation: $\neg p$, conjunction: $p \wedge q$, disjunction: $p \vee q$, implication: $p \rightarrow q$ (equivalent to: $\neg q \vee p$), and equivalence: $p \Leftrightarrow q$.

Discrete Finite Automaton (DFA)

DFAs are transition systems. A transition system is defined as a labeled digraph, where vertices denote state (memory), edges denote transitions and labels.

Formally, a DFA can be expressed as a 5-tuple $\langle Q, \Sigma, \delta, q0 , F\rangle$ where Q is a finite set of states, Σ is a finite alphabet, δ is the set of state-transition functions which map a state and an symbol from Σ to a state, q0 is the starting state, and F is the set of accepting states. An input string is read into the DFA one character at a time beginning q0. The first character and q0 will be passed as input to the appropriate transition function in δ to determine the next state. That next state will be passed along with the next character in the input to the appropriate transition function in δ to determine the next state, and so on. If the repeated application of δ to the (state, symbol) pairs in a computation leaves the DFA in a state in F, accept the input; else reject.

Context Free Grammar

Context Free Grammars define a Context Free Language using production rules which map non-terminal symbols to one or more sequence of terminal and non-terminal symbols.

Formally, a CFG can be expressed as a 4-tuple $\langle \Sigma, NT, R, S \rangle$, where Σ is a set of terminal symbols, NT is a set of nonterminals, R is the set of production rules s.t. $R \subseteq NT \times (\Sigma \cup NT)$, and S is the start symbol s.t. $S \in NT$. Production rules take the form $X \rightarrow Y Z\ldots$ where $X \in NT$ and $Y, Z\ldots \in (\Sigma \cup NT)$.

Turing Machine

Turing machines are augmented DFAs with a two-way read/write tape with infinite capacity. Each transition function uses the current symbol being read by the tape head and the current state of the DFA to determine both a tape operation and a next state. The tape could be that the tape head moves left, moves right, or writes symbol at the current location. The machine halts when it enters a "halting" (accept or reject) state. In a Turing machine, the tape is infinite in both directions and the tape head starts at leftmost nonblank cell. Also, the first blank to right of a symbol in Σ has infinitely many blanks to its right.

Formally, a Turing machine can be expressed as a 7-tuple $\langle Q, \Sigma, G, \delta, q0 , qaccept, qreject\rangle$ where Σ and G are input and tape alphabets, Q is the set of states, q0 is the start state, qaccept is the set of accept states, qreject is the set of request states, and δ is the set of transition functions s.t. δ: $Q \times (\Sigma \cup \{\text{'\#'}\} \cup G) \rightarrow Q \times (\Sigma \cup G \cup \{\text{'\#'}, L, R\})$ and L, R denote left or right moves.

Kripke Structures

Unlabeled transition system, used to diagram and reason about reactive systems. A Kripke structure corresponds an infinite computation tree reflecting all possible paths through the system.

Proofs

Computing power of finite domain models

A Loopless transition system is an example of a finite domain model and an acceptor. It is a graph in which the vertices denotes states and labeled edges denote transitions on input. Given a finite language, a Loopless transition system can be constructed that will accept every valid string in the language and reject all other strings.

Proof (by construction):

- Language L is a finite set of finite strings - each composed of symbols in alphabet Σ.
- The root vertex of our loopless transition system will be the start.
- The depth *d* of our loopless transition system will be equal to the length of the longest string in L.
- Every vertex of our loopless transition system (except for those at depth *d*) will have a degree = |Σ|.
- Each edge coming from the each vertex will be labeled with a different symbol in Σ.
- Once our graph is constructed, we individually read each string in L starting at the root vertex and following the path of the string through the graph until we reach the end of the string. Whichever vertex we are at when we reach the end of the string is an 'accept' vertex.
- At this point, our loopless transition system can now read any string composed of symbols in alphabet Σ and we have traversed to an accept vertex, then the string is in language L, otherwise, it is not.

Limitations of finite domain models

A loopless transition system is incapable of handling any language whose regular expression contains a Kleene star, or any infinite languages.

Proof (by contradiction):

- Assume the loopless transition system *L* models the infinite language 1*.
- *L* has depth *d* and accepts any string composed of 1s of length *d* or smaller.
- The string composed of *d+1* 1s is also in the language 1*, but it will not be accepted by *L*.
- Therefore, a string in the language 1* is not accepted by *L*, but *L* models the language 1*. This is a contradiction; therefore, L cannot model the infinite language 1*.

Computing power of DFA

Any regular language (finite or infinite) can be modeled by a DFA. A DFA is a finite state machine

Proof (by construction):

- Language L is an infinite language modeled by a regular expression, which is composed of symbols in alphabet Σ.
- Build a start state, which is only an accept state if L contains the null string.
- Begin reading the regular expression from left to right.
  - If the regular expression implies that a specific symbol from Σ must be selected, then create a transition from your current state to a new state with that symbol as the label over the transition.

    - If the regular expression implies that a selection of specific symbols from Σ must be selected, then create a transition from your current state to a new state with those symbols comma delimited as the label over the transition.
    - If a portion of the regular expression can be iterated, or is encased in a Kleene star, then create transitions and states representing the sequence of symbols in the iterateable substring stemming from the current state. When reaching the end of the iterateable substring, add a transition back to the current state so that there is a loop.
    - Once you've reached the end of your regular expression, make the state you finished with an accepting state.

Limitations of DFA and finite state models

A DFA cannot model language which relies on counting to define a valid string in that language. For example, the language 0^k1^k will only accept strings with k 0s preceding k 1s. Such languages are not regular and by definition, DFAs can only model regular languages.

Proof (by Pumping Lemma):

- The Pumping Lemma states that if language L is an infinite regular language then $(\forall\ x \in L)\ (\exists\ u, v, w)$ s.t. $(uvw = x \wedge |v| > 0 \wedge (\forall k)\ uv^kw \in L)$. In other words, ever string in L can be broken down into three substrings, (u, v, and w). The middle substring (v) can be "pumped" any number of times with the result also in the language.
- If language L is defined by the regular expression 0^k1^k then v could be all 0s, all 1s or a combination of 0s and 1s.
    - If v were all 0s then we could pump It with a large enough k so that there are more 0s than 1s, so v is not all 0s.
    - If v were all 1s then we could pump it with a large enough k so that there are more 1s than 0s, so v is not all 1s.
    - If v is 0s and 1s, then pumping quickly produces a string long in L. For example, u=0, w=1, v=01. If k=1, the string produced of uv^kw is 0011. If k=2, the string produced of uv^kw is 001011, which is not in L.
    - Therefore, language L is not regular.
- We could not build a DFA to model language L because language L is not regular.

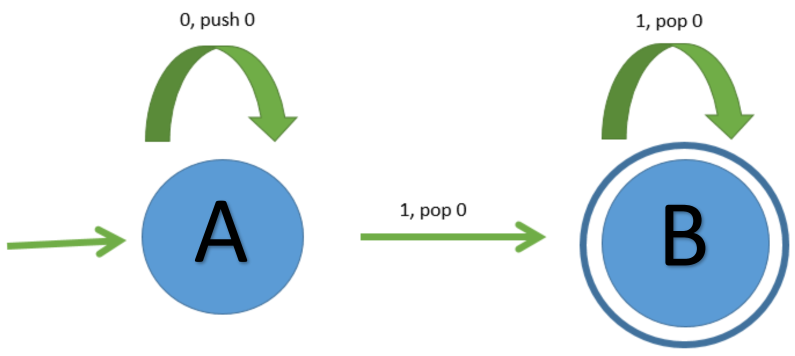
Computing power of PDA

By definition, PDAs model context free languages. I will prove by construction that all regular languages are also context free. Also by definition, DFAs model regular languages. Therefore, PDAs have at least as much computing power as DFAs.

Proof (by construction)

- DFA *D* models regular language L_p.
- Construct PDA *P* from *D*
    - Every state in *D* will be a state in *P*, so that *P* and *D* have the same start state and accept states as well.
    - Every transition in D will be a transition in P, but since P's transitions expect stack operations, you will make every transition in P include a 'no-op' stack operation.
- Since any DFA can be converted to a PDA, then every regular language is also a context free language.

- Proof regarding the Limitations of DFA addressed a language that cannot be modeled by a DFA. That language was the context free language 0^k1^k.
- The following DFA can model 0^k1^k:



- Since some CFL's are not regular, the PDA model is more expressive than DFA

Computing power of TM

A TM is at least as expressive as a PDA (in reality, it is more expressive than a PDA, but this proof will deal with TMs being as expressive as PDAs).

Proof (by construction) - One can construct a two tape TM that simulates PDA by doing the following:

- The set of states for TM = the set of states for PDA (including starting state and accepting states)
- The set of transitions for TM = the set of transitions for PDA except
    - For every stack popping operation of the PDA, let the TM travel to the rightmost cell of tape 2, read it, and replace it with '#'.
    - For every stack pushing operation of the PDA, let the TM travel to the rightmost cell of tape 2, travel one cell further to the right and replace the '#' with the symbol to be pushed.
    - In this way, tape 2 is acting like a stack.
- This will create a TM that simulates the any PDA and accepts the same language as the PDA.

Coding

Code a Finite Automaton

My program builds a finite automaton based on a definition provided in a text file. It is then able to evaluate strings against that finite automaton. There are a number of notable components to the program.

First, the program expects a comma separated file as input. In this file, each line is formatted the following way:

stateName, startState(1|0), acceptState(1|0), possibleInput, resultingNextStateName

*Note: There is no limit to the number of possibleInput-resultingNextStateName pairs on any given line.

Next, the program builds a finite automaton to represent the file in the *buildAutomatonFromFile* method. It does this by building an object of class State for every line in the file. Every state has a name, a boolean representing whether or not it's an accept state, and a map representing the possibleInput-resultingNextStateName pairs.

Once each state is built, it is put onto the static 'automaton' variable which maps state names to states and represents the whole finite automaton. Also, during this time, the start state is identified.

After the finite automaton is built, the program waits for the user to input a string to be evaluated against the finite automaton. In the *evaluateString* method, we set the currentState to be the startState. We then begin looping over each character in the string. Using the currentState and the current character in the string, we are able to determine the nextState. Each iteration of the loop determines the nextState and makes that nextState the currentState for the next iteration. When we've reached the end of the input string, we ask if the current state whether or not it's an accept state. If it is, the input string is 'ACCEPTED', otherwise it's 'REJECTED.'

This program allows the user to input as many strings as they'd like. The user would have to type 'exit' to quit the program. It also has some error handling to make sure that even if the user uses characters the finite automaton wasn't expecting to see, the program simply rejects the input string rather than throwing an error.

```java
public class Main {
  // map of state names to states
  private static Map<String, State> automaton = new HashMap<String, State>();
  private static State startState;
  private static BufferedReader bufferRead = new BufferedReader(new InputStreamReader(System.in));

  public static void main(String[] args) {
    // this is a comma separated file where each line is formatted the following way:
    // stateName,startState(1|0),acceptState(1|0),possibleInput,resultingStateName...
    // there is no limit on the number of possibleInput-resultingStateName pairs on any given line.
    String fileName = "automaton.txt";
    try {
        buildAutomatonFromFile(fileName);

        System.out.println("Provide a string to be evaluated by the provided automaton.");
        System.out.println("Type 'exit' to exit program.");

        // run the program until the user types 'exit'
        while(true) {
            String inputString = bufferRead.readLine();
            if (inputString.equals("exit")) {
                break;
            }
            evaluateString(inputString);
        }
    } catch(Exception e) {
        e.printStackTrace();
    }
 }

 private static void buildAutomatonFromFile(String fileName) throws FileNotFoundException{
    Scanner input = new Scanner(new File("atomaton.txt"));

    while (input.hasNextLine()) {
        String currentLine = input.nextLine();
        String[] stateInfo = currentLine.split(",");

        String stateName = stateInfo[0];
        boolean isStartState = stateInfo[1].equals("1");
        boolean acceptState = stateInfo[2].equals("1");

        State state = new State(stateName, acceptState);

        for (int i = 3; i < stateInfo.length; i=i+2) {
            char possibleInput = stateInfo[i].charAt(0);
            String resultingStateName = stateInfo[i+1];
            state.addNextState(possibleInput, resultingStateName);
            }
        automaton.put(stateName, state);

        if (isStartState) {
            startState = state;
        }
    }
}

  private static void evaluateString(String string) {
    try {
        State currentState = startState{
        for (int i = 0; i < string.length(); i++) {
            Character input = string.charAt(i);
            currentState = automaton.get(currentState.getNextState(input));
        }
        System.out.println(currentState.isAcceptState() ? "ACCEPTED" : "REJECTED");
    } catch (NullPointerException e) {
        // nullPointerException is thrown when the input does not have a defined next state
        // meaning the input string will NOT end up in an accept state
        System.out.println("REJECTED");
    }
  }

}
```

```java
public class State {

    private String stateName;
    private boolean acceptState;
    // a map from next input to the next state name.
    private Map<Character, String> nextStates = new HashMap<Character, String>();

    public State(String stateName, boolean acceptState) {
        this.stateName = stateName;
        this.acceptState = acceptState;
    }

    public String getStateName() {
        return stateName;
    }

    public void setStateName(String stateName) {
        this.stateName = stateName;
    }

    public boolean isAcceptState() {
        return acceptState;
    }

    public void setAcceptState(boolean acceptState) {
        this.acceptState = acceptState;
    }

    public Map<Character, String> getNextStates() {
        return nextStates;
    }

    public void setNextStates(Map<Character, String> nextStates) {
        this.nextStates = nextStates;
    }

    public void addNextState(Character input, String name) {
        nextStates.put(input, name);
    }

    public String getNextState(Character input) {
        return nextStates.get(input);
    }
}
```

First test case: finite automaton that accepts only the language of 0*

Automaton.txt

```
A,1,1,0,A
```

Test output:

```
Provide a string to be evaluated by the provided automaton.
Type 'exit' to exit program.
0000000
ACCEPTED
1000000
REJECTED
0000001
REJECTED
010101
REJECTED

REJECTED
0
ACCEPTED
1111111
REJECTED
randomteststring
REJECTED
Exit
```

Second test case: finite automaton that accepts only the language where no 0 precedes a 1

Automaton.txt

```
A,1,1,0,B,1,A
B,0,1,0,B
```

Test output:

```
Provide a string to be evaluated by the provided automaton.

Type 'exit' to exit program.

000000000

ACCEPTED

111111111

ACCEPTED

11110000

ACCEPTED

0111111

REJECTED

01

REJECTED

10

ACCEPTED

11111101

REJECTED

100000001

REJECTED


ACCEPTED

exit
```

Third test case: finite automaton that accepts only the language with only one 0

Automaton.txt

```
A,1,0,0,B,1,A

B,0,1,1,B
```

Test output:

```
Provide a string to be evaluated by the provided automaton.

Type 'exit' to exit program.

0

ACCEPTED

1

REJECTED

10

ACCEPTED

100

REJECTED

1111

REJECTED

11101111

ACCEPTED

1000000001

REJECTED

00100

REJECTED


REJECTED

Exit
```

Fourth test case (for fun): Finite Automaton that only accepts my husband's and my "Celebrity Couple Name" (that is, a mashup of my husband's name – Hunter, and my name – Nicole. This demonstrates that my finite state machine can use any sort of name for the states (rather than just A,B,C… etc) and can use any inputs (rather than just 0,1).

Automaton.txt

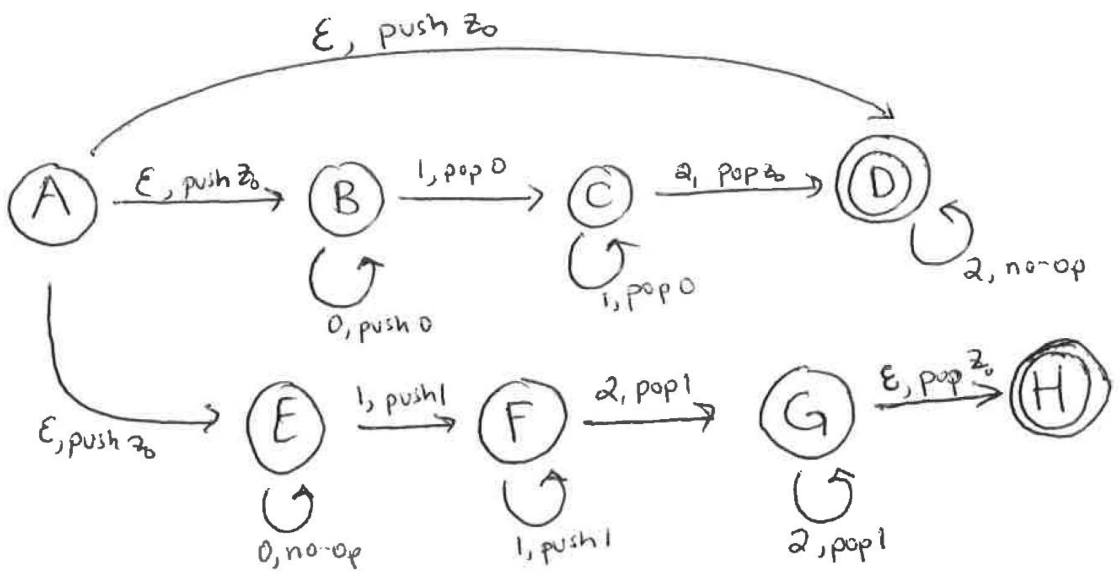
```
State A,1,0,N,State B,H,State B
State B,0,0,i,State C,u,State C
State C,0,0,c,State D,n,State D
State D,0,0,o,State E,t,State E
State E,0,0,l,State F,e,State F
State F,0,0,e,State G,r,State G
State G,0,1
```

Test output:

```
Provide a string to be evaluated by the provided automaton.

Type 'exit' to exit program.

N

REJECTED

Nicole

ACCEPTED

Hunter

ACCEPTED

Hunole

ACCEPTED

Nicter

ACCEPTED

Hicole

ACCEPTED

Hucole

ACCEPTED

Hinoee

ACCEPTED

Anything

REJECTED

Hunterr

REJECTED
```

Code a Push Down Automaton with a Stack Machine

My PDA accepts the language {0^m 1^n 2^p | m=n or n=p}. The user inputs a string to be evaluated by the program and the program outputs 'ACCEPTED' if it conforms to the defined grammar, and it outputs 'REJECTED' if it does not. The pictographic representation of the PDA which accepts this language follows:



The *PDA.buildAutomaton()* function builds one state object for each of the states above. Each state contains a map of 'transitions' which (given the current state) map the current input character and the symbol at the top of the stack to a stack operation and the next state.

To evaluate the user provided input string, the function, *isAcceptedWhenStartingFromState()*, is given a start state and the input string. It loops over the input string and for each character in the input string, it performs both the stack operations and the state transition associated with that 'current state-current input character-current top stack symbol' tuple. So each iteration of the loop manipulates the stack and updates the current state.

My PDA is non-deterministic, so each input could go down one of three possible paths in the automaton. If any of those paths terminate in an accept state, then the input string is accepted.

```java
public class Main {

  private static BufferedReader bufferRead = new BufferedReader(new InputStreamReader(System.in));

  public static void main(String[] args) {
    try {

      PDA.buildAutomaton();

      System.out.println("Provide a string to be evaluated by the provided automaton.");
      System.out.println("Type 'exit' to exit program.");

      // run the program until the user types 'exit'
      while(true) {
        String inputString = bufferRead.readLine();
        if (inputString.equals("exit")) {
           break;
        }

        // my PDA is non-deterministic, so the 'startState' A simply
        // puts the start symbol onto the stack and does a
        // lambda transition into states B, D, or E.
        if( isAcceptedWhenStartingFromState("B", inputString) ||
            isAcceptedWhenStartingFromState("D", inputString) ||
            isAcceptedWhenStartingFromState("E", inputString)) {
          System.out.println("ACCEPTED");
        } else {
          System.out.println("REJECTED");
        }
      }

    } catch(Exception e) {
      e.printStackTrace();
    }
  }

  private static boolean isAcceptedWhenStartingFromState(String stateName, String inputString) {

    State currentState = PDA.automaton.get(stateName);
    PDA.stack = new Stack();

    // begin iterating over the input string and performing state transitions
    // (and stack operations)
    for (int i = 0; i < inputString.length(); i++){
        currentState = PDA.performTransition(inputString.charAt(i), currentState);
    }

    // once i'm done iterating, I can answer the question of whether or
    // not this input is accepted by evaluating that this path landed me
    //in an accept state AND the stack is empty.
    return (PDA.stack.isEmpty() && currentState != null && currentState.isAcceptState());
  }
}
```

```
public class PDA {

    static Stack stack;

    // My PDA accepts the language {0^m 1^n 2^p | m=n or n=p}
    public static Map<String, State> automaton = new HashMap<String, State>();

    public static void buildAutomaton(){

        State state = new State(false);
        Map<String, Transition> transitions = new HashMap<String, Transition>();
        transitions.put("0" + Stack.stackStart, new Transition(Stack.StackOperation.PUSH, "B"));
        transitions.put("0" + "0", new Transition(Stack.StackOperation.PUSH, "B"));
        transitions.put("1" + "0", new Transition(Stack.StackOperation.POP, "C"));
        state.setTransitions(transitions);
        automaton.put("B", state);

        state = new State(true);
        transitions = new HashMap<String, Transition>();
        transitions.put("1" + "0", new Transition(Stack.StackOperation.POP, "C"));
        transitions.put("2" + Stack.stackStart, new Transition(Stack.StackOperation.NONE,
        "D"));
        state.setTransitions(transitions);
        automaton.put("C", state);

        state = new State(true);
        transitions = new HashMap<String, Transition>();
        transitions.put("2" + Stack.stackStart, new Transition(Stack.StackOperation.NONE,
        "D"));
        state.setTransitions(transitions);
        automaton.put("D", state);

        state = new State(true);
        transitions = new HashMap<String, Transition>();
        transitions.put("0" + Stack.stackStart, new Transition(Stack.StackOperation.NONE,
        "E"));
        transitions.put("1" + Stack.stackStart, new Transition(Stack.StackOperation.PUSH,
        "F"));
        state.setTransitions(transitions);
        automaton.put("E", state);

        state = new State(false);
        transitions = new HashMap<String, Transition>();
        transitions.put("1" + Stack.stackStart, new Transition(Stack.StackOperation.PUSH,
        "F"));
        transitions.put("1" + "1", new Transition(Stack.StackOperation.PUSH, "F"));
        transitions.put("2" + "1", new Transition(Stack.StackOperation.POP, "G"));
        state.setTransitions(transitions);
        automaton.put("F", state);

        state = new State(true);
        transitions = new HashMap<String, Transition>();
        transitions.put("2" + "1", new Transition(Stack.StackOperation.POP, "G"));
        state.setTransitions(transitions);
        automaton.put("G", state);

    }
    public static State performTransition(char input, State currentState) {

        /*
         determining which will be the next transition will require three pieces of information
          1. the next input character
          2. the symbol at the top of the stack
          3. the current state

         each state knows which stack operation to perform and what the next state will be given
         the next input character and the symbol at the top of the stack so the
         'currentSituation' below is a string representing of that information to be fed to
         the current state to help determine our next steps.
        */

        String topStackSymbol = stack.getTop();
        String currentSituation = input + topStackSymbol;

        // if the currentState is null, then we have somehow landed ourselves off
        // the PDA and can never get into an accept state on this path.
        if (currentState != null) {

            // if the currentState does not know what the transition would be give then
            // currentSituation, then we will never get into an accept state on this path
            // and will return a null currentState
            Transition nextTransition = currentState.getTransitions().get(currentSituation);

            if (nextTransition != null) {
                // this state knows which transition to follow next.

                // perform the appropriate stack operation as defined by the transition
                stack.performOperation(nextTransition.getStackOperation(), input + "");

                // move to the next state so that the next state is now the current state
                currentState = automaton.get(nextTransition.getNextStateName());

            } else {
                return null;
            }
        }

        return currentState;
    }
}
```

```
public class State {

    // if the stack is empty and we end in this state, is the input string accepted?
    private boolean acceptState;
    private Map<String, Transition> transitions = new HashMap<String, Transition>();

    public State(boolean acceptState) {
        this.acceptState = acceptState;
    }

    public boolean isAcceptState() {
        return acceptState;
    }

    public Map<String, Transition> getTransitions() {
        return transitions;
    }

    public void setTransitions(Map<String, Transition> transitions) {
        this.transitions = transitions;
    }

}


public class Transition {

    private Stack.StackOperation stackOperation;
    private String nextStateName;

    public Transition(Stack.StackOperation stackOperation, String nextStateName) {
        this.stackOperation = stackOperation;
        this.nextStateName = nextStateName;
    }

    public Stack.StackOperation getStackOperation() {
        return stackOperation;
    }

    public String getNextStateName() {
        return nextStateName;
    }

}
```

```
public class Stack {

    static String stackStart = "\u03B3";
    private Node top;

    public Stack() {
        push(stackStart);
    }

    private class Node {
        private String item;
        private Node next;
    }

    public enum StackOperation {
        POP,
        PUSH,
        NONE
    }

    public String performOperation(StackOperation stackOperation, String input) {
        switch (stackOperation) {
            case POP:
                return pop();
            case PUSH:
                push(input);
            case NONE:
            default:
                return "";
        }
    }

    public String getTop() {
        return top.item;
    }

    public boolean isEmpty() {
        return stackStart.equals(top.item);
    }

    public void push(String item) {
        Node oldfirst = top;
        top = new Node();
        top.item = item;
        top.next = oldfirst;
    }

    public String pop() {
        if (isEmpty()) throw new RuntimeException("Stack underflow");
        String item = top.item;
        top = top.next;
        return item;
    }

}
```

Test output: *My PDA accepts the language {0^m 1^n 2^p | m=n or n=p}*

```
Provide a string to be evaluated by the provided automaton.

Type 'exit' to exit program.

0
ACCEPTED

1
REJECTED

2
ACCEPTED

01
ACCEPTED

012
ACCEPTED

011222
REJECTED

0112
REJECTED

12
ACCEPTED

02
REJECTED

000012
ACCEPTED

011112222
ACCEPTED

0122
ACCEPTED

00000
ACCEPTED

222222
ACCEPTED

exit
```

Research (TODO: lexical analyzer state transition)

Demonstration of a Simple Recursive Descent Parser in Java

For my research project, I have demonstrated a simple recursive descent parser using Java. In order to build my parser, there were a couple things I needed to accomplish. I first built a lexical analyzer to identify the substrings of the source program and group the lexemes according to their tokens. My lexical analyzer recognizes the following tokens (In the program they are called *LexicalCategories*):

INTEGER_LITERAL
STRING_LITERAL
CHARACTER_LITERAL
FLOATING_POINT_LITERAL
IDENTIFIER
OPEN_PAREN
CLOSE_PAREN
SEMICOLON
BINARY_ARITHMETIC_OPERATOR
BINARY_RELATIONAL_OPERATOR
BINARY_LOGICAL_OPERATOR
BITWISE_LOGICAL_OPERATOR
UNARY_LOGICAL_OPERATOR

I built this lexical analyzer by designing a state transition diagram that describes the tokens of the input language. This state transition system is diagramed in Appendix A at the end of this project. In order to simplify the state transition system, I grouped possible symbols in the source program. For example, when recognizing an INTEGER_LITERAL or a FLOATING_POINT_LITERAL, all digits are equivalent. I grouped all input symbols according to the following *InputCategories:*

DIGIT,
CHARACTER,
DOT,
DOUBLE_QUOTE,
SINGLE_QUOTE,
UNDERSCORE,
OPEN_PAREN,
CLOSE_PAREN,
ARITHMETIC_SYMBOL, // +, -, *, /, %
LOGICAL_SYMBOL, // &, |
RELATIONAL_SYMBOL, // >, <,
BANG, // !
EQUALS, // =
SEMICOLON

In general, recursive decent parsers are top-down parsers built from a set of mutually-recursive procedures where each such procedure implements one of the production rules of the grammar. Thus the structure of the resulting program closely mirrors that of the grammar it recognizes. There is a function for each nonterminal in the grammar, which can parse sentences that can be generated by that nonterminal.

Once the lexical analyzer has finished processing the source program, a map from lexeme to token is built in the Main class called *lex*. My program loops over the *lex,* parses and evaluates the validity of the source program according to the context-free grammar I define which is expressed using BNF notation. It validates source programs consisting of one statement, terminating with a semicolon. The statements could be an arithmetic expression, a relational expression, a logical expression, or a string (and string concatenation). My context-free grammar is diagramed in Appendix B at the end of this project.

My recursive decent parser performs a leftmost derivation, meaning it traces a 'parse tree' from left to right starting at a root. I put 'parse tree' in quotes because the output of my program is not formatted visually like a tree, even though it could be interpreted that way. By convention, my recursive decent parser also does look ahead. This means, in order to determine which production rule to follow when more than one is possible for a non-terminal, it considers the current token and the next token together.

I have included test output of my RD Parser as Appendix C at the end of this project.

Known limitations to my RD Parser include:

- The arithmetic functions do not deal with negative numbers. Any attempts to input an arithmetic statement will result in a failure to parse.
- Inability to parse multiple statements terminating with a semicolon at once.
- None of the unary arithmetic operators are handled by this lexical analyzer / parser.

Code

```
public class Main {

    // This parser will parse numeric, logical, and relational expressions

    private static String javaProgram = "";
    public static List<Object[]> lex = new ArrayList<>();

    public static void main(String[] args) {
        readJavaProgramForParsing();
        try {
            LexicalAnalyzerDFA.buildLex(javaProgram);
            printLex();
            boolean successful = ParsingPDA.buildParseTree();
            System.out.println(successful ? "Parsed Successfully." : "Unable to parse.");

        } catch (FailedToParseException e) {
            System.out.println("There was an error parsing. " + e.getMessage());
        }
    }

    public static void readJavaProgramForParsing() {
        try {
            Scanner input = new Scanner(new File("testFile.java"));
            while (input.hasNextLine()) {
                javaProgram += input.nextLine();
            }
            javaProgram = javaProgram.replaceAll("\\s","");
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }

    public static void printLex() {
        for(Object[] tuple : lex) {
            System.out.println(tuple[0] + " -> " + tuple[1]);
        }
    }
}


public class FailedToParseException extends Exception {

    public FailedToParseException(String message) {
        super(message);
    }
}


public class NoTerminalSymbol extends Exception {

    public NoTerminalSymbol(String message) {
        super(message);
    }
}
```

```
public enum InputCategory {
    DIGIT,
    CHARACTER,
    DOT,
    DOUBLE_QUOTE,
    SINGLE_QUOTE,
    UNDERSCORE,
    OPEN_PAREN,
    CLOSE_PAREN,
    ARITHMETIC_SYMBOL, // +, -, *, /, %
    LOGICAL_SYMBOL,    // &, |
    RELATIONAL_SYMBOL, // >, <,
    BANG,              // !
    EQUALS,            // =
    SEMICOLON          // ;
}
```

```
public enum LexicalCategory {
    NONE,
    INTEGER_LITERAL,
    STRING_LITERAL,
    CHARACTER_LITERAL,
    FLOATING_POINT_LITERAL,
    IDENTIFIER,
    OPEN_PAREN,
    CLOSE_PAREN,
    SEMICOLON,
    BINARY_ARITHMETIC_OPERATOR,
    BINARY_RELATIONAL_OPERATOR,
    BINARY_LOGICAL_OPERATOR,
    BITWISE_LOGICAL_OPERATOR,
    UNARY_LOGICAL_OPERATOR,
    ASSIGNMENT
}
```

```java
public class LexicalAnalyzerDFA {

    public static int numberOfDoubleQuotes = 0;
    public static int numberOfSingleQuotes = 0;

    // This method groups the input symbols by their category for simpler transitions
    public static InputCategory getInputCategory(char a) {

        if (Character.isDigit(a)){
            return InputCategory.DIGIT;
        }
        if (Character.isLetter(a)){
            return InputCategory.CHARACTER;
        }
        if (a == '_'){
            return InputCategory.UNDERSCORE;
        }
        if (a == '.'){
            return InputCategory.DOT;
        }
        if (a == '\''){
            numberOfSingleQuotes++;
            return InputCategory.SINGLE_QUOTE;
        }
        if (a == '"'){
            numberOfDoubleQuotes++;
            return InputCategory.DOUBLE_QUOTE;
        }
        if (a == '+' ||
            a == '-' ||
            a == '*' ||
            a == '/' ||
            a == '%') {
            return InputCategory.ARITHMETIC_SYMBOL;
        }
        if (a == '<' ||
            a == '>') {
            return InputCategory.RELATIONAL_SYMBOL;
        }
        if (a == '&' ||
            a == '|') {
            return InputCategory.LOGICAL_SYMBOL;
        }
        if (a == '!') {
            return InputCategory.BANG;
        }
        if (a == '(') {
            return InputCategory.OPEN_PAREN;
        }
        if (a == ')') {
            return InputCategory.CLOSE_PAREN;
        }
        if (a == '=') {
            return InputCategory.EQUALS;
        }
        if (a == ';') {
            return InputCategory.SEMICOLON;
        }

        return null;
    }
```

```
// this method takes the inputCategory of the current symbol from the source program,
// The token we are currently in the middle of processing, and the all the characters already
// processed in the token we are currently in the middle of processing. It returns the
// LexicalCategory that the current character should be a part of.
// Still part of class: LexicalAnalyzerDFA

public static LexicalCategory getLexicalCategory(LexicalCategory currentLexicalCategory,
                                                 InputCategory inputCategory,
                                                 String currentToken)
                                                 throws FailedToParseException {


        // take care of STRING_LITERAL and CHARACTER_LITERAL first because
        // any sort of input can be in a string or char.
        if (currentLexicalCategory.equals(LexicalCategory.STRING_LITERAL) &&
                numberOfDoubleQuotes % 2 == 1){
            return LexicalCategory.STRING_LITERAL;
        }
        if (currentLexicalCategory.equals(LexicalCategory.CHARACTER_LITERAL) &&
                numberOfSingleQuotes % 2 == 1){
            return LexicalCategory.CHARACTER_LITERAL;
        }
        // check the char is not empty - not allowed in java
        if (currentLexicalCategory.equals(LexicalCategory.CHARACTER_LITERAL) &&
                numberOfSingleQuotes % 2 == 0 &&
                currentToken.equals("''")) {
            throw new FailedToParseException("empty character literal");
        }

        // since the input is not part of a STRING_LITERAL or CHARACTER_LITERAL:
        switch (inputCategory) {

            case DIGIT:
                switch (currentLexicalCategory) {
                    case IDENTIFIER:
                        return LexicalCategory.IDENTIFIER;
                    case FLOATING_POINT_LITERAL:
                        return LexicalCategory.FLOATING_POINT_LITERAL;
                    default:
                        return LexicalCategory.INTEGER_LITERAL;
                }

            case CHARACTER:
                return LexicalCategory.IDENTIFIER;

            case SINGLE_QUOTE:
                return LexicalCategory.CHARACTER_LITERAL;

            case DOUBLE_QUOTE:
                return LexicalCategory.STRING_LITERAL;

            case DOT:
                // maybe I'll eventually deal with functions?
                return LexicalCategory.FLOATING_POINT_LITERAL;

            case UNDERSCORE:
                return LexicalCategory.IDENTIFIER;

            case OPEN_PAREN:
                return LexicalCategory.OPEN_PAREN;

            case CLOSE_PAREN:
                return LexicalCategory.CLOSE_PAREN;

            case SEMICOLON:
```

```java
                return LexicalCategory.SEMICOLON;

            case BANG:
                return LexicalCategory.UNARY_LOGICAL_OPERATOR;

            case LOGICAL_SYMBOL:
                return (currentToken.equals("&") || currentToken.equals("|")) ?
                         LexicalCategory.BINARY_LOGICAL_OPERATOR :
                         LexicalCategory.BITWISE_LOGICAL_OPERATOR;

            case RELATIONAL_SYMBOL:
                return LexicalCategory.BINARY_RELATIONAL_OPERATOR;

            case EQUALS:
                if (currentToken.equals("=") || currentToken.equals("!") ||
                        currentToken.equals(">") || currentToken.equals("<")){
                    return LexicalCategory.BINARY_RELATIONAL_OPERATOR;
                }
                return LexicalCategory.ASSIGNMENT;

            case ARITHMETIC_SYMBOL:
                return LexicalCategory.BINARY_ARITHMETIC_OPERATOR;

            default:
                throw new FailedToParseException("I haven't been told how to interpret " +
                                                                           inputCategory);
        }
    }
```

```
// This method returns true if the next character marks the beginning of a new lexeme, or is
// part of the lexeme preceeding it.
// Still part of class: LexicalAnalyzerDFA

    public static boolean isFirstCharacterOfNewLexeme(LexicalCategory currentLexicalCategory,
                                                      LexicalCategory futureLexicalCategory,
                                                      char nextCharacter,
                                                      InputCategory currentInputCategory,
                                                      String currentToken)
                                                      throws FailedToParseException {

        switch (currentLexicalCategory) {
            case NONE:
                return true;

            case STRING_LITERAL:
                return !futureLexicalCategory.equals(LexicalCategory.STRING_LITERAL) ||
                        (numberOfDoubleQuotes % 2 == 1 &&
                                 currentInputCategory.equals(InputCategory.DOUBLE_QUOTE));

            case CHARACTER_LITERAL:
                if (currentToken.length() > 3) {
                          throw new FailedToParseException("unclosed character literal");
                 }
                return !futureLexicalCategory.equals(LexicalCategory.CHARACTER_LITERAL) ||
                        (numberOfSingleQuotes % 2 == 1 &&
                         currentInputCategory.equals(InputCategory.SINGLE_QUOTE));

            case INTEGER_LITERAL:
                return !futureLexicalCategory.equals(LexicalCategory.FLOATING_POINT_LITERAL)
                         && !currentLexicalCategory.equals(futureLexicalCategory);

            case CLOSE_PAREN:
            case OPEN_PAREN:
            case SEMICOLON:
            case BINARY_ARITHMETIC_OPERATOR:
                return true;

            case BITWISE_LOGICAL_OPERATOR:
                return (!currentToken.equals("" + nextCharacter) ||
                 !futureLexicalCategory.equals(LexicalCategory.BINARY_LOGICAL_OPERATOR));

            case UNARY_LOGICAL_OPERATOR:
                return
                 (!futureLexicalCategory.equals(LexicalCategory.BINARY_RELATIONAL_OPERATOR));

            case BINARY_LOGICAL_OPERATOR:
            case BINARY_RELATIONAL_OPERATOR:
                return (currentToken.length() > 2) ||
                           !currentLexicalCategory.equals(futureLexicalCategory);

            default:
                return !currentLexicalCategory.equals(futureLexicalCategory);
        }
    }
```

```java
// Still part of class: LexicalAnalyzerDFA
// builds 'lex' object in Main class
public static void buildLex(String javaProgram) throws FailedToParseException {
        // begin at the start state
        LexicalCategory currentLexicalCategory = LexicalCategory.NONE;
        String currentToken = "";

        for (int i = 0; i < javaProgram.length(); i++) {

            // get the next input
            char nextCharacter = javaProgram.charAt(i);
            // figure out what kind of input it is
            InputCategory inputCategory = LexicalAnalyzerDFA.getInputCategory(nextCharacter);

            // given the current state (i.e. lexicalCategory), the type of input, and the
            // inputs that precede this input tell me what the next state (lexicalCategory)
            // will be.
            LexicalCategory nextLexicalCategory =
                          LexicalAnalyzerDFA.getLexicalCategory(currentLexicalCategory,
                          inputCategory,
                          currentToken);

            // figure out if this new state marks the beginning of a new token/lexeme or if,
            // rather, the current token/lexeme should change its classification
            // for example : and INTEGER_LITERAL should change its classification to
            //                FLOATING_POINT_LITERAL if the next character is a DOT
            boolean firstCharOfNewToken =
                 LexicalAnalyzerDFA.isFirstCharacterOfNewLexeme(currentLexicalCategory,
                                                                 nextLexicalCategory,
                                                                 nextCharacter,
                                                                 inputCategory,
                                                                 currentToken);

            // if the next input marks the beginning of a new token/lexeme
            if (firstCharOfNewToken && !currentLexicalCategory.equals(LexicalCategory.NONE))
{
                // then take the token you've been building and add it to the lex, along with
                // its corresponding lexicalCategory.
                Object[] tokenLexeme = {currentToken, currentLexicalCategory};
                Main.lex.add(tokenLexeme);
                // then use the next input to begin building the new token/lexeme
                currentToken = "" + nextCharacter;
            } else {
                // if the next input does NOT mark the beginning of a new token/lexeme
                // then add it to the token you've been building.
                currentToken += nextCharacter;
            }

            // if the next input is the last symbol in the program
            if (i == javaProgram.length() - 1) {
                // it has already been added to the token you've been building in the
                // previous if/else block so add the token and next lexicalCategory to the
                // lex one last time.
                Object[] tokenLexeme = {currentToken, nextLexicalCategory};
                Main.lex.add(tokenLexeme);
            }

            // update the currentLexicalCategory to reflect which state the most
            // recent input put us in.
            currentLexicalCategory = nextLexicalCategory;
        }
    }
}
```

```
public class ParsingPDA {

    public static int counter = 0;
    public static Object[] currentLexicalCategory;
    public static Object[] nextLexicalCategory;

    private static void setCurrentNextSymbols() {
        currentLexicalCategory = Main.lex.get(counter);
        if (counter + 1 < Main.lex.size()) nextLexicalCategory = Main.lex.get(counter + 1);
        counter ++;
    }

    public static boolean buildParseTree() {
        setCurrentNextSymbols();

        try {
            program();
            return true;
        } catch (NoTerminalSymbol e) {
            e.printStackTrace();
            return false;
        }

    }

    public static void program() throws NoTerminalSymbol {
        try {
            System.out.println("program");
            while (true) {
                exp();
            }
        } catch (NoTerminalSymbol e) {
            semicolon();
        }

    }

    private static void exp() throws NoTerminalSymbol {
        System.out.println("exp");
        try {
            mathExp();
        } catch (NoTerminalSymbol e) {
            try {
                logicalExp();
            } catch (NoTerminalSymbol e2) {
                stringExp();
            }
        }
    }

    private static void lex() {
        System.out.println(currentLexicalCategory[0] + " --> " + currentLexicalCategory[1]);
        setCurrentNextSymbols();
    }

    private static void semicolon() throws NoTerminalSymbol {
        if (currentLexicalCategory[1].equals(LexicalCategory.SEMICOLON)) {
                 System.out.println("SEMICOLON");
         }
        else throw new NoTerminalSymbol(
                            "expected a semicolon to terminate string. it didn't.");
    }
```

```
////////////   arithmetic expressions (part of class: ParsingPDA) //////////////////////

private static void mathExp() throws NoTerminalSymbol{
    System.out.println("mathExp");
    if (nextLexicalCategory[1].equals(LexicalCategory.BINARY_LOGICAL_OPERATOR) ||
            nextLexicalCategory[1].equals(LexicalCategory.BINARY_RELATIONAL_OPERATOR) ||
            nextLexicalCategory[1].equals(LexicalCategory.BITWISE_LOGICAL_OPERATOR)) {
        throw new NoTerminalSymbol(
             "There is a non-mathematical operator in the expression");
    } else if (nextLexicalCategory[0].equals("+") ||
            nextLexicalCategory[0].equals("-")){
        multMathExp();
        lex();
        multMathExp();
    } else {
        multMathExp();
    }
}

private static void multMathExp() throws NoTerminalSymbol{
    System.out.println("multMathExp");
    if (nextLexicalCategory[0].equals("*") ||
            nextLexicalCategory[0].equals("/")){
        subMathExp();
        lex();
        subMathExp();
    } else {
        subMathExp();
    }
}

private static void subMathExp() throws NoTerminalSymbol{
    System.out.println("subMathExp");
    if (currentLexicalCategory[1].equals(LexicalCategory.OPEN_PAREN)) {
        lex();
        mathExp();
        lex();
    } else if (currentLexicalCategory[1].equals(LexicalCategory.IDENTIFIER) ||
            currentLexicalCategory[1].equals(LexicalCategory.FLOATING_POINT_LITERAL) ||
            currentLexicalCategory[1].equals(LexicalCategory.INTEGER_LITERAL) ||
            currentLexicalCategory[1].equals(LexicalCategory.CHARACTER_LITERAL)) {
        lex();
    } else {
        throw new NoTerminalSymbol("There is no math terminal symbol");
    }
}
```

```
////////////   string expressions  (part of class: ParsingPDA) ////////////////////////

private static void stringExp() throws NoTerminalSymbol{
    System.out.println("stringExp");
    if (nextLexicalCategory[1].equals(LexicalCategory.BINARY_LOGICAL_OPERATOR) ||
            nextLexicalCategory[1].equals(LexicalCategory.BINARY_RELATIONAL_OPERATOR) ||
            nextLexicalCategory[1].equals(LexicalCategory.BITWISE_LOGICAL_OPERATOR) ||
            nextLexicalCategory[1].equals(LexicalCategory.BINARY_ARITHMETIC_OPERATOR)) {
        if (nextLexicalCategory[0].equals("+")) {
            string();
            lex();
            string();
        } else {
            throw new NoTerminalSymbol(
                                "There is a non-string operator in the expression");
        }
    } else {
        string();
    }
}

private static void string() throws NoTerminalSymbol{
    System.out.println("string");
    if (currentLexicalCategory[1].equals(LexicalCategory.OPEN_PAREN)) {
        lex();
        mathExp();
        lex();
    } else if (currentLexicalCategory[1].equals(LexicalCategory.IDENTIFIER) ||
            currentLexicalCategory[1].equals(LexicalCategory.FLOATING_POINT_LITERAL) ||
            currentLexicalCategory[1].equals(LexicalCategory.INTEGER_LITERAL) ||
            currentLexicalCategory[1].equals(LexicalCategory.CHARACTER_LITERAL) ||
            currentLexicalCategory[1].equals(LexicalCategory.STRING_LITERAL)){
        lex();
    } else {
        throw new NoTerminalSymbol("There is no string terminal symbol");
    }
}
```

```
////////////    logical expressions  (part of class: ParsingPDA) //////////////////////

private static void logicalExp() throws NoTerminalSymbol{
    System.out.println("logicalExp");
    if (nextLexicalCategory[0].equals("||")){
        andExp();
        lex();
        andExp();
    } else {
        andExp();
    }
}

private static void andExp() throws NoTerminalSymbol{
    System.out.println("andExp");
    if (nextLexicalCategory[0].equals("&&")){
        bitExp();
        lex();
        bitExp();
    } else {
        bitExp();
    }
}

private static void bitExp() throws NoTerminalSymbol{
    System.out.println("bitExp");
    if (nextLexicalCategory[1].equals(LexicalCategory.BITWISE_LOGICAL_OPERATOR)){
        subLogExp();
        lex();
        subLogExp();
    } else {
        subLogExp();
    }
}

private static void subLogExp() throws NoTerminalSymbol{
    System.out.println("subLogExp");
    if (currentLexicalCategory[0].equals("!")) {
        lex();
        logicalExp();
    } else if (currentLexicalCategory[1].equals(LexicalCategory.OPEN_PAREN)) {
        lex();
        logicalExp();
        lex();
    } else if (currentLexicalCategory[1].equals(LexicalCategory.IDENTIFIER)) {
        lex();
    } else {
        relExp();
    }
}
```

```
////////////////////   relational expressions  (part of class: ParsingPDA) /////////////////
    private static void relExp() throws NoTerminalSymbol{
        System.out.println("relExp");
        if (nextLexicalCategory[0].equals("==") ||
                nextLexicalCategory[0].equals("!=")){
            equalsRelExp();
            lex();
            equalsRelExp();
        } else {
            equalsRelExp();
        }
    }

    private static void equalsRelExp() throws NoTerminalSymbol{
        System.out.println("equalsRelExp");
        if (nextLexicalCategory[1].equals(LexicalCategory.BINARY_RELATIONAL_OPERATOR)) {
            subRelExp();
            lex();
            subRelExp();
        } else {
            subRelExp();
        }
    }

    private static void subRelExp() throws NoTerminalSymbol{
        System.out.println("subRelExp");
        if (currentLexicalCategory[1].equals(LexicalCategory.OPEN_PAREN)) {
            lex();
            relExp();
            lex();
        } else if (currentLexicalCategory[1].equals(LexicalCategory.IDENTIFIER) ||
                currentLexicalCategory[1].equals(LexicalCategory.CHARACTER_LITERAL) ||
                currentLexicalCategory[1].equals(LexicalCategory.INTEGER_LITERAL) ||
                currentLexicalCategory[1].equals(LexicalCategory.FLOATING_POINT_LITERAL)) {
            lex();
        } else {
            throw new NoTerminalSymbol("no terminal logical or relational symbol");
        }
    }
```

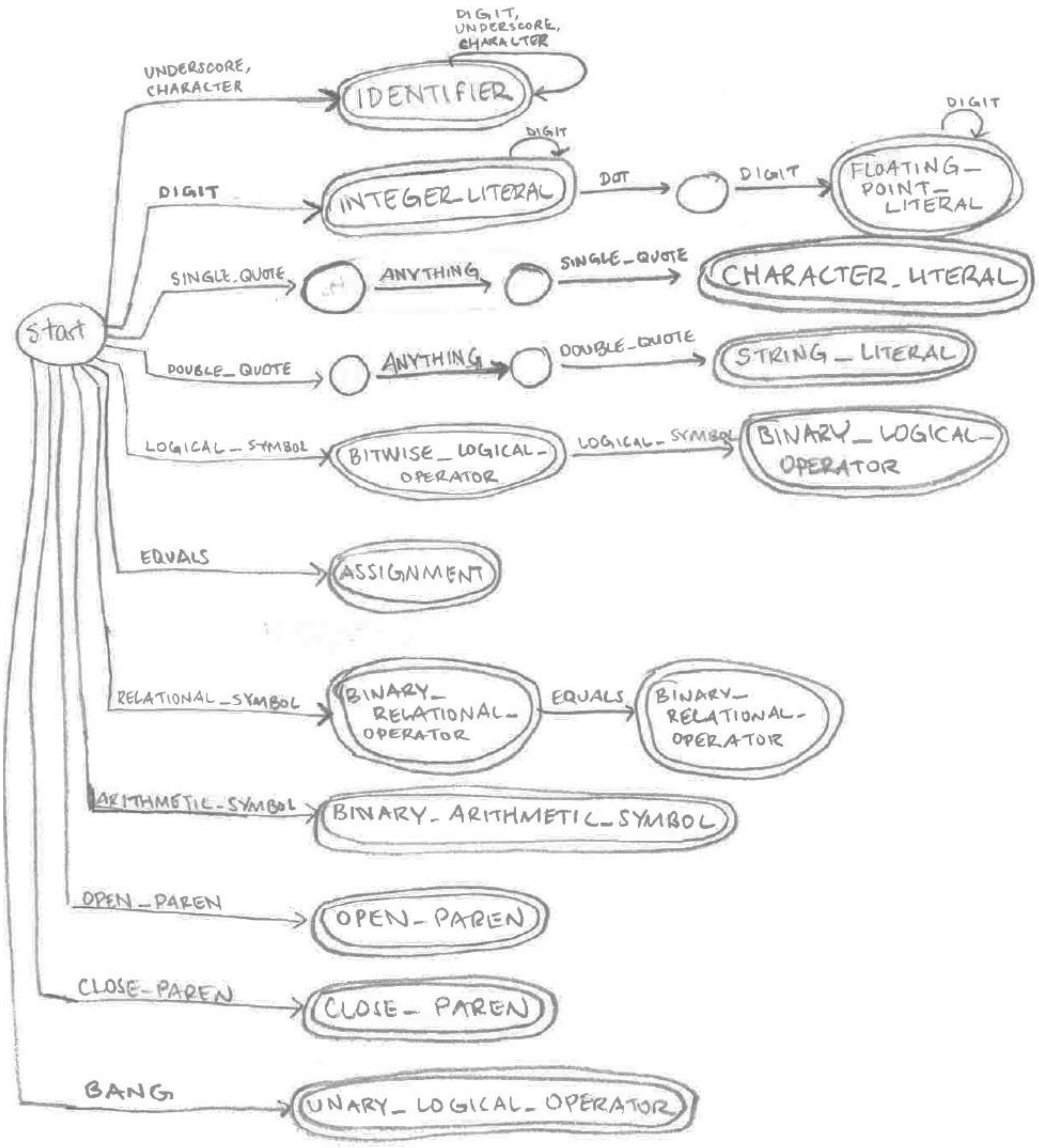
References

Power, James "Notes on Formal Language Theory and Parsing" National University of Ireland, Maynooth, Nov 29, 2002

Sikkel, Klaas, and Anton Nijholt. "Parsing of Context Free Languages." *The Handbook of Formal Languages Vol II,* Springer Verlag, Berlin, 1997.

Appendix A

Lexical Analyzer Transition System



Appendix B

Context Free Grammar

```
<program>      -> <exp>;
<exp>          -> <logicalExp> | <mathExp> | <stringExp>
<mathExp>      -> <multMathExp> + <multMathExp> |
                  <multMathExp> - <multMathExp> |
                  <multMathExp>
<multMathExp> -> <subMathExp> * <subMathExp> |
                  <subMathExp> / <subMathExp> |
                  <subMathExp>
<subMathExp>   -> (<mathExp>) | IDENTIFIER | FLOATING_POINT_LITERAL |
                  INTEGER_LITERAL | CHARACTER_LITERAL
<logicalExp>   -> <andExp> || <andExp> |
                  <andExp>
<andExp>       -> <bitExp> && <bitExp> |
                  <bitExp>
<bitExp>       -> <subLogExp> '|' <subLogExp> |
                  <subLogExp> & <subLogExp> |
                  <subLogExp>
<subLogExp>    -> !<logicalExp> | (<logicalExp>) | <relExp> | IDENTIFIER
<relExp>       -> <equalsRelExp> == <equalsRelExp> |
                  <equalsRelExp> != <equalsRelExp> |
                  <equalsRelExp>
<equalsRelExp> -> <subRelExp> '>' <subRelExp> |
                  <subRelExp> '<' <subRelExp> |
                  <subRelExp> >= <subRelExp> |
                  <subRelExp> <= <subRelExp> |
                  <subRelExp>
<subRelExp>    -> (<relExp>) | IDENTIFIER | FLOATING_POINT_LITERAL |
                    INTEGER_LITERAL | CHARACTER_LITERAL
<stringExp>    -> <string> + <string> | <string>
<string>       -> (<stringExp>) | STRING_LITERAL | CHARACTER_LITERAL |
                   INTEGER_LITERAL | FLOATING_POINT_LITERAL | IDENTIFIER
```

Appendix C

Test Output

testFile.java

```
1+2*3;
```

Output

```
1 -> INTEGER_LITERAL
+ -> BINARY_ARITHMETIC_OPERATOR
2 -> INTEGER_LITERAL
* -> BINARY_ARITHMETIC_OPERATOR
3 -> INTEGER_LITERAL
; -> SEMICOLON

program
exp
mathExp
multMathExp
subMathExp
1 --> INTEGER_LITERAL
+ --> BINARY_ARITHMETIC_OPERATOR
multMathExp
subMathExp
2 --> INTEGER_LITERAL
* --> BINARY_ARITHMETIC_OPERATOR
subMathExp
3 --> INTEGER_LITERAL
exp
mathExp
multMathExp
subMathExp
logicalExp
andExp
bitExp
subLogExp
relExp
equalsRelExp
subRelExp
stringExp
string
; --> SEMICOLON
Parsed Successfully.
```

testFile.java
```
oneVar && (anotherVar >= 2);
```

Output

```
oneVar -> IDENTIFIER
&& -> BINARY_LOGICAL_OPERATOR
( -> OPEN_PAREN
anotherVar -> IDENTIFIER
>= -> BINARY_RELATIONAL_OPERATOR
2 -> INTEGER_LITERAL
) -> CLOSE_PAREN
; -> SEMICOLON
program
exp
mathExp
logicalExp
andExp
bitExp
subLogExp
oneVar --> IDENTIFIER
&& --> BINARY_LOGICAL_OPERATOR
bitExp
subLogExp
( --> OPEN_PAREN
logicalExp
andExp
bitExp
subLogExp
anotherVar --> IDENTIFIER
>= --> BINARY_RELATIONAL_OPERATOR
exp
mathExp
multMathExp
subMathExp
2 --> INTEGER_LITERAL
exp
mathExp
multMathExp
subMathExp
logicalExp
) --> CLOSE_PAREN
exp
mathExp
multMathExp
subMathExp
logicalExp
andExp
bitExp
subLogExp
relExp
equalsRelExp
subRelExp
stringExp
string
; --> SEMICOLON
Parsed Successfully.
```

```
testFile.java
"This is a string" + 'a';
```

Output

```
"Thisisastring" -> STRING_LITERAL
+ -> BINARY_ARITHMETIC_OPERATOR
'a' -> CHARACTER_LITERAL
; -> SEMICOLON
program
exp
mathExp
multMathExp
subMathExp
logicalExp
andExp
bitExp
subLogExp
relExp
equalsRelExp
subRelExp
stringExp
string
"Thisisastring" --> STRING_LITERAL
+ --> BINARY_ARITHMETIC_OPERATOR
string
'a' --> CHARACTER_LITERAL
exp
mathExp
multMathExp
subMathExp
logicalExp
andExp
bitExp
subLogExp
relExp
equalsRelExp
subRelExp
stringExp
string
; -> SEMICOLON
Parsed Successfully.
```