# Tealang Interpreter

Tony Valentine

April 5, 2021

## Contents

## 1 Language Definition

The Tealang language is defined using the following EBNF. The '[]' wrapper is equivalent to '?' in regex, and the '{}' wrapper is equivalent to '*' in regex.

```
<Letter>             ::= [A-Za-z]
<Digit>              ::= [0-9]
<Printable>          ::= [\x20-\x7E]
<Type>               ::= 'float' | 'int' | 'bool' | 'string'
<BooleanLiteral>     ::= 'true' | 'false'
<IntegerLiteral>     ::= <Digit> { <Digit> }
<FloatLiteral>       ::= <Digit> { <Digit> } '.' <Digit> { <Digit> }
<StringLiteral>      ::= '"' {<Printable> } '"'
<Literal>            ::= <BooleanLiteral>
                         |<IntegerLiteral>
                         |<FloatLiteral>
                         |<StringLiteral>
<Identifier >        ::= ('_'| <Letter>) {'_'| <Letter> | <Digit> }
<MultiplicativeOp>   ::= '*' | '/' | 'and'
<AdditiveOp>         ::= '+' | '-' | 'or'
<RelationalOp>       ::= '<' | '>' | '==' | '!=' | '<=' | '>='
<ActualParams>       ::= <Expression> { ',' <Expression> }
<FunctionCall >      ::= <Identifier > '(' [ <ActualParams> ] ')'
<SubExpression>      ::= '(' <Expression> ')'
<Unary>              ::= ( '-' | 'not' ) <Expression>
<Factor >            ::= <Literal >
                         |<Identifier >
                         |<FunctionCall >
                         |<SubExpr>
                         |<Unary>
<Term>               ::= <Factor > { <MultiplicativeOp> <Factor > }
<SimpleExpression>   ::= <Term> { <AdditiveOp> <Term> }
<Expression>         ::= <SimpleExpression> { <RelationalOp> <SimpleExpression> }
<Assignment>         ::= <Identifier > '=' <Expression>
<VariableDecl >      ::= 'let' <Identifier > ':' <Type> '=' <Expression>
<PrintStatement>     ::= 'print' <Expression>
<RtrnStatement>      ::= 'return' <Expression>
<IfStatement>        ::= 'if' '(' <Expression> ')' <Block > [ 'else' <Block > ]
<ForStatement>       ::= 'for' '(' [ <VariableDecl > ] ';' <Expression> ';'
                        [<Assignment> ] ')' <Block >
<WhileStatement>    ::= 'while' '(' <Expression> ')' <Block >
<FormalParam>       ::= <Identifier > ':' <Type>
<FormalParams>      ::= <FormalParam> { ',' <FormalParam> }
<FunctionDecl >     ::= <type> <Identifier > '(' [ <FormalParams> ] ')' <Block >
<Statement>         ::= <VariableDecl > ';'
                        |<Assignment> ';'
                        |<PrintStatement> ';'
                        |<IfStatement>
                        |<ForStatement>
                        |<WhileStatement>
                        |<RtrnStatement> ';'
                        |<FunctionDecl >
                        |<Block >
<Block >            ::= '{' { <Statement> } '}'
<Program>           ::= { <Statement> }
```

## 2 Lexer

### 2.1 Character classes

Finite Automata transitions are triggered by reading characters and then using a look-up table to determine the new state. The total number of characters needed for the lexer to operate is over 100, meaning for each state of the automata we would need to define over 100 transitions. This number can be drastically reduced by using character classes wherein characters are grouped together. This does increase the performance overhead of the lexer however this is rather negligible compared to complexity savings.

| Class Name | Class Capture Group |
|---|---|
| Digit | Numbers: [0-9] |
| Decimal | Period/Decimal Point: . |
| Identifier | ASCII Letters and Underscore |
| Comparison | Angle Brackets |
| Equals | Equality Sign |
| Bang | Exclamation Mark |
| Punctuation | Brackets (Curly and Round), Semi/Normal Colons |
| FSlash | Forward Slash Symbol |
| PlusMinus | Plus and Minus symbol |
| Asterisk | Asterisk Sign |
| BSlash | Backslash symbol |
| Qoute | Captures double quotes |
| Newline | ASCII Newline character |
| Printable | All printable ascii symbols 32-126 |

The character classes are implemented such that there is no overlap between any two classes except for the printable classification. This classification is the final catch-all for all characters meaning it can only be assigned if the tests for other classes all failed.

## 2.2 Lexer Components

### 2.2.1 Identifiers and Keywords

Keywords are reserved identifiers required for operation of the language. Since the set of all keywords is a subset of all identifiers we can focus on detecting correct identifiers first and then check for keywords. The regex for an identifier is `(_|[A-z])(_|[A-z]|[0-9])+`. To determine the exact token a map can be used to locate the correct token in logarithmic time.

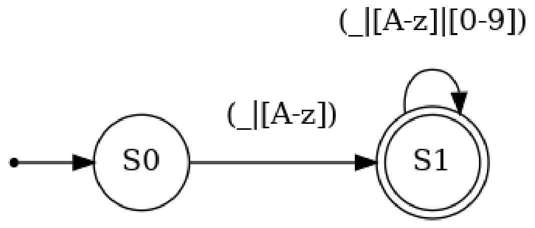


Figure 1: Identifier DFA

### 2.2.2 Integers and Floats

Taking a look at the BNF it can be seen that a the definition for `float` is a concatenation of `int` with `.` and another `int`. As such a dfa can be defined to accept both of these strings, and they can be interpreted according the final state of the automaton. In the automaton below if the system stops in state `Int` we know it must be an integer, whereas stopping in state `Float` assures that the number is a float.

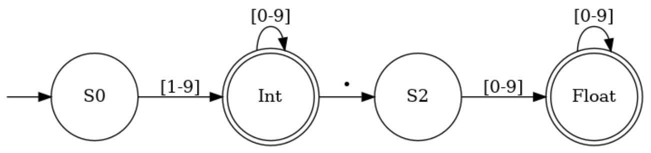


Figure 2: Integer and Float DFA

### 2.2.3 Relational Operators and Equality

The relational operators require a slightly more complex DFA than previous examples. This is specifically due to the `!=` and singular `=` operator. However these can easily be resolved with 2 extra cases in the DFA.

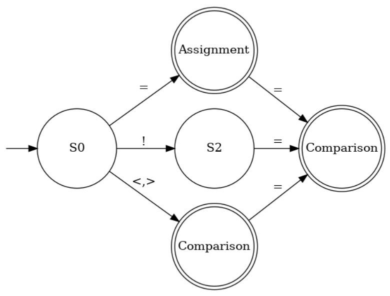
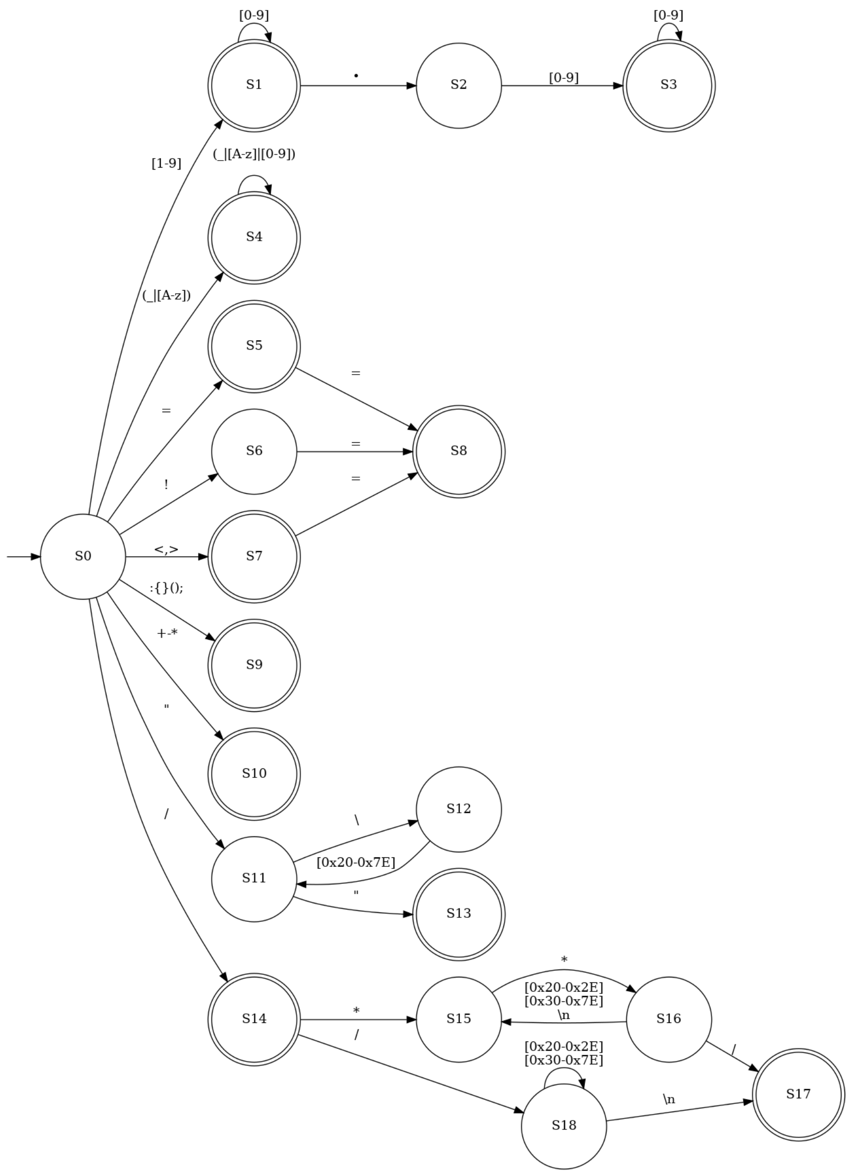


Figure 3: Comparison and Assignment DFA

## 2.3 Lexer DFA


[0-9]
S1
.
S2
[0-9]
S3
[0-9]
[1-9]
(_|[A-z]|[0-9])
S4
(_|[A-z])
S5
=
=
S6
=
S8
!
=
S0
<,>
S7
:{}();
S9
+-*
"
S10
/
\
S12
[0x20-0x7E]
S11
"
S13
*
[0x20-0x2E]
[0x30-0x7E]
\n
S14
*
S15
S16
/
/
[0x20-0x2E]
[0x30-0x7E]
S17
\n
S18