Lexical Considerations

All Decaf keywords are lowercase. Keywords and identifiers are case-sensitive.

For example, 'if' is a keyword, but 'IF' is a variable name; 'name' and 'Name' are two different names referring to two distinct variables.

The reserved words are: **False, def, if, raise, None, del, import, return, True, elif, in, try, and, else, is, while, as, except, lambda, with, assert, finally, nonlocal, yield, break, for, not, class, from, or, continue, global, pass**

Comments are started by '#' and are terminated by the end of the line.

White space may appear between any lexical tokens. White space is defined as one or more spaces, tabs, page and line-breaking characters, and comments.

Keywords and identifiers must be separated by white space or a token that is neither a keyword nor an identifier. For example, 'thisfortrue' is a single identifier, not three distinct keywords.

If a sequence begins with an alphabetic character or an underscore, then it, and the longest sequence of characters following it forms a token.

String literals are composed of <char>s enclosed in double quotes.

A character literal consists of a <char> enclosed in single quotes.

Numbers in Decaf are 32-bit signed. That is, decimal values between -2147483648 and 2147483647.

If a sequence begins with '0x', then these first two characters and the longest sequence of characters drawn from [0-9a-fA-F] form a hexadecimal integer literal.

If a sequence begins with a decimal digit (but not '0x'), then the longest prefix of decimal digits forms a decimal integer literal.

A 'char' is any printable ASCII character (ASCII values between decimal value 32 and 126, or octal 40 and 176) other than quote ("), single quote ('), or backslash (\), plus the 2-character sequences '\"' to denote quote, '\'' to denote single quote, '\\' to denote backslash, '\t' to denote a literal tab, or '\n' to denote newline.

<foo>	means foo is a nonterminal.
foo	(in bold font) means that foo is a terminal i.e., a token or a part of a token
[x]	means zero or one occurrence of x, i.e., x is optional; note that brackets in quotes ’ [’ ’]’ are terminals.
x*	means zero or more occurrences of x.
x^+	a comma-separated list of one or more x’s
()	large braces are used for grouping; note that braces in quotes ’ {’ ’ }’ are terminals
|	separates alternatives

<prog> := <pipeline_def> <var_def> <pipe_block> EOF;

<pipeline_flag> := 'pipe';

<arg> := INT | CHAR | STRING | FLOAT | VAR_NAME;

<args> := <arg> (<comma> <space>* <arg>)*;

<function_args> := <left_par> <args> <right_par>;

<function_name> := VAR_NAME;

<pipeline_def> := <pipeline_flag> <space> <function_name> <left_par> <right_par> <two_points>
| <pipeline_flag> <space> <function_name> <function_args> <two_points>;

<single_pipe_symbol> := '|>';

<double_pipe_symbol> := '||>';

<tripple_pipe_symbol> := '|||>';

<pipe> := <tab> <single_pipe_symbol> <space> <function_name> <left_par> <right_par>
| <tab> <double_pipe_symbol> <space> <function_name> <left_par> <right_par>
| <tab> <tripple_pipe_symbol> <space> <function_name> <left_par> <right_par>
| <tab> <single_pipe_symbol> <space> <function_name> <function_args>
| <tab> <double_pipe_symbol> <space> <function_name> <function_args>
| <tab> <tripple_pipe_symbol> <space> <function_name> <function_args>;

<pipe_block> := <pipe> (<pipe>)*;

<tab> := '   ';

<space> := ' ';

<left_par> := '(';

<right_par> := ')';

<comma> := ',';

<two_points> := ':';

<var_name> := VAR_NAME | CHAR;

<var_def> := <tab> <var_name>;

NEWLINE : [\r\n]+ -> skip;
INT   : [0-9]+ ;
CHAR   : [a-zA-Z] ;
STRING : ['][a-zA-Z]+['] ;
FLOAT   : [0-9]+[.][0-9]+ ;
VAR_NAME : [a-zA-Z][a-zA-Z0-9]* ;
COMMENT : '#' ~('\r' | '\n')* -> skip;

Parsing Example

```
grammar Expr;
prog: pipeline_def var_def pipe_block EOF ;
pipeline_flag: 'pipe';
arg: (INT | CHAR | STRING | FLOAT | VAR_NAME);
```

```
args: arg (comma space* arg)*;

function_args: left_par args right_par;

function_name: VAR_NAME;

pipeline_def: pipeline_flag space function_name left_par right_par two_points
            | pipeline_flag space function_name function_args two_points;

single_pipe_symbol: '|>';

double_pipe_symbol: '||>';

tripple_pipe_symbol: '|||>';

pipe: tab single_pipe_symbol space function_name left_par right_par
    | tab double_pipe_symbol space function_name left_par right_par
    | tab tripple_pipe_symbol space function_name left_par right_par
    | tab single_pipe_symbol space function_name function_args
    | tab double_pipe_symbol space function_name function_args
    | tab tripple_pipe_symbol space function_name function_args;

pipe_block: pipe (pipe)*;

tab: '    ';

space: ' ';

left_par: '(';

right_par: ')';

comma: ',';

two_points: ':';

var_name: VAR_NAME | CHAR;

var_def: tab var_name;




NEWLINE : [\r\n]+ -> skip ;

INT     : [0-9]+ ;

CHAR    : [a-zA-Z] ;
```

```
STRING  : ['][a-zA-Z]+['] ;

FLOAT   : [0-9]+[.][0-9]+ ;

VAR_NAME : [a-zA-Z][a-zA-Z0-9]* ;

COMMENT : '#' ~( '\r' | '\n' )* -> skip;
```

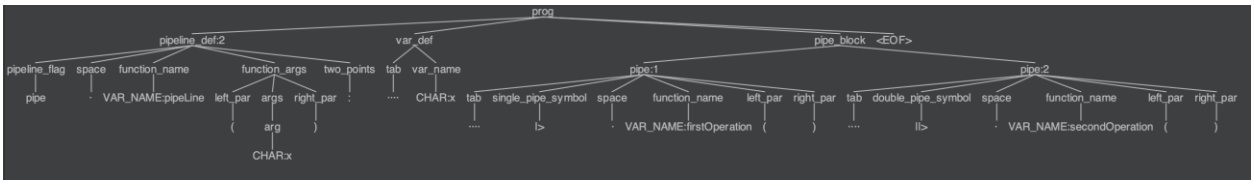
ANTLR Grammar definiton:

```
pipe pipeLine(x):

    x

    |> firstOperation()

    ||> secondOperation()
```



Semantics

The pipe operation is a mechanism for sequentially applying a series of functions to a given value. In the provided example, the pipe function facilitates this operation. The functions to be applied are specified as arguments to the pipe function, and they are executed in the order they are provided. The result of each function becomes the input for the next one, creating a sequential transformation of the original value.

1) **Keyword and Identifier Case Sensitivity:** Keywords must be written in lowercase, and identifiers are also case-sensitive. For instance, "bool" is a keyword, while "IF" could be an identifier.

2) **Comments:** Comments in the DSL start with the "#" symbol and terminate at the end of the line.

3) **White Space:** White space is allowed between any lexical tokens and is defined as one or more spaces, tabs, page and line-breaking characters, and comments.

4) **Keyword and Identifier Separation:** Keywords and identifiers must be

separated by white space or a token that is neither a keyword nor an identifier.

5) **String Literals:** String literals consist of characters enclosed in double quotes. Character literals are enclosed in single quotes.

6) **Numbers:** Numbers are 32-bit signed integers, ranging from -2147483648 to 2147483647.

7) **Character:** A <char> is any printable ASCII character except for quotes (both single and double) and backslashes, as well as specific escape sequences for special characters.

8) **String Literal Type:** String literals ("<char>*") must have the string type.

9) **Pipe Symbols:** Pipe symbols are special symbols to identify the functions that are applied on the reference variable.

10) **Reference Grammar:** Defines the grammar rules for constructing programs in the DSL, including statements, method declarations, variable declarations, expressions, literals, and identifiers.

Scope Rules

Declaration before Use: All identifiers, such as variables and functions, must be declared before they are used.

Method Invocation: A method can only be invoked or called by code that appears after its header. This rule ensures that methods are defined and their functionality is known before any attempts to use them in the program.

No Redefinition in the Same Scope: No identifier may be defined more than once in the same scope. This rule ensures that there is clarity and uniqueness in the naming of fields, methods, local variables, and formal parameters.

Locations

In terms of data types, can include both scalar and array types. Scalar locations might hold individual quotes or author names, typically represented as strings. On the other hand, array locations could accommodate collections of quotes or authors, potentially organized by theme, category, or other criteria. This flexibility allows for the effective management of diverse data structures, enhancing the capabilities of the DSL.