Tealang Interpreter

Tony Valentine

March 30, 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 Lexer Components

2.1.1 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.

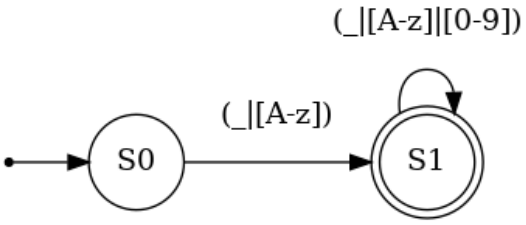

Figure 1: Identifier DFA

2.1.2 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.

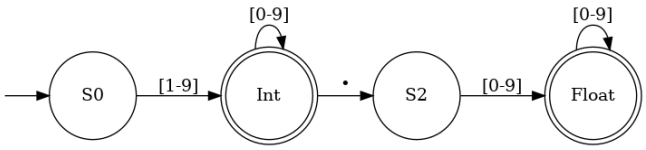

Figure 2: Integer and Float DFA

2.1.3 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.

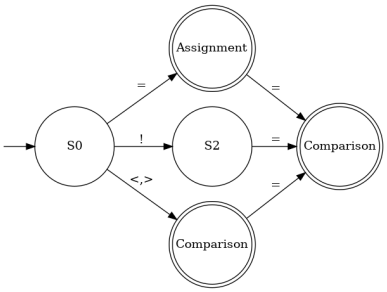

Figure 3: Comparison and Assignment DFA