1 Lexical Conventions

1.1 Identifiers
An identifier is a series of alphanumeric characters including the underscore. Identifiers must not begin with a digit except when used within a graph.

1.2 Keywords
The following immutable identifiers are reserved as keywords:

```
bool  string  float  int  graph
if    else    elif   for  break
null  while   func   const void
```

1.3 Separators
A separator delimits two tokens. Belonging to this class of tokens are all whitespace characters, commas, colons, semicolons, and the paired delimiters parentheses, braces, and brackets.

1.4 Operators
The following operators are defined:

1.5 Literals

1.5.1 Numeric Literals
A numeric literal consists of a series of digits representing an integer, or a series of digits containing a decimal point or scientific e-notation representing a floating point. The number can be signed by prefixing it with an optional plus or hyphen character. Numbers will normally be stored using 32-bits unless overflow will occur, in which case 64-bits will be used.

1.5.2 String Literals
A string literal is a series of ASCII characters surrounded by single or double quotation marks. Non-printing characters, backslashes, and quotation marks of the same type surrounding the literal are represented within the literal by escaping them with a backslash. String literals separated from other string or numeric literals by whitespace will be concatenated together.
1.5.3 Graph Literals

A graph literal is a series of graph expressions surrounded by braces. Delimiting
braces delimiting the block of a function or conditional statement are excluded
from representation of a graph literal. Any identifiers within a graph literal are
unique to the scope of the literal.

1.6 Comments

Both single and multi-line comments are represented by an opening /* and a
closing */. Anything within the delimiting comment pair will be ignored by the
compiler.

2 Syntax Notation

2.1 Program Structure

A program is composed entirely of one or more expressions. An expression
always returns a value, although it must not always be used. There are no
true declarations, because even when not assigned a value, the declaration is an
expression returning null. For the purpose of clarity, however, declarations will
be differentiated from expressions.

2.2 Declarations

Variable declarations take place within any scope except that of a graph. Decla-
rations take the following form.

```
declaration:
    specifier identifier definition_opt
specifier:
    type-qualifier type-specifier
type-specifier
definition:
    = initializer
```

2.2.1 Type Specifiers

A valid type-specifier of the following must be present in every declaration.

```
type-specifier:
    void
    bool
    int
    float
    string
    graph
```
2.2.2 Initialization
Upon declaration, the variable can initialized to a user-defined value or to null. The value to the right of the equals sign is either an expression or a function. The value returned by the expression must be of equivalent types with the declaration.

initializer:
expression
function

2.3 Expressions
2.3.1 Primary Expressions
Primary expressions take any of the following forms of identifiers, literals, and parenthesized expressions. Identifiers and literals adhere to their previous descriptions, and an identifier must be properly typed and defined prior to use in a primary expression.

primary-expression:
identifier
literal
(expression)

2.3.2 Function Call
Function calls adhere to the following form where the preceding expression must have a type of func.

func-call:
expression (arg-list opt)

arg-list:
expression
arg-list , expression

2.3.3 Index-Of
The index-of operation requires the preceding expression to have a type of graph.

index-of:
expression [ expression ]


2.3.4 Unary Operators

A unary-operator is one of positivity, numerical negation, or bitwise negation.

\[
\text{unary-operator:}
\begin{align*}
+ \\
- \\
! 
\end{align*}
\]

2.3.5 Binary Operators

A binary-operator is one of the following in order of precedence (grouped by numbers), multiplication, division, modulo, addition, subtraction, bit shifts, inequalities, equalities, bitwise and, bitwise xor, bitwise or, logical and, logical xor, logical or.

\[
\text{bin-op-expression:}
\begin{align*}
\text{expression bin-op expression}
\end{align*}
\]

\[
\text{bin-op:}
\begin{align*}
1 & * \\
& / \\
& \% \\
2 & + \\
& - \\
3 & >> \\
& << \\
4 & < \\
& <= \\
& > \\
& >= \\
5 & == \\
& != \\
6 & k \\
7 & ^ \\
8 & | \\
9 & kk \\
10 & ^^ \\
11 & ||
\end{align*}
\]

2.3.6 Graph Initialization Literals

Within a graph literal/initialization, the following rules are in effect.

\[
\text{graph-init-expression:}
\begin{align*}
\text{graph-init-expression edge literal} \\
\text{literal edge literal}
\end{align*}
\]
edge:
  -> directed
  <> undirected