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. In this paper, the fundamentals topics such as lexical analysis, deterministic finite automata, and regular expressions will be explored and 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. BackgroundThis 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 AutomataThe 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:LexemesThese 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.TokensIn a programming language, these are generally things like identifiers, variable types, operations, and punctuations. See our grammar that defines these tokens in the appendix.AlphabetsFinite set of symbols that are the building blocks of Strings.StringsFinite sequence of symbols from an alphabet that is building blocks of a language.LanguageDepending on the language implementation these can be finite or infinite. In our case we are working with a finite regular language.SentenceA 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 RuleWhen 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.pdfI

became interested in somewhat combining the “Token Output Stream”, “State-Oriented”, and “Table Driven” approaches together. I ended up combining these together with regex as you will see in the next section.

Our Java Lexical Analysis Tool

This project has a very simple implementation of lexical analysis by leveraging Java Enums and fancy regex imports. By using regex we can define clever Enum properties that define our grammar. By using this Enum we can then use it to tokenize the incoming program. The algorithm in which we apply the regex is also quite simple, but intense. We will condense the program by stripping all comments and white space, then traverse through this, now what can be imagined to be, one liner code sequence. By condensing the code this way we reduce any spacing errors from occuring. Now we also introduced overhead, namely the look current and look ahead pointer. We didn't neccessarily implement a nasty look a head pointer, but one that got the job done (I'll do that in compilers...). Now we can use these pointers and decide what token each componenent of the program should be referred to as, or if we should exit because the program has invalid syntax accordance to it's claimed grammar. We have established test cases to demonstrate this functionality, as will be explained in further sections.

Requirements

The requirements necessary to run this project are very few: Git and Java. Once Git and Java have been installed on your machine, you will simply need to pull down our repository:

`**https://github.com/Vezio/cmpt440Rimaldi**`

Then run our script (more mentioned in the User Manual section) which runs the entire project and it’s test cases. This assumes that you will be using our given grammar, as specified in our appendix. If you wish to use a different grammar, fear not, as you will be able to make those changes very easily. How you might ask, well simple, just regex! Edit the “Token.Java” file and adjust the Enum accordingly to adapt to your language. If you are perfectly fine with using our preset language, then just simply install Java and you’ll be one step closer to using our application.

User Manual

To get started, please ensure Git and Java are both installed on your machine. As mentioned in the previous section, you will need to clone the following repository from github by entering the command:

`https://github.com/Vezio/cmpt440Rimaldi.git`

Next, navigate to the following directory:

`cmpt440Rimaldi/project/writeup/code/src`

To see working demo of our lexer using preset functions, run the following commands:

`chmod +x ./run.sh`

`./run.sh`

To show the project command line arguments and the usage, run the following commands (the first three ensure that you have compiled the code before trying to execute it):

`javac Driver.java`

`javac token/Token.java` `

`javac lexer/Lexer.java`

`java Driver.java -h`

To run the lexer against a custom function (which adheres to our grammar found in the appendix), run the following:

`javac Driver.java`

`javac token/Token.java`

`javac lexer/Lexer.java`

`java Driver.java -f “path-to-function.txt"`

To run the lexer against your own grammar, please edit the

`Token.Java`

Enum's regex, this is likely the only spot you will need to change. The lexer will be able to perform everything else as expected, but it just needs to know your grammar's definitions.

Test Case Analysis

The previous section illustrates how a user might run our test cases, let's analyze what our test cases are and what their respective outputs are:

- Test Case 1
  - **Input, (Function closed by our grammar)**:

```
/*
  Test out our comment handling!
*/

{
  /*Comment*/
  int x =  /*Comment*/ 0

  while(x !=  /*Comment*/ 10) {
    x = x + 1
  }
}$
```

  - **Output, (clearly you will see that we chop up each component of the test function into its tokenized form, and spit back out to the user as displayed below)**:

```
=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~
RUNNING TEST CASE 1:
=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~
{                  :  OPEN_BRACKET
int                :  TYPE_INT
x                  :  IDENTIFIER
=                  :  OP_ASSIGN
0                  :  INTEGER
while              :  KEY_WHILE
(                  :  OPEN_PAREN
x                  :  IDENTIFIER
!=                 :  OP_NOT_EQ
10                 :  INTEGER
)                  :  CLOSE_PAREN
{                  :  OPEN_BRACKET
x                  :  IDENTIFIER
=                  :  OP_ASSIGN
x                  :  IDENTIFIER
+                  :  OP_PLUS
1                  :  INTEGER
}                  :  CLOSE_BRACKET
}                  :  CLOSE_BRACKET
$                  :  EOF_MARKER
```

- Test Case 2
  - **Input, (Function closed by our grammar)**:

    ```
    {
      print("hello world")
      int x = 0
      while (x != 100) {
        if (x == /* this is ignored! */ 2) {
          print("test")
        }
      }
    }$
    ```

  - **Output, (clearly you will see that we chop up each component of the test function into its tokenized form, and spit back out to the user as displayed below)**:

    ```
    =~=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~
    RUNNING TEST CASE 2:
    =~=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~
    {                   :  OPEN_BRACKET
    print               :  KEY_PRINT
    (                   :  OPEN_PAREN
    "helloworld"        :  STRING
    )                   :  CLOSE_PAREN
    int                 :  TYPE_INT
    x                   :  IDENTIFIER
    =                   :  OP_ASSIGN
    0                   :  INTEGER
    while               :  KEY_WHILE
    (                   :  OPEN_PAREN
    x                   :  IDENTIFIER
    !=                  :  OP_NOT_EQ
    100                 :  INTEGER
    )                   :  CLOSE_PAREN
    {                   :  OPEN_BRACKET
    if                  :  KEY_IF
    (                   :  OPEN_PAREN
    x                   :  IDENTIFIER
    ==                  :  OP_EQ
    2                   :  INTEGER
    )                   :  CLOSE_PAREN
    {                   :  OPEN_BRACKET
    print               :  KEY_PRINT
    (                   :  OPEN_PAREN
    "test"              :  STRING
    ```

```
)                   :  CLOSE_PAREN
}                   :  CLOSE_BRACKET
}                   :  CLOSE_BRACKET
}                   :  CLOSE_BRACKET
$                   :  EOF_MARKER
```

- Test Case 3
  - **Input, (Function closed by our grammar)**:

```
{
  boolean x = true
  boolean y = false
  int z = 0 /* this is ignored! */
  string s = "teststring"
  while (x) { /* this is ignored! */
    print(z /* this is ignored! */)
    if (z == 10) {
      x = false
    }
    z = z + 1
  }
}$
```

  - **Output, (clearly you will see that we chop up each component of the test function into its tokenized form, and spit back out to the user as displayed below)**:

```
=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~
RUNNING TEST CASE 3:
=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~
{                   :  OPEN_BRACKET
boolean             :  TYPE_BOOLEAN
x                   :  IDENTIFIER
=                   :  OP_ASSIGN
true                :  TRUE
boolean             :  TYPE_BOOLEAN
y                   :  IDENTIFIER
=                   :  OP_ASSIGN
false               :  FALSE
int                 :  TYPE_INT
z                   :  IDENTIFIER
=                   :  OP_ASSIGN
0                   :  INTEGER
string              :  TYPE_STRING
s                   :  IDENTIFIER
=                   :  OP_ASSIGN
```

```
"teststring"        :  STRING
while               :  KEY_WHILE
(                   :  OPEN_PAREN
x                   :  IDENTIFIER
)                   :  CLOSE_PAREN
{                   :  OPEN_BRACKET
print               :  KEY_PRINT
(                   :  OPEN_PAREN
z                   :  IDENTIFIER
)                   :  CLOSE_PAREN
if                  :  KEY_IF
(                   :  OPEN_PAREN
z                   :  IDENTIFIER
==                  :  OP_EQ
10                  :  INTEGER
)                   :  CLOSE_PAREN
{                   :  OPEN_BRACKET
x                   :  IDENTIFIER
=                   :  OP_ASSIGN
false               :  FALSE
}                   :  CLOSE_BRACKET
z                   :  IDENTIFIER
=                   :  OP_ASSIGN
z                   :  IDENTIFIER
+                   :  OP_PLUS
1                   :  INTEGER
}                   :  CLOSE_BRACKET
}                   :  CLOSE_BRACKET
$                   :  EOF_MARKER
```

- What if we want to see what an error looks like?
    - Test Case 3 w/ induced error
        - **Input, (Function not closed by our grammar)**:

```
{
  boolean x = true
  boolean y = false
  int z = 0 /* this is ignored! */
  string s = "testERRORstring"
  while (x) { /* this is ignored! */
    print(z /* this is ignored! */)
    if (z == 10) {
      x = false
    }
    z = z + 1
  }
}$
```

- Output, "testERRORstring" does not fit our grammar, as it specifically only allows lowercase characters (see appendix for grammar specific information)

```
=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~
RUNNING TEST CASE 3:
=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~=~
{                   :  OPEN_BRACKET
boolean             :  TYPE_BOOLEAN
x                   :  IDENTIFIER
=                   :  OP_ASSIGN
true                :  TRUE
boolean             :  TYPE_BOOLEAN
y                   :  IDENTIFIER
=                   :  OP_ASSIGN
false               :  FALSE
int                 :  TYPE_INT
z                   :  IDENTIFIER
=                   :  OP_ASSIGN
0                   :  INTEGER
string              :  TYPE_STRING
s                   :  IDENTIFIER
=                   :  OP_ASSIGN
-----------------------------------------------------------------
-----------
Syntax Error:

Error: Unexpected symbol
Context: "
-----------------------------------------------------------------
-----------
```

- We get an error thrown just before we get into the string. In another version of this Lexer, we will have much more verbose output to explain any errors as they occur.

Conclusion

This paper has explained the fundamentals about the front end component of a compiler: Lexical Analysis. It decomposes parts of a given program and strips each input into individual words: lexical tokens. Each lexical token consists of characters that are categorized by type. Our Java implementation of lexical analysis defined lexical tokens through regular expressions based on a particular grammar. Several test cases were developed and ran against our Java Lexer. We analyzed the outputs of those test cases and provided examples of test cases that would fail to pass our lexical analysis as per defined via grammar. All code is accessible to the public via the github username “vezio” under code repository “cmpt440Rimaldi”.

Appendix

Grammar (excluding "while statement") Provided by Professor Labouseur

Our Language Grammar

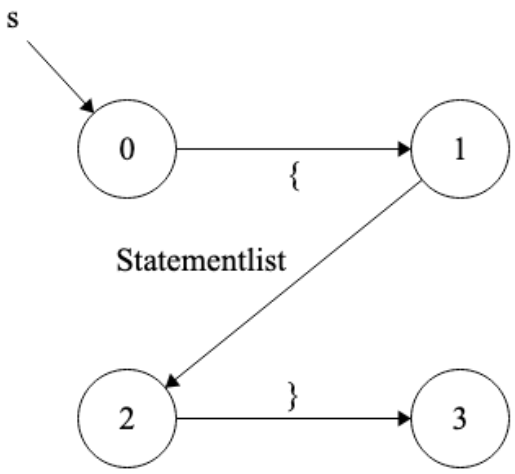
```
Program             ::== Block $
Block               ::== { StatementList  }
StatementList       ::== Statement StatementList
                    ::== ε
Statement           ::== PrintStatement
                    ::== AssignmentStatement
                    ::== VarDecl
                    ::== WhileStatement
                    ::== 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.

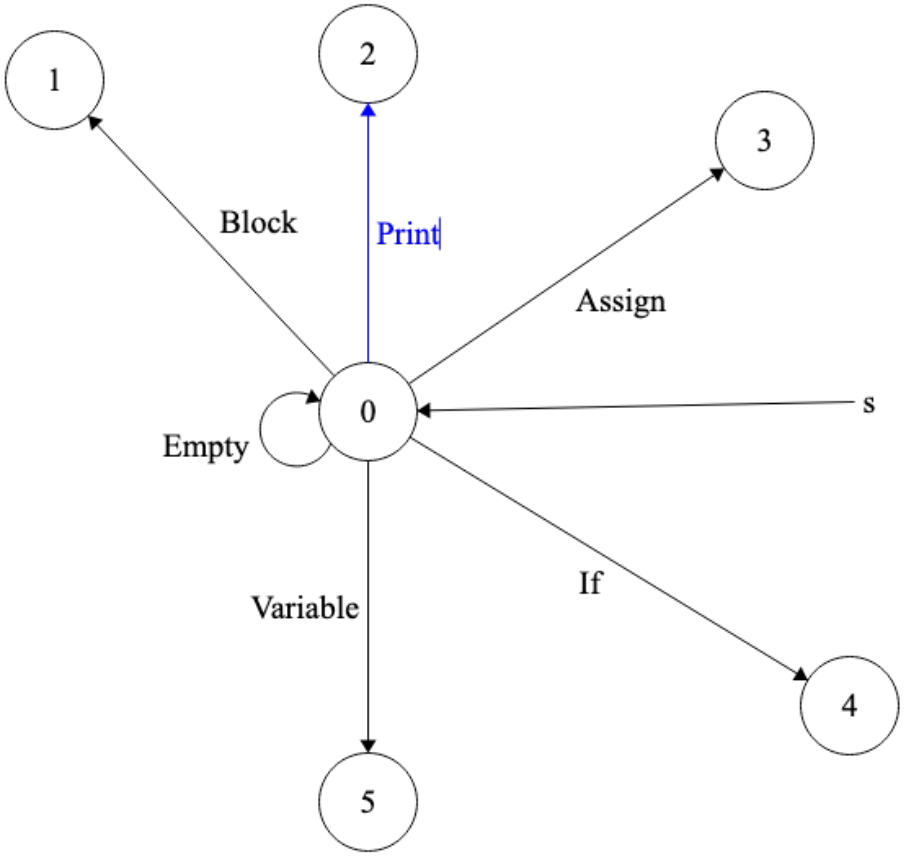
= is assignment.

== is test for equality.

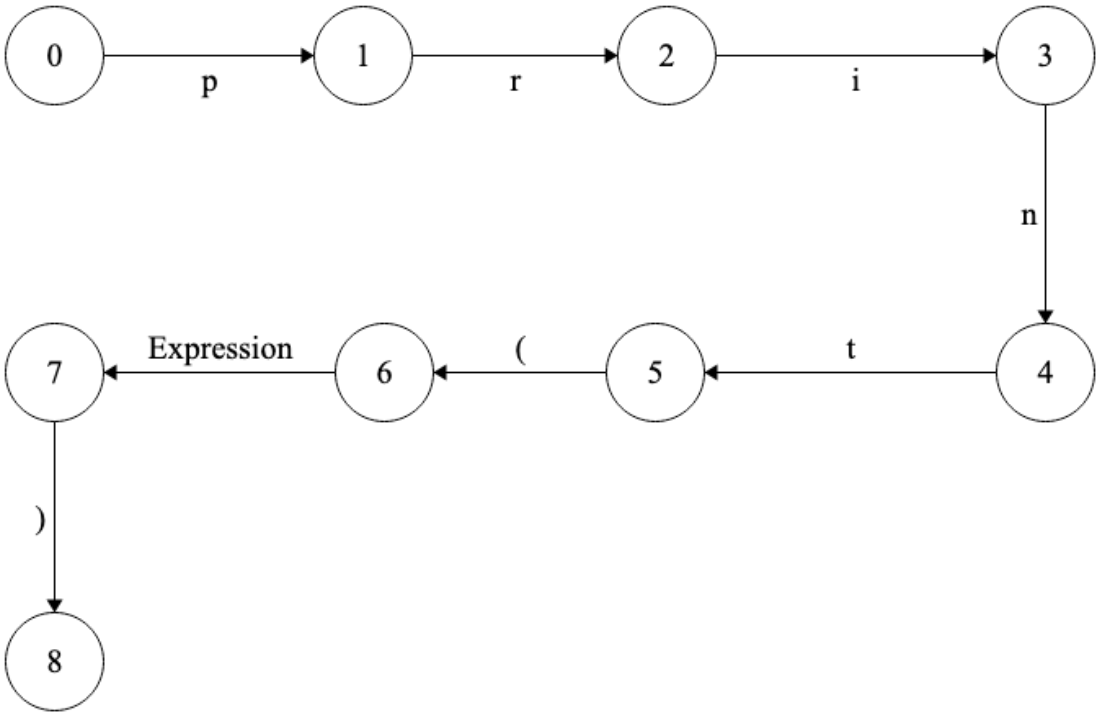
2) **DFA per type as defined by the grammar above (the dfa would be massive, so I've chopped it into components which shall define our grammar): Block (3 is the accepting state):**



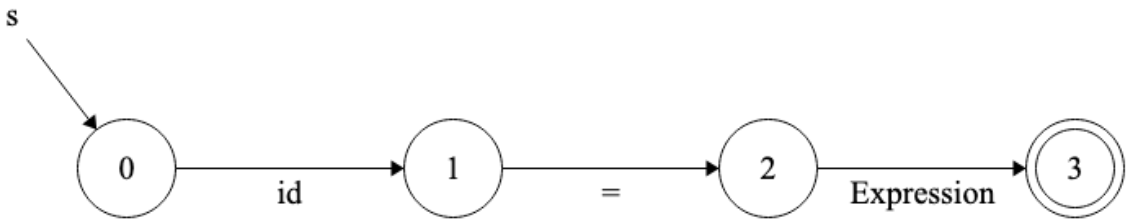
Statement: (0,1,2,3,4,5) are accepting states--ignore the blue.


1
2
3
Block
Print
Assign
0
s
Empty
If
Variable
4
5

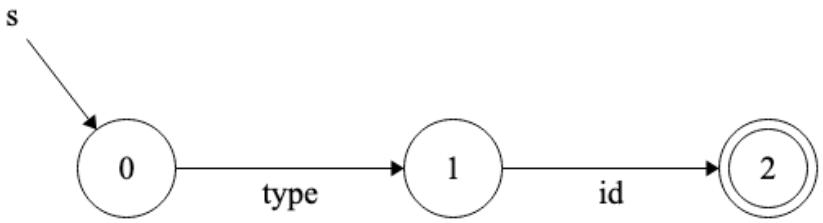
Print Statement (8 is the accepting state):



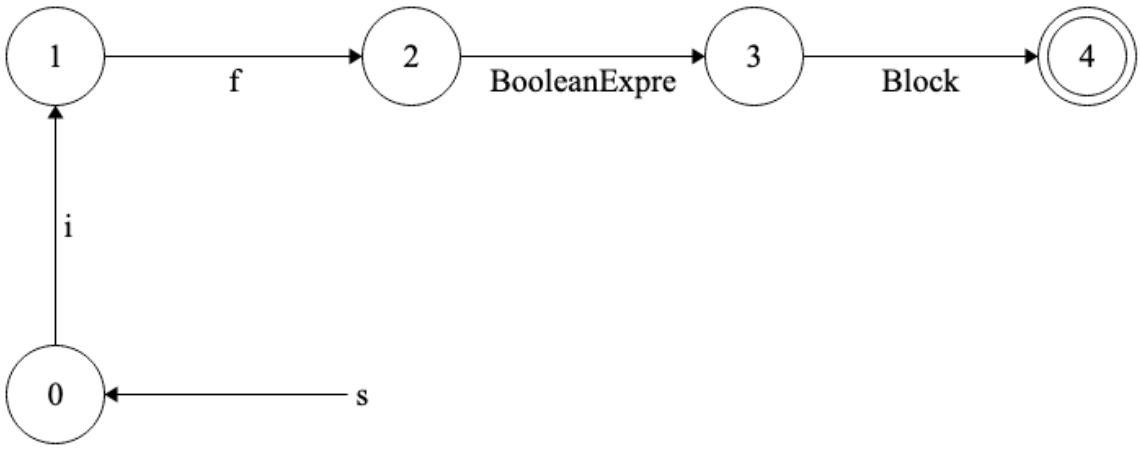
Assignment Statement:



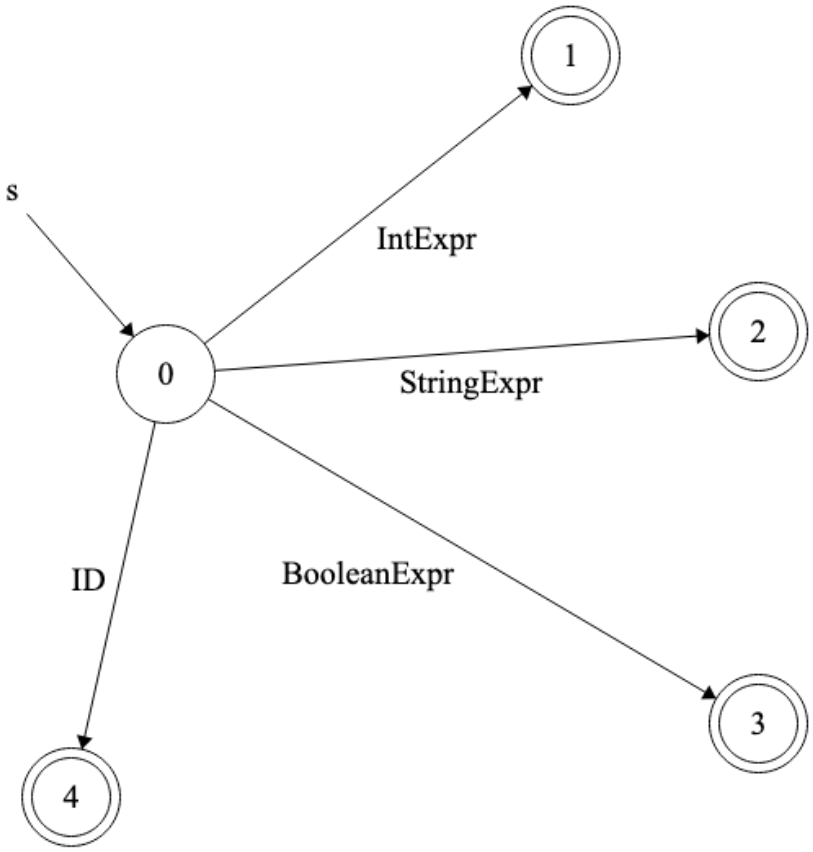
Variable Statement:



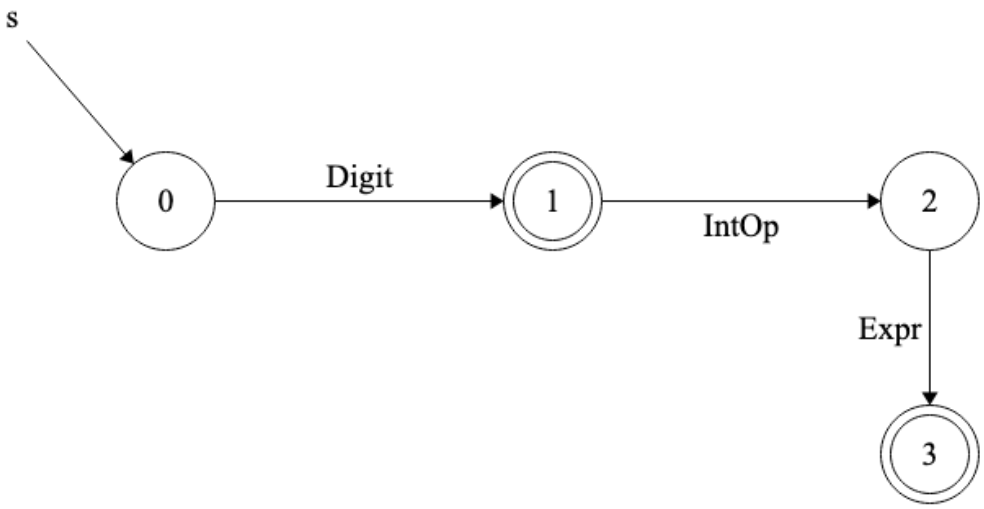
If Statement:



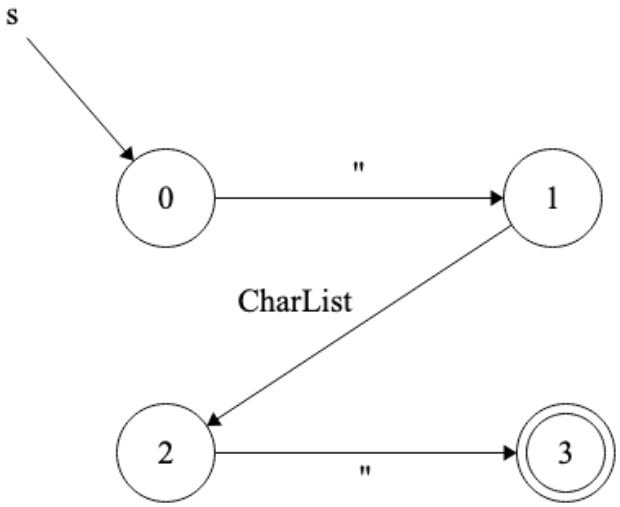
Expression:



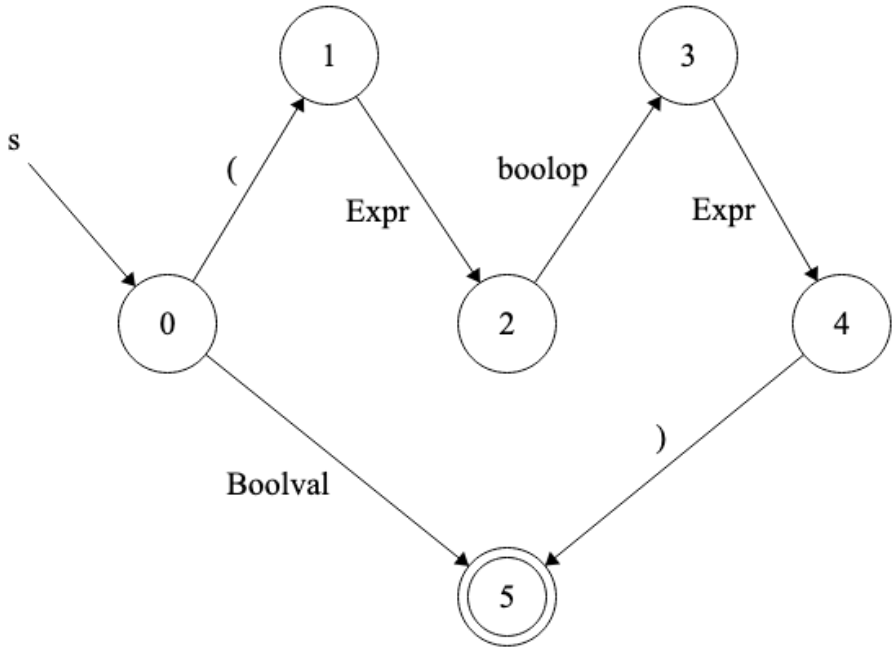
Int Expression:



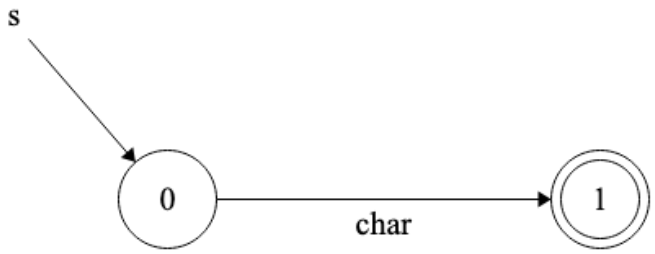
String Expression:



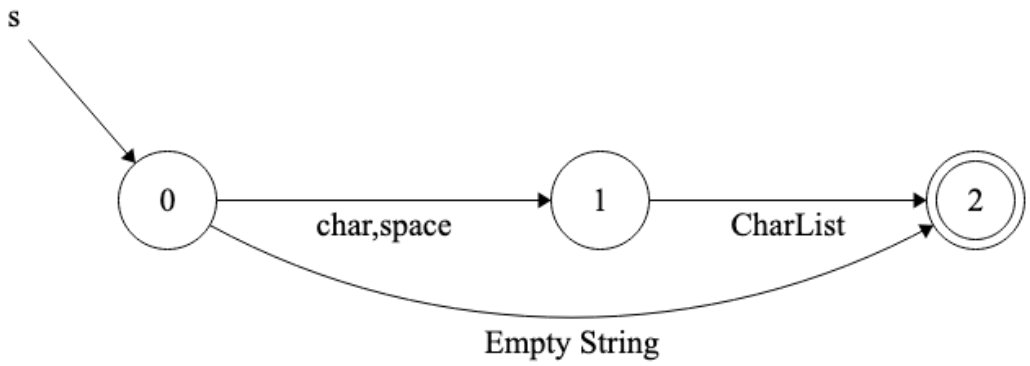
Boolean Expression:



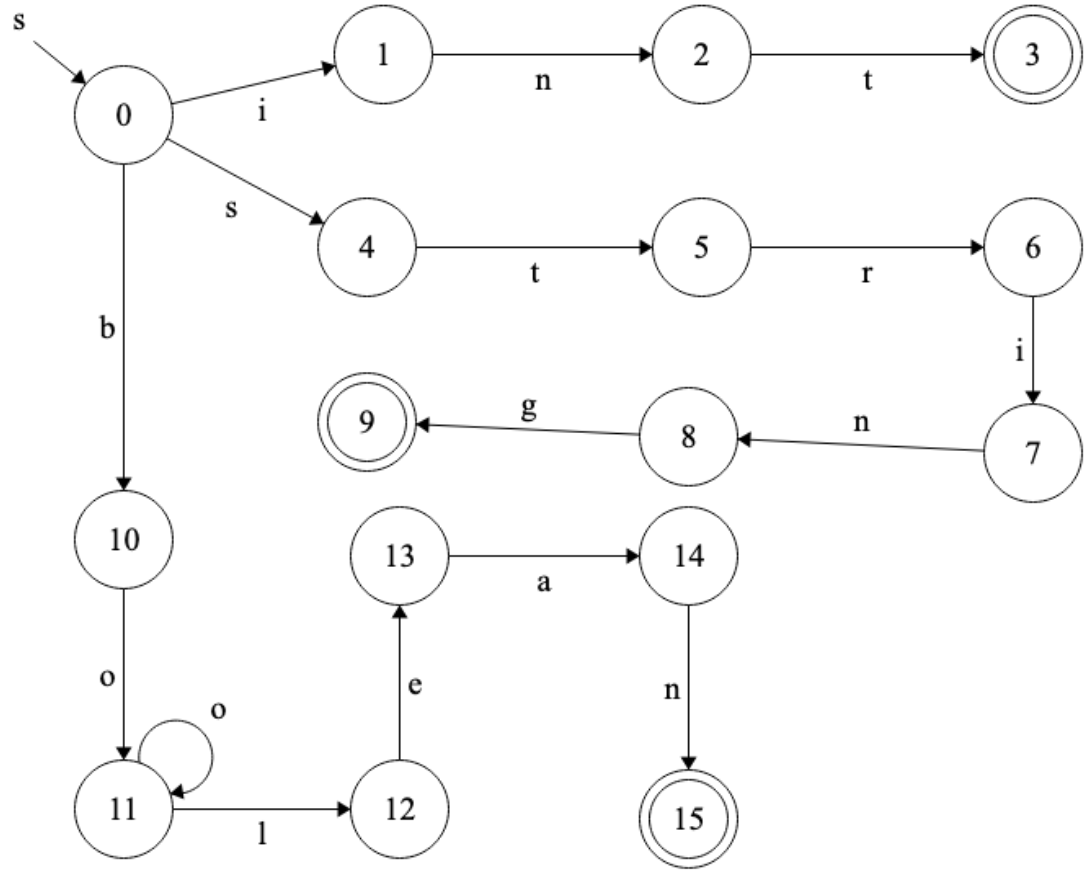
ID Expression:



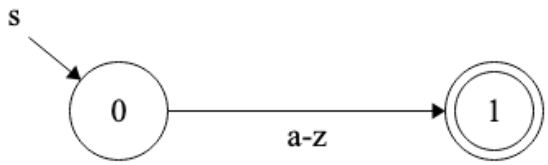
CharList:



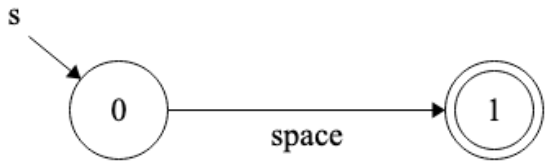
Type:



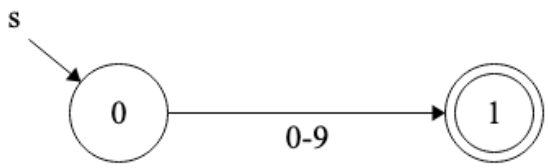
Char:



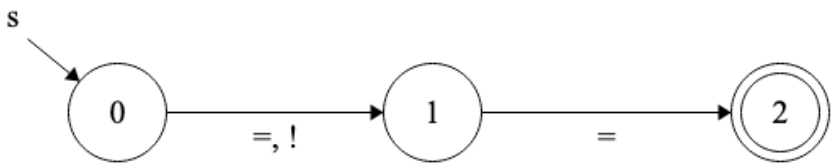
Space:



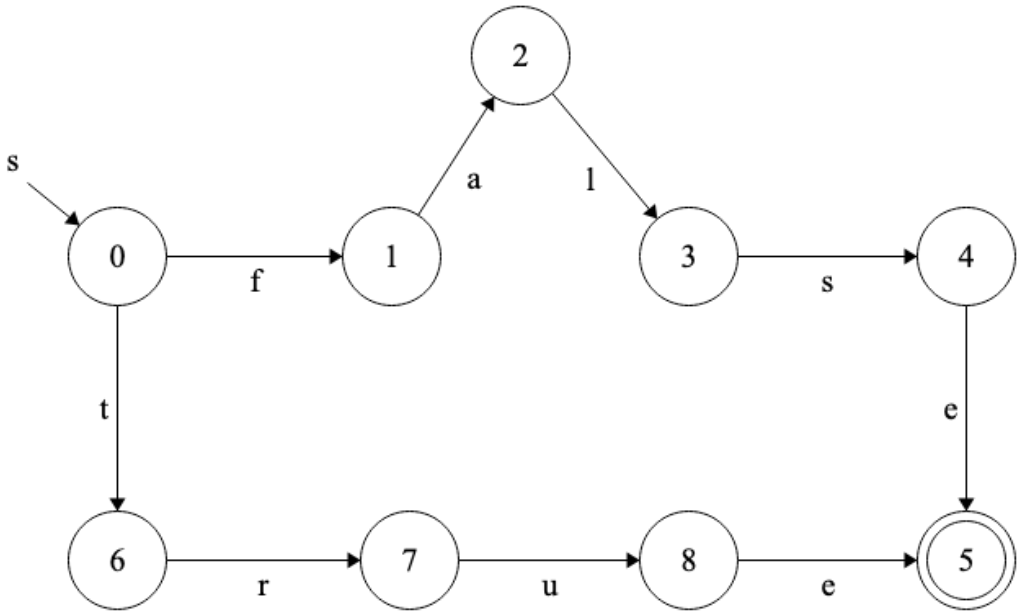
Digit:



Boolop:


s
0
=, !
1
=
2

Boolval:



IntOp: