Tokenizing a Grammar:

A Java Implementation of Lexical Analysis

Tyler Rimaldi

Marist College School of Computer Science and Mathematics

Poughkeepsie, NY 12601

{Tyler.Rimaldi1}@Marist.edu

CMPT 440 111 20S

Professor Pablo Rivas

Abstract

Lexical analysis is the front end piece of a compiler; it pulls apart a program and strips each input into individual words: lexical tokens. A lexical token is a sequence of characters that are categorized by type. This Java implementation of lexical analysis will specify lexical tokens through regular expressions and implement lexers through deterministic finite automata. In this paper, the fundamentals topics such as lexical analysis, deterministic finite automata, and regular expressions will be explored in detail and will be demonstrated via a Java lexical analysis application.

Introduction:

At Marist College, I majored in Computer Science. Prior to my time at college, I had no programming experience. When I took my first object oriented programming course, the class was taught in Java and I quickly became fascinated with the inner workings of the Java Virtual Machine (JVM). Soon I learned about compiling and what it meant to compile a program; this was the spark: knowing what happens when I click run. I will be using this motivation in this project to explore my fascination and hope to create the very first stage of a compiler: a lexer.

Background

This section provides the necessary background knowledge that will further help the reader understand my implementation of the Java lexical analysis tool that I have come to build. This information is presented in the order that it was used during the development.

Deterministic Finite Automata

The first step to this project is recognizing validity, namely what symbol or collection of symbols ought to be accepted or rejected; the concept of Deterministic Finite Automata (DFA) provides guidance in this area. Very informally, a DFA is a simple diagram that shows the flow of inputs as they transition to different states until they are finally rejected or accepted. Formally, a DFA is a mathematically defined 5-tuple consisting of a set of states, an alphabet, a transition function, a start state, and a set of accepting states. This culmination of tools proves a string of symbols to be accepted or rejected by a DFA with respect to a given regular language. In this Java program, DFAs will show whether programmatic inputs match certain types as defined by a language grammar. The next subsection discusses how this process ought to be performed.

Lexical Analysis

As mentioned previously, the lexical analysis phase is the first step in developing a compiler. Essentially, it takes in source code from language in the form of sentences. The lexical analyzer (lexer) will break down the sentence, removing white space and comments, and collect any remaining syntaxes. It will then group the remaining syntaxes into a series of tokens defined by a language grammar. Similar to a DFA, if a token is in valid, the lexer will reject that token and raise an error--and if all tokens are accepted, well then they all move to the next phase of the compiler which of course, will not be covered here.

I shall now define definitions for our use during the later parts of this paper:

Lexemes	These are sequences of symbols that fall into a token. A grammar defines rules for each lexeme to be identified as a valid token. The process of doing so is by pattern matching, or leveraging regular expressions (DFAs). In this implementation, we have a DFA established for each accepted token. Please see the appendix.
Tokens	In a programming language, these are generally things like identifiers, variable types, operations, and punctuations. See our grammar that defines these tokens in the appendix.
Alphabets	Finite set of symbols that are the building blocks of Strings.
Strings	Finite sequence of symbols from an alphabet that is building blocks of a language.
Language	Depending on the language implementation these can be finite or infinite. In our case we are working with a finite regular language.
Sentence	A collection of strings that fall under a language. In this implementation, we will be scanning sentences to validate their contents match our defined language grammar.
Longest Match Rule	When scanning source code from a language in the form of sentences, the lexer will scan the sentence symbol symbol until it decides each word is complete. During this process, the lexer may not be able to determine what exact token the lexeme falls under, so it uses the longest match among all tokens present to make a decision via rule priority.

Current Lexical Analysis Tools

There currently exist numerous amounts of lexical analysis tools. However, after looking around and discovering a few educational resources such as:

http://infolab.stanford.edu/~ullman/dragon/slides1.pdf

https://cs.nyu.edu/courses/spring11/G22.2130-001/lecture4.pdf

https://www.cs.yale.edu/flint/cs421/lectureNotes/c02.pdf

I became interested in somewhat combining the "Token Output Stream", "State-Oriented", and "Table Driven" approaches together. This is obviously subject to change, as I have not yet begun implementing anything. I am still learning the theory behind the lexical analysis process and how I can make sure to implement it in a way that ties in most of the topics we touched upon in class.

Our Java Lexical Analysis Tool

Project has not been finished so this is subject to change massively.

Requirements

Project has not been finished so this is subject to change massively.

User Manual

Project has not been finished so this is subject to change massively.

Conclusion

Project has not been finished so this is subject to change massively.

References

Project has not been finished so this is subject to change massively.

Things for Professor Rivas:

1) Thank you for reading this.
2) Please move to the appendix section and let me know if there is any other way you'd like me to present the DFAs I've created. I could attempt to create the very large and complex version of all of the DFAs unionized, but I think that would take far more attention away from the project--and I mean this by saying that it would require far too much time when programmatically, it would be easier for me to follow each DFA structure it self--I could be terribly wrong, so please do scold me or provide text in all caps to simulate the yelling.
3) What the heck should I do for the 5-tuple, this answer is dependent on your answer to question 2? Also, you will find references to this question on the last page of this milestone.
4) Any additional advice related to the programmatic part is welcomed.
5) Stay healthy :)

Appendix

1. Grammar (excluding “while statement”) Provided by Professor Labouseur

Our Language Grammar

```
Program             ::== Block $
Block               ::== { StatementList  }
StatementList       ::== Statement StatementList
                    ::== ε
Statement           ::== PrintStatement
                    ::== AssignmentStatement
                    ::== VarDecl
                    ::== WhileStatement
                    ::== IfStatement
                    ::== Block
PrintStatement      ::== print ( Expr )
AssignmentStatement ::== Id = Expr
VarDecl             ::== type Id
WhileStatement      ::== while BooleanExpr Block
IfStatement         ::== if BooleanExpr Block

Expr                ::== IntExpr
                    ::== StringExpr
                    ::== BooleanExpr
                    ::== Id
IntExpr             ::== digit intop Expr
                    ::== digit
StringExpr          ::== " CharList "
BooleanExpr         ::== ( Expr boolop Expr )
                    ::== boolval
Id                  ::== char
CharList            ::== char CharList
                    ::== space CharList
                    ::== ε

type                ::== int | string | boolean
char                ::== a | b | c ... z
space               ::== the space character
digit               ::== 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

boolop              ::== == | !=
boolval             ::== false | true
intop               ::== +

Comments are bounded by /* and */ and ignored by the lexer.
```

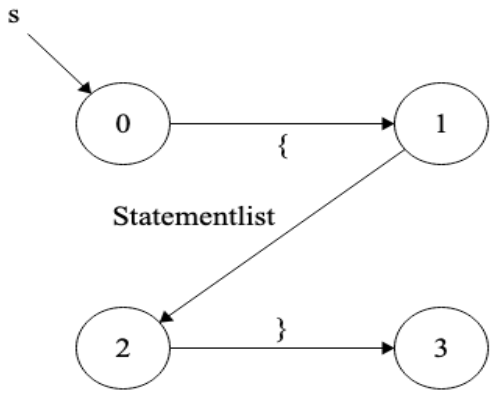
Curly braces denote scope.

= is assignment.

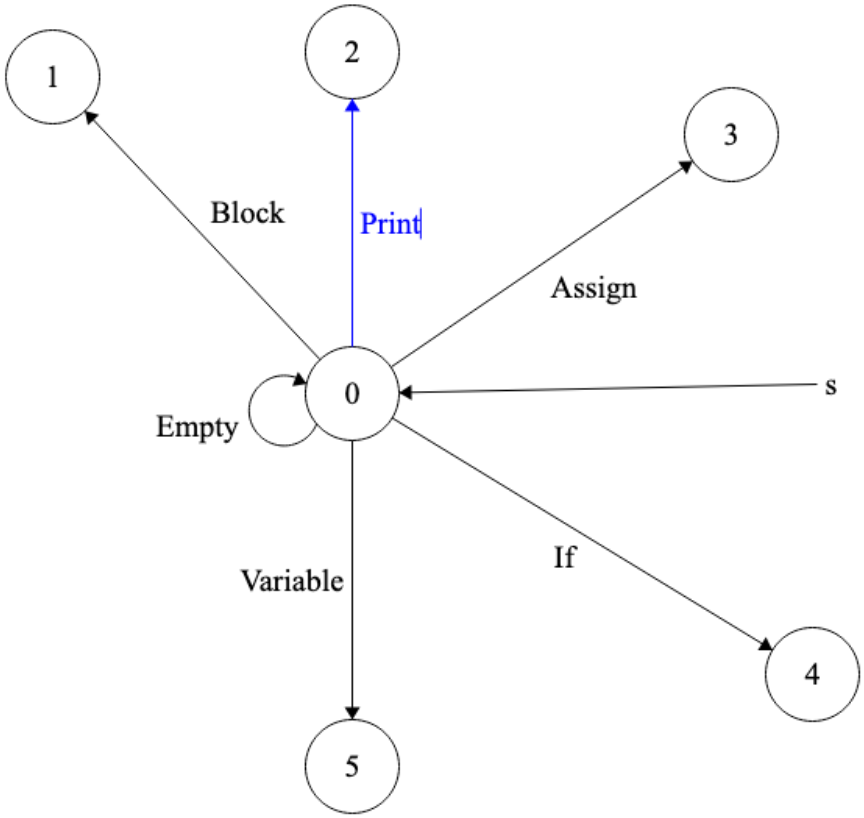
== is test for equality.

2) DFA per type as defined by the grammar above (the dfa would be massive, so I've chopped it into components which shall define our grammar):

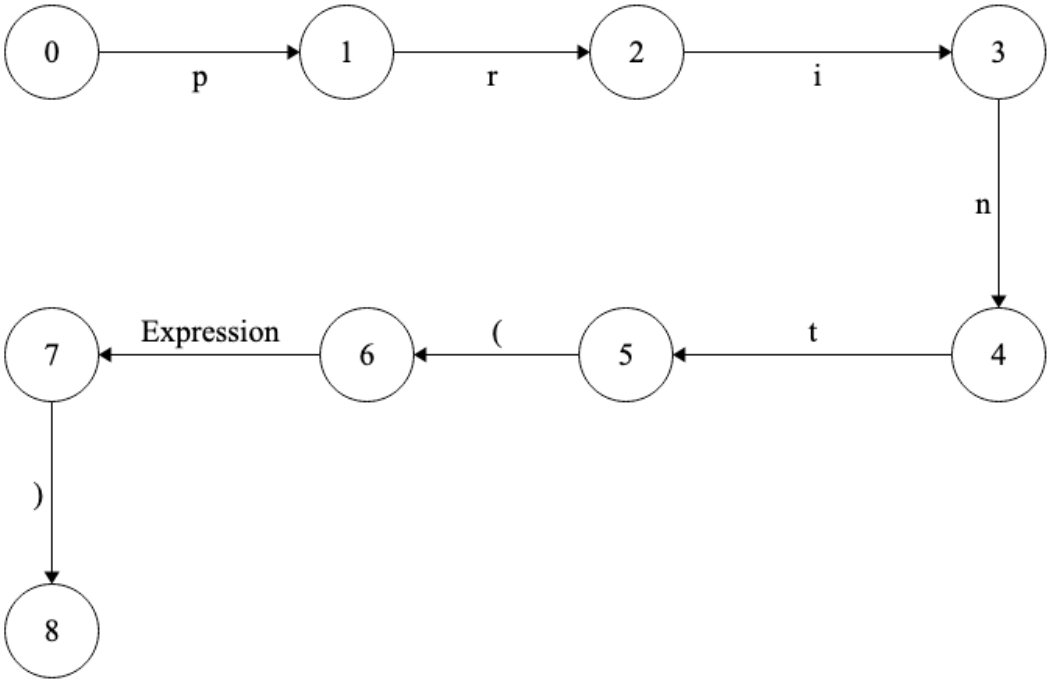
Block (3 is the accepting state):



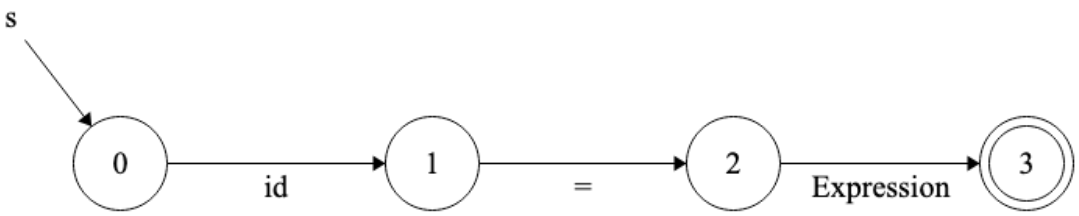
Statement: (0,1,2,3,4,5) are accepting states--ignore the blue.



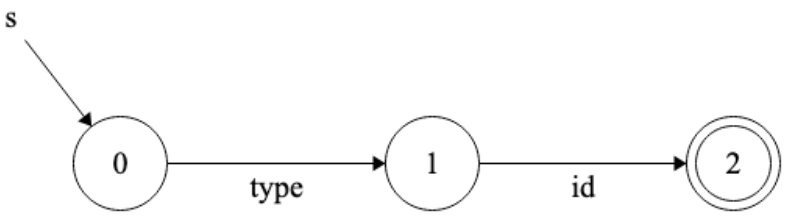
Print Statement (8 is the accepting state):



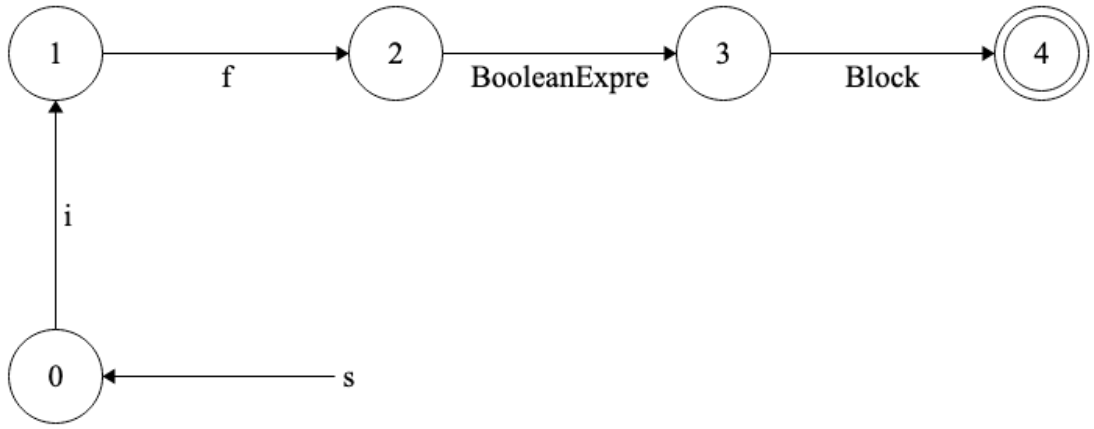
Assignment Statement:



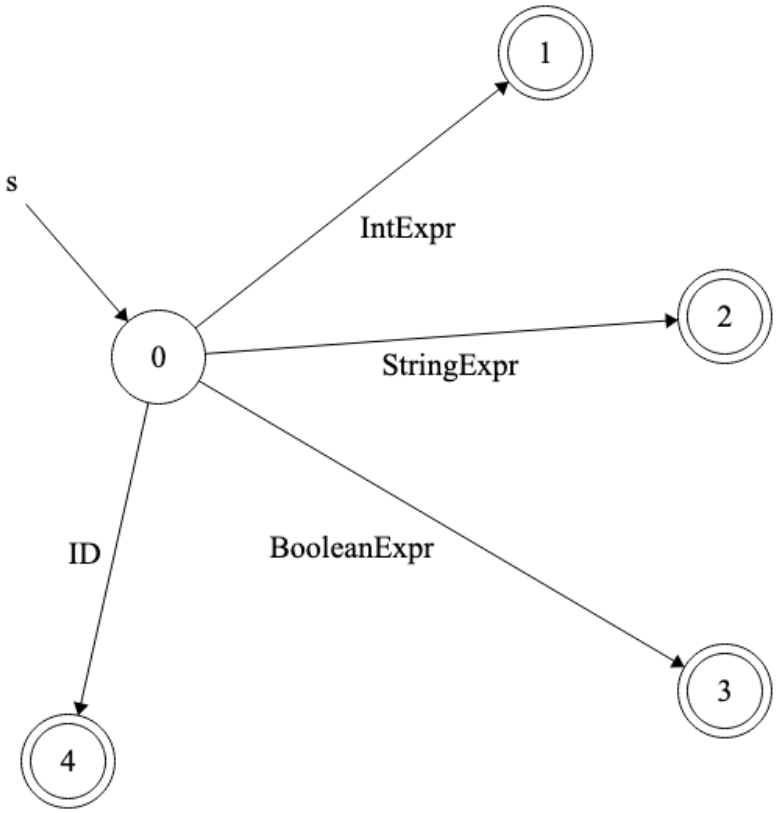
Variable Statement:



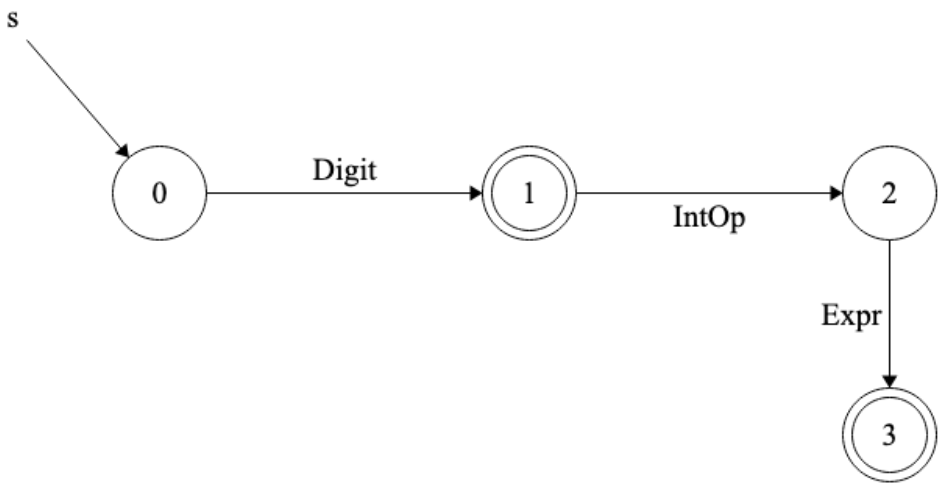
If Statement:



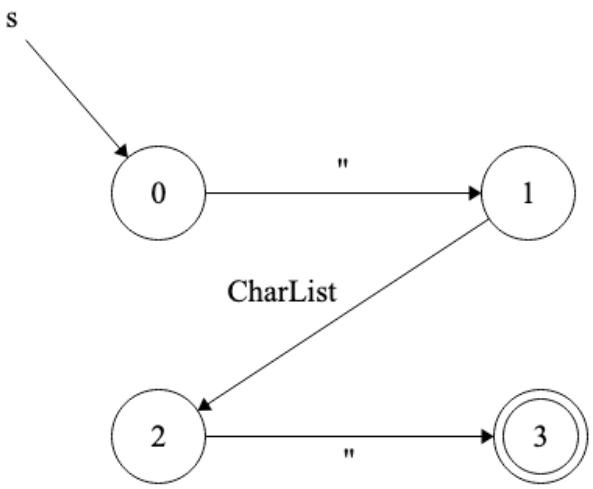
Expression:



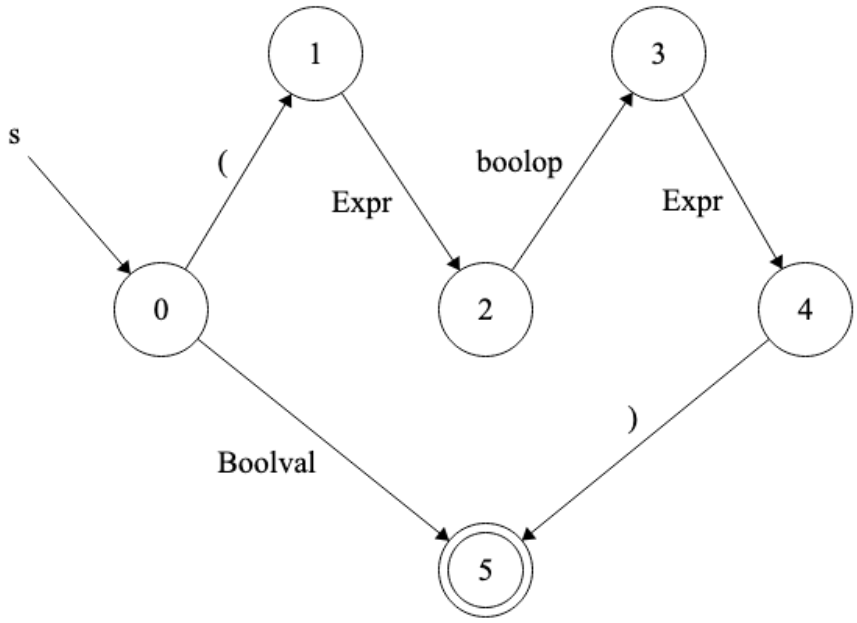
Int Expression:



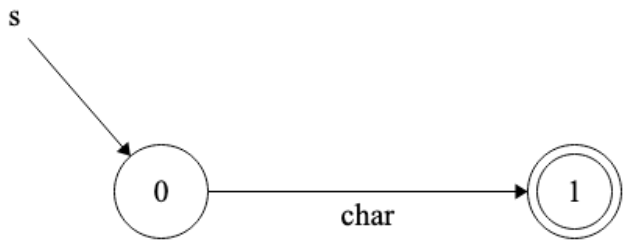
String Expression:



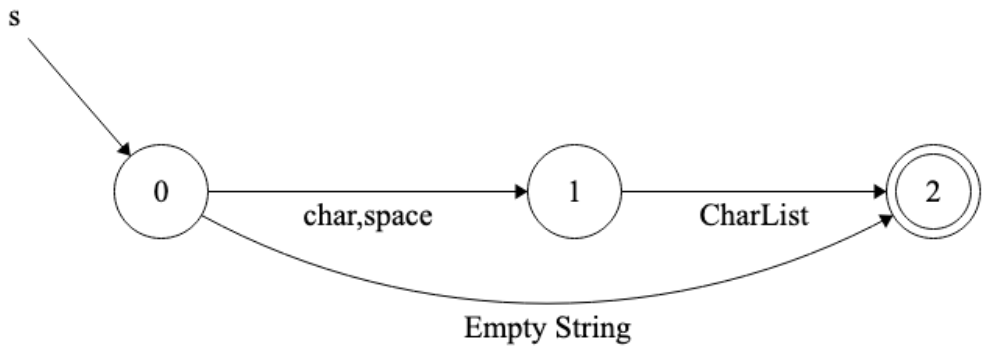
Boolean Expression:



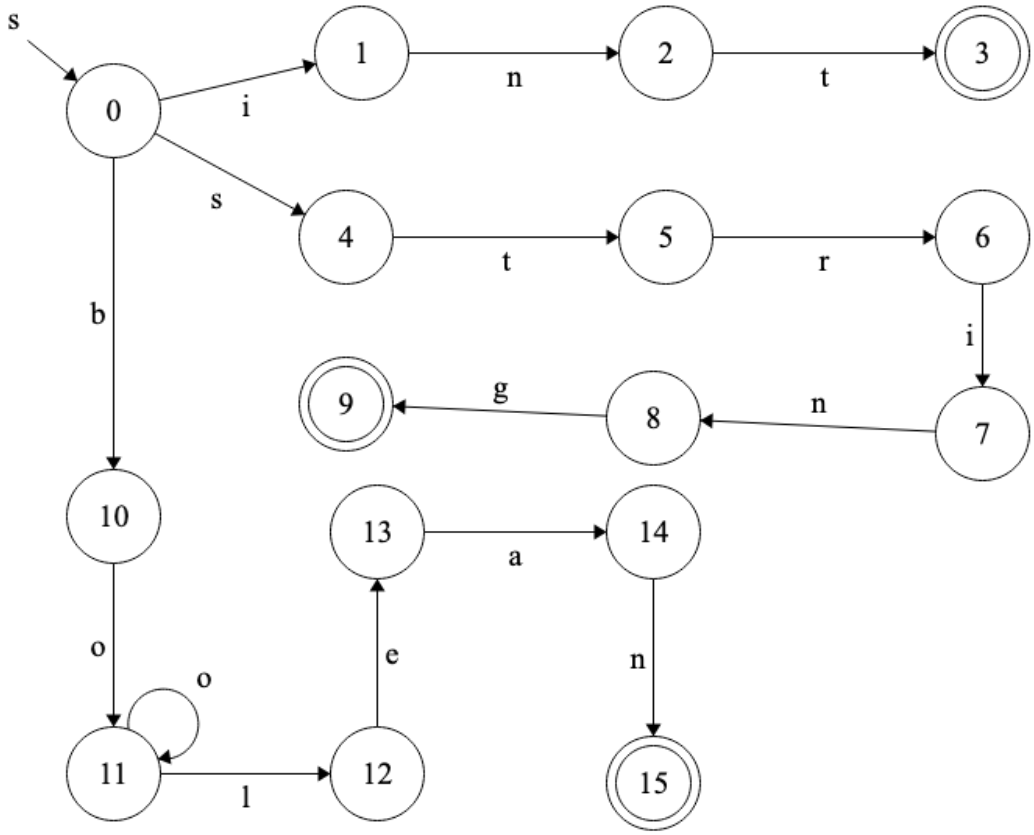
ID Expression:



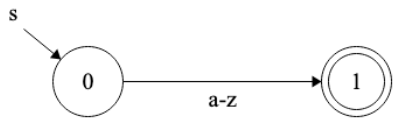
CharList:



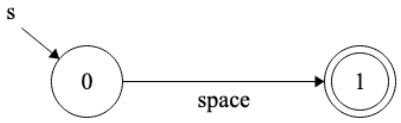
Type:



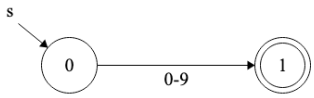
Char:



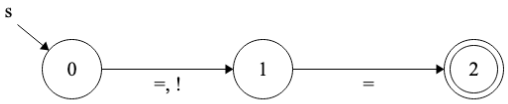
Space:



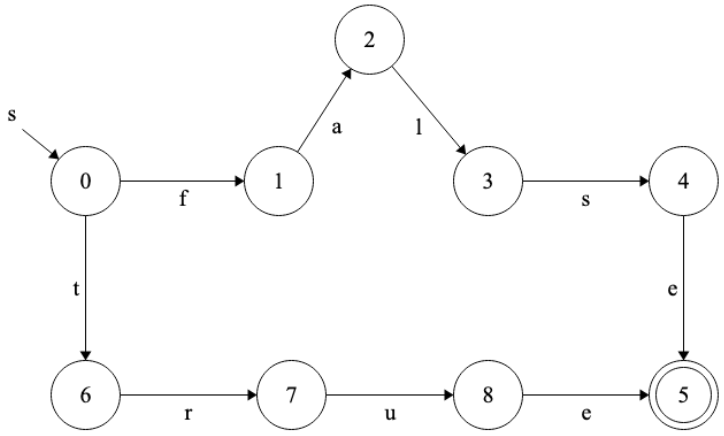
Digit:



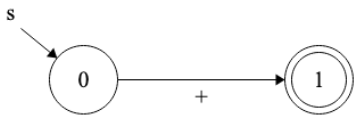
Boolop:



Boolval:



IntOp:



Below is the 5-Tuple--it is not filled in because I have questions for you which are asked in the "Things For Professor Rivas" section.

Alphabet:

Start State:

Set of States:

Transition Function:

Accepting States: