NSBL : Language Reference Manual

Chantal Galvez, Kunal Mishra, Lixing Pan and Jing Zhang

March 18, 2012

1 Introduction

This manual describes the NSBL language, which is intended for graph data structure manipulation.

2 Lexical Conventions

2.1 Tokens

There are six classes of tokens: identifiers, keywords, constants, string literals, operators, and other separators. Blanks, horizontal and vertical tabs, newlines, formfeeds, and comments as described below (collectively, ”white space”) are ignored except as they separate token.

Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been separated into tokens up to a given character, the next token is the longest string of characters that could constitute a token.

2.2 Comments

The characters /* introduce a comment, which terminates with the characters */.

The characters // also introduce a comment, which terminates with the newline.

2.3 Identifiers

An identifier is a sequence of letters and digits. The first character must be a letter; th underscore _ counts as a letter. Upper and lower case letters are different. Identifiers may have any length.

2.4 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

void	float	int	string
list	vertex	edge	graph
def	if	else	for
foreach	while	break	continue
return	outE	inE	strtV
endV	allV	allE	true
false	boolean		

2.5 Constants

There are several kinds of constants. Each has a data type; Sec.4.1 discusses the basic types.

```
constant :
  integer_constant
  float_constant
  boolean_constant

boolean_constant :
  true
  false
```

2.6 Integer Constants

An integer constant consisting of a sequence of digits is taken to be decimal. An integer constant is always signed.

2.7 Float Constants

A float condstant consists of an integer part, a decimal point, fraction part, an e or E, an optionally signed integer exponent.The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the e (E) or the exponent (not both) may be missing.

2.8 Boolean Constants

A boolean constant can either be true or false.

2.9 String Literals

A string literal, also called a string constant, is a sequence of characters surrounded by double quotes, as in "..." and has type string. The behavior of a program that attempts to alter a string literal is undefined.

Adjacent string literals are concatenated into a single string. Unlike C language, you do not need a '$\backslash 0$' to mark the end of a string. String literals do not contain newline or double-quote characters; in order to

represent them, escape sequence '$\backslash n$' and '$\backslash$"' will be needed.

3 Syntax Notation

In the syntax notation used in this manual, syntactic categories are indicated by `italic` type, and literal words and characters in $typewriter$ style. Alternative categories are usually listed on separate lines; in a few cases, a long set of narrow alternatives is presented on one line, marked by the phrase "one of." The syntax is summarized in Sec.10.

4 Meaning of Identifiers

Identifiers, or names refer to two things: functions and objects. An object, sometimes called a variable, is a location in storage, and its interpretation depends on two main attributes: its storage class and its type. The storage class determines the lifetime of the storage associated with the identified object; the type determines the meaning of the values found in the identified object. A name a so has a scope, which is the region of the program in which it is known, and a linkage, which determines whether the same name in another scope refers to the same object or function. Scope is discussed in Sec.9.

4.1 Basic Types

There are four fundamental types: int, float, string, boolean.

The int type, which is signed, is used to represent all integer numbers.

The float type, which is signed, is used to represent all decimal numbers or exponential numbers.

There is not char type defined here, string is the basic unit to represent a character or a sequence of characters. The string can be any length and there is no end of string symbol needed.

The boolean type has two values, true and false.

4.2 Derived Types

There are six derived types Vertex, edge, graph, list, functions, function_literal.

Vertex is the type to represent a node of a graph. Attributes can be added or deleted dynamically to the node. An attribute can be any one of the fundamental types.

Edge is the type to represent an edge of a graph. It has a outgoing vertex and an incoming vertex; it can also contain any number of attributes which can be added or deleted dynamically. The attribute can be any one of the fundamental types.

Graph is the type to represent a whole graph. It keeps lists of vertices and edges. Vertices and edges can be added or deleted dynamically.

List is the type to a list of objects. The objects can be any of the fundamental types, vertices and edges. The type of objects in the same list should be same.

Functions return objects of a given type.

Function_literal, after evaluating the conditional expression given to it, returns a boolean type.

5 Expressions

5.1 Primary Expressions

Primary expressions are attributes, identifiers, constants, strings or expressions in parentheses.

```
primary_expression :
  identifier
  attribute
  constant
  string
  ( expression )

attribute :
  @ identifier
```

An identifier is a primary expression, and its type is statically specified by its declaration.

An attribute is a primary expression consisting of the @ operator and an identifier. Its type dynamically depends the context.

A constant is a primary expression. Its type depends on its form as discussed in Sec.2.5.

A string literal is a primary expression. Its type is ”string”.

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression.

5.2 Postfix Expressions

The operations in postfix expressions group left to right.

```
postfix_expression :
  primary_expression
  primary_expression : primary_expression -> primary_expression
  primary_expression : primary_expression -> primary_expression MARK
     primary_expression
```

```
  postfix_expression ( argument_expression_list )
  postfix_expression ( )
  postfix_expression | pipe_property
  postfix_expression [ conditional_expression ]
  postfix_expression . identifier
  postfix_expression . graph_property

graph_property :
  ALL_VERTICES
  ALL_EDGES

pipe_property :
  OUTCOMING_EDGES
  INCOMING_EDGES
  STARTING_VERTICES
  ENDING_VERTICES

argument_expression_list :
  assignment_expression
  argument_expression_list , assignment_expression
```

5.2.1 Edge Assignment

A postfix expression followed by colon, primary expression, $\rightarrow$ operator and another primary expression is a postfix expression assigning the starting vertex and ending vertex for an directed edge. Obviously, the first primary expression must have type ”edge”, and the second and third primary expressions must have type ”vertex”. The additional MARK operator and fourth primary expression is optional, used to describe the feature of the edge. The type of the fourth primary expression must be ”string”. The postfix expression of edge assignment is finally evaluated as the edge corresponding to the first primary expression.

5.2.2 Function Calls and Function Literal Calls

The syntax of function call and function literal call are the same, a postfix expression followed by parentheses containing a possibly empty, comma-separated list of assignment expressions, which constitute the arguments to the function or function literal.

In preparing for the call to a function or function literal, a copy is made of each argument; all argument-passing is strictly by value.

5.2.3 Pipe Operations

A pipe operation is a postfix expression of type ”list”, followed by the | operator and pipe property. The allowed choices of pipe property depend on the type of the list element. If list element has type of ”vertex”, the pipe property can be OUTCOMING_EDGES or INCOMING_EDGES; similarly, if the type of list element is ”edge”, the pipe property can be STARTING_VERTICES or ENDING_VERTICES.

This pipe operation is eventually evaluated by appending all piped properties to an empty list, and the return type is ”list” whose element has the type specified by pipe property.

5.2.4 Match Operations

A match operation is a postfix expression of type ”list”, followed by and conditional expression in square brackets. The type of first operand must have type list, and the type of conditional expression must have type $boolean$. The return of match operation is a list containing all elements satisfying the conditional expression.

An example is

```
list result = v[ @name=="Tom" && @age>17 ];
```

5.2.5 List References

List References share the same syntax as match operation, but requires different type of conditional expression, which must be int. The starting index of a list is $zero$, and the return of list reference $E1[E2]$ is the $(E2 + 1)$-th element of the list.

5.2.6 Property References

A postfix expression followed by a dot followed by an identifier or graph property is a postfix expression. The first operand expression must be a graph, vertex or edge, and the identifier must name an existing property of the first operand. The value is the named property of the first operand, and its type is the type of the property. The existence and type checks are dynamic, depend on the implementation, and are not included in this grammar.

5.3 Unary Operations

Expressions with unary operators group right-to-left.

```
unary_expression :
    postfix_expression
    unary_operator unary_expression

unary_operator : one of
  + - !
```

5.3.1 Unary Plus Operator

The operand of the unary + operator must be integer or float type, and the result is the value of the operand.

5.3.2 Unary Minus Operator

The operand of the unary - operator must be integer or float type, and the result is the negative of its operand.

5.3.3 Logical Negation Operator

The operand of the ! operator must be boolean type, and the result is true if the value of its operand compares equal to false (boolean). The type of the result is `boolean`.

5.4 Casts

A unary expression preceded by the parenthesized name of a type causes conversion of the value of the expression to the named type.

```
cast_expression :
  unary_expression
  ( declaration_specifiers ) cast_expression
```

The construction is called a $cast$. The cast is allowed only between int and $float$.

5.5 Multiplicative Operators

The multiplicative operators * and / group left-to-right.

```
multiplicative_expression :
  cast_expression
  multiplicative_expression * cast_expression
  multiplicative_expression / cast_expression
```

The operand of * and / operators have integer or float type. The usual arithmetic conversions are performed on the operands between type int and type $float$, and predict the type of the result.

The binary * operator denotes multiplication.

The binary / operator yields the quotient of the division of the first operand by the second; if the second operand is 0, the result is undefined.

5.6 Additive Operators

The additive operators + and - group left-to-right.

```
additive_expression :
  multiplicative_expression
  additive_expression + multiplicative_expression
  additive_expression - multiplicative_expression
```

The operands of the + and - operator can have type int or float. The result of the + operator is the sum of the operands; the result of the - operator is the differece of the operands. The usual arithmetic conversions are performed when necessary, and predict the type of the result.

The operands of the + operator can also have string type, and the result of the + operator is the concatenation of the operands.

5.7 Relational Operators

The relational operands group left-to-right.

```
relational_expression :
  additive_expression
  relational_expression <  additive_expression
  relational_expression >  additive_expression
  relational_expression <= additive_expression
  relational_expression >= additive_expression
```

The operators $<$(less), $>$(greater), $<=$(less or equal) and $>=$(greater or equal) all yield false if the specified relation is false and true if it is true. The type of the result is `boolean`. The operand of the four relational operators must have type int or float. The example $a < b < c$ is parsed as $(a < b) < c$, and the type of $(a < b)$ is boolean and is incompatible with the second $<$ operator.

5.8 Equality Operators

```
equality_expression :
  relational_expression
  equality_expression == relational_expression
  equality_expression != relational_expression
```

The $==$ (equal to) and the $!=$ (not equal to) operators yield boolean result on the specified relation, which is similar to the relational operators except for lower precedence. The two operands of $==$ and $!=$ operators must have identical type, which can be boolean, number (int or float) or string.

5.9 Logical AND Operator

```
logical_AND_expression :
  equality_expression
  logical_AND_expression && equality_expression
```

The && operator groups left-to-right. The operands of && operator have boolean type. If both operands are true, the result is true; otherwise, false is returned. && guarantees left-to-right evalution: the first operand is evaluated first, and the second operand is evaluated only when the evaluated result of first operand is true. The result is `boolean`.

5.10 Logical OR Operator

```
logical_AND_expression
logical_OR_expression || logical_AND_expression
```

The || operator groups left-to-right. The operands of || operator have boolean type. If either operand is true, the result is true; if both operands are false, the result is false. || guarantees left-to-right evalution: the first operand is evaluated first, and the second operand is evaluated only when the evaluated result of first operand is false. The result is `boolean`.

5.11 Assignment Expressions

There are six assignment operators; all group right-to-left.

```
assignment_expression :
  logical_OR_expression
  unary_expression assignment_operator assignment_expression

assignment_operator : one of
  = += -= *= /= <:
```

5.11.1 Assignment Operators

In the simple assignment with =, the value of first operand replaces by the value of second operand. The operands of = operator must have the same type, which can be any type of boolean, int, float, string, list, vertex, edge and graph.

An expression of the form $E1\ op = E2$ is equivalent to $E1 = E1\ op\ (E2)$ except that E1 is evaluated only once.

5.11.2 Eat Operator

The first operand of <: operator can have type of vertex, edge, or graph. If it is vertex or edge, the second operand can have type of boolean, int, float, or string; the <: operation is to add the value of the second operand as a new property to the first operand, and the name of the property is the name of the second operand. The property value can be referenced via property operator (Sec.5.2.6). An example is

```
vertex v;
v <: (name="Tom") ;
v <: (age=17) ;
```

If the first operand has type of graph, the second operand can have type of vertex, edge, or graph; the <: operation is to append a vertex, an edge or a subgraph into the graph specified by the first operand. An example is

```
graph g;
vertex v1;
```

```
g <: v1;
```

The type and value of the result are the type and value of the first operand.

5.12 Comma Operator

```
expression :
  assignment_expression
  expression , assignment_expression
```

A pair of expression separated by a comma is evaluated left-to-right, and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand.

6 Declarations

Declaration specify the interpretation given to each identifier. Declarations have the form

```
declaration :
  declaration_specifiers init_declarator_list ;
```

The declarators in the init-declarator list contain the identifiers being declared.

```
declaration_specifiers :
  basic_type_specifier

init_declarator_list :
  init_declarator
  init_declarator_list , init_declarator

init_declarator :
  declarator
  declarator = initializer
```

6.1 Type Specifiers

The type-specifiers are

```
basic_type_specifier : one of
  void boolean int float string list vertex edge graph

function_literal_type_sepcifier :
  func
```

At most, only one type-specifier can be given in a declaration. If the type-specifier is missing from a declaration, a syntax error is generated; in other words, there is no default type assumed.

6.2 Declarators

Declarators have the syntax:

```
declarator :
  direct_declarator

direct_declarator :
  identifier
  identifier ( parameter_list )
  identifier ( )
```

6.2.1 Function Declarators

The function declaration is an identifier followed by a parenthesized list of parameters. The type of the identifier of D in $T\ D(parameter\_list)$, where T is declaration specifier, is "function with arguments $parameter\_list$ returning T". The syntax of the parameters is

```
parameter_list :
  parameter_declaration
  parameter_list , parameter_declaration

parameter_declaration :
  declaration_specifiers identifier
  function_literal_type_sepcifier identifier
```

The parameter list specifies the type of the parameters. All the types (Sec.6.1) are allowed to be the function parameters, including the function literal type (Sec.6.2.2).

6.2.2 Function Literal Declaration

Function literal declarations have the form

```
function_literal_declaration :
  function_literal_type_sepcifier declarator = compound_statement ;
  function_literal_type_sepcifier declarator : declaration_specifiers =
     compound_statement ;
```

The difference of the two declaration forms is that the second one explicitly specifies the return type as $declaration\_specifiers$, while the first one omits the return type. If the return type is omitted in function literal declarations, the default return type is the type of the last statement in $compound\_statement$. The allowed return types of function literal are all types except the function literal itself.

A complete example of function literal declaration is

```
func add(int a, int b):int = { a+b; };
```

Similar to simple declaration, the function literal declarations are allowed to be nested within the function definition.

The function literal can be used together with $attribute$ and match operator. The $attributes$ used within the $compound\_statement$ are not required to be declared in the $argument\_list$. An example is

```
func cond() = { @name=="Tom" && age>17 ; } ;
// list out all Vertices match the condition
list v = g.allV[cond()];
```

6.2.3 Initialization

An initial value may be specified for an identifier during its declaration via initializer. The initializer is preceded by $=$, is either an expression, or a list of initializers nested in square bracket.

```
initializer :
  assignment_expression
  [ initializer_list ]
  [ ]

initializer_list :
  initializer
  initializer_list , initializer
```

The initializer for a list is either an expression of the same type, or a square-bracket-enclosed list of initializers for its members in order; empty list is allowed. The initializers in the square bracket for the initialization of a list must have the same type, which can be one of boolean, int, float, string, vertex, edge.

The initializer for all types except list is an expression of the same type.

7 Statements

Statements are executed in sequence. Statements are executed for their effect, and do not have values. They fall into several groups.

```
statement :
  expression_statement
  compound_statement
  selection_statement
  iteration_statement
  jump_statement
  declaration_statement
```

7.1 Expression Statement

Most statements are expression statements, which have the form

```
expression_statement :
  expression ;
  ;
```

Most expression statements are assignment or function calls. All side effects from the expression are completed before the next statement is executed. If the expression is missing, the construction is called a null statement; it is often use to supply an empty body to an iteration statement.

7.2 Compound Statements

So that several statements can be used where one is expected, the compound statement (also called ”block”) is provided. The body of a function definition is a compound statement.

```
compound_statemnet :
  { }
  { statement_list }

statement_list :
  statement
  statement_list statement
```

If an identifier in the statement-list was in scope outside the block, the outer declaration is suspended within the block, after which it resumes its force. An identifier may be declared only once in the same block. Initialization of objects is performed each time the block is entered at the top, and proceeds in the order of the declarators.

7.3 Selection Statements

Selection statements choose one of the two flows of control.

```
selection_statement :
  if ( expression ) statement
  if ( expression ) statement else statement
```

In both forms of the is statement, the expression has a boolean type after evaluated. If the boolean value is true, the first substatement is executed. If the boolean value is false in the second form, the second substatement is executed. The else ambiguity is resolved by connecting an els with the last encountered else-less if at the same block nesting level.

7.4 Iteration Statements

```
iteration_statement :
  while ( expression ) statement
  for ( expression ; expression ; expression ) statement
  for ( expression ; expression ; ) statement
  for ( expression ; ; expression ) statement
  for ( ; expression ; expression ) statement
```

```
  for ( expression ; ; ) statement
  for ( ; expression ; expression ) statement
  for ( ; expression ; ) statement
  for ( ; ; ) statement
  foreach ( identifier : postfix_expression ) statement
```

In the while statement, the substatement is executed repeatedly so long as the value of the expression remains true; the expression must have boolean type. The test, including all side effects from the expression, occurs before each execution of the statement.

In the for statement, the first expression is evaluated once, and thus specifies initialization for the loop. There is no restriction on its type. The second expression must have boolean type; it is evaluated before each iteration, and if it becomes false, the for is terminated. The third expression is evaluated after each iteration, and thus specifies a re-initialization for the loop. There is no restriction on its type. Side-effects from each expression are completed immediately after its evaluation. If the substatement does not contain continue, a statement

```
for ( expression1 ; expression2 ; expression3 ) statement
```

is equivalent to

```
expressin1 ;
while ( expression2 ) {
  statement
  experssion3 ;
}
```

Any of the three expressions may be dropped. A missing second expression makes the implied test equivalent to test a true constant.

In the foreach statement, the identifier must have a vertex or edge type and is assigned a vertex or edge objects at each iteration. The postfix expression must have a list type, which consists of elements of vertex of edge type. The identifier iterate over all the elements in the list and do the substatement once at in each iteration.

7.5 Jump Statements

Jump statements transfer control unconditionally.

```
jump_statement :
  break ;
  continue ;
  return expression ;
  return
```

A continue statement may appear only within an iteration statement. It causes control to pass to the loop-continuation portion of the smallest enclosing such statement.

A break statement may appear only in an iteration statement and terminates execuation of the smallest enclosing such statement; control passes to the statement following the terminated statement.

A function returns to its caller by the return statement. When return is followed by an expression, the value is returned to the caller of the function. The expression type must agree with the function return type.

Flowing off the end of a function is equivalent to a return with no expression. In either case, the returned value is undefined and the function should have type void.

7.6 Declaration Statements

Declaration statements have the syntax

```
declaration_statement :
  declaration
  function_literal_declaration
```

Simple declaration and function literal declaration can be specified in the declaration statements.

8 External Statements

The unit of input provided to the NSBL compiler is called a translation unit; it consists of a sequence of external statements, which are either statement or function definitions.

```
translation_unit :
  external_statement
  translation_unit external_statement

external_statement :
  function_definition
  statement
```

The scope of declarations within the external statements persists to the end of the translation unit in which they are declared. Only at the level of external statement may the function definition be given, therefore nested function is not allowed.

8.1 Function Definitions

Function definition have the form

```
function_definition :
  declaration_specifiers declarator compound_statement
```

A function may return any type (Sec.6.1) except the function literal type. The declarator in a function declaration must specify explicitly that the declared identifier has function type; that is, it must contain one of the forms

```
identifier ( parameter_list )
identifier ( )
```

The syntax of function definition of NSBL is similar to the new-style syntax in Language C, but not compatible to the old-style syntax.

8.2 External Statements

External statements refers to the statements which are located outside functions, and are automatically executed after the program is loaded. Unlike C language, in NSBL language there does not exist a starting function (in C language, that is $main$ function); and the external statements are conceptually equivalent to the $main$ function in C.

External statements can include $declaration\_statement$. All objects declared within external statements have global scope, which persists to the end of the translation unit in which the external statements are declared, and are visible in functions.

Each object or function must have exactly one definition.

9 Scope

At this time, a program must be compiled at one time: the source text must be kept in one file. Therefore, the external function or data does not exist in NSBL language.

The scope of an identifier is the region of the program text within which the identifier's characteristics are understood. Each type has a name space, and the name spaces of different types are disjoint. Identifiers in different name spaces that do not interfere with one another; the same identifier may be used for different purposes, even in the same scope, if the uses are in different name spaces.

The scope of an object or function identifier in an external statement begins at the end of its declarator and persists to the end of the translation unit in which it appears. The scope of a parameter of a function definition begins at the start of the block defining the function, and persists through the function; the scope of a parameter in a function declaration ends at the end of the declarator. The scope of an identifier declared at the head of a block begins at the end of its declarator, and persists to the end of the block.

If an identifier is declared in external statement, it can be shadowed by the same identifier in the same name space within a function.

10 Grammar

Below is the implementation of the grammar that was given throughout this language specification in YACC.

```
%{
#include <stdio.h>
#include <stdlib.h>

extern FILE *yyin; /* Input for yacc parser. */
extern void yyerror(char *str); /* Our version. */
extern int yywrap(void); /* Our version. */
extern int yylex(void); /* Lexical analyzer function. */
extern int yyparse(void); /* Parser function. */

%}

/*************************
 *      TOKEN LIST       *
 *************************/
/* TYPE RELATED */
%token VOID BOOLEAN INTEGER FLOAT STRING LIST VERTEX EDGE GRAPH
%token IDENTIFIER INTEGER_CONSTANT FLOAT_CONSTANT STRING_LITERAL
%token TRUE FALSE
/* FUNCTIONS RELATED */
%token FUNC_LITERAL
/* GRAPH RELATED */
%token OUTCOMING_EDGES INCOMING_EDGES STARTING_VERTICES ENDING_VERTICES
%token ALL_VERTICES ALL_EDGES
/* OPERATOR */
%token OR AND
%token EQ NE
%token GT LT GE LE
%token ADD_ASSIGN SUB_ASSIGN MUL_ASSIGN DIV_ASSIGN
%token EAT ARROW PIPE AT MARK
/* CONTROL */
%token IF ELSE
%token FOR FOREACH WHILE
%token BREAK CONTINUE
%token RETURN

/*************************
 *  PRECEDENCE & ASSOC   *
 *************************/
%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE

/*************************
 *     START SYMBOL      *
 *************************/
%start translation_unit

%%
```

```
/*************************
 *   BASIC CONCEPTS      *
 *************************/
translation_unit
    : external_statement
    | translation_unit external_statement
    ;

/*************************
 *   STATEMENTS          *
 *************************/
external_statement
    : function_definition
    | statement
    ;

statement
    : expression_statement
    | compound_statement
    | selection_statement
    | iteration_statement
    | jump_statement
    | declaration_statement
    ;

expression_statement
    : expression ';'
    | ';'
    ;

statement_list
    : statement
    | statement_list statement
    ;

compound_statement
    : '{' '}'
    | '{' statement_list '}'
    ;

selection_statement
    : IF '(' expression ')' statement %prec LOWER_THAN_ELSE ;
    | IF '(' expression ')' statement ELSE statement
    ;

iteration_statement
    : WHILE '(' expression ')' statement
    | FOR '(' expression ';' expression ';' expression ')' statement
    | FOR '(' expression ';' expression ';' ')' statement
    | FOR '(' expression ';' ';' expression ')' statement
    | FOR '(' expression ';' ';' ')' statement
    | FOR '(' ';' expression ';' expression ')' statement
    | FOR '(' ';' expression ';' ')' statement
    | FOR '(' ';' ';' expression ')' statement
```

```
    | FOR '(' ';' ';' ')' statement
    | FOREACH '(' IDENTIFIER ':' postfix_expression ')' statement
    ;

jump_statement
    : BREAK ';'
    | CONTINUE ';'
    | RETURN expression ';'
    | RETURN ';'
    ;

declaration_statement
    : declaration
  | function_literal_declaration
    ;

/*************************
 *   EXPRESSIONS          *
 *************************/

expression
    : assignment_expression
    | expression ',' assignment_expression
    ;

assignment_expression
    : logical_OR_expression
    | unary_expression assignment_operator assignment_expression
    ;

assignment_operator
    : '='
    | ADD_ASSIGN
    | SUB_ASSIGN
    | MUL_ASSIGN
    | DIV_ASSIGN
    | EAT
    ;

logical_OR_expression
    : logical_AND_expression
    | logical_OR_expression OR logical_AND_expression
    ;

logical_AND_expression
    : equality_expression
    | logical_AND_expression AND equality_expression
    ;

equality_expression
    : relational_expression
    | equality_expression EQ relational_expression
    | equality_expression NE relational_expression
    ;
```

```
relational_expression
    : additive_expression
    | relational_expression LT additive_expression
    | relational_expression GT additive_expression
    | relational_expression LE additive_expression
    | relational_expression GE additive_expression
    ;

additive_expression
    : multiplicative_expression
    | additive_expression '+' multiplicative_expression
    | additive_expression '-' multiplicative_expression
    ;

multiplicative_expression
    : cast_expression
    | multiplicative_expression '*' cast_expression
    | multiplicative_expression '/' cast_expression
    ;

cast_expression
    : unary_expression
    | '(' declaration_specifiers ')' cast_expression
    ;

unary_expression
    : postfix_expression
    | unary_operator cast_expression
    ;

unary_operator
    : '+'
    | '-'
    | '!'
    ;

postfix_expression
    : primary_expression
    | primary_expression ':' primary_expression ARROW primary_expression
  | primary_expression ':' primary_expression ARROW primary_expression MARK
     primary_expression
    | postfix_expression '(' argument_expression_list ')'
    | postfix_expression '(' ')'
    | postfix_expression PIPE pipe_property
    | postfix_expression '[' logical_OR_expression ']'
    | postfix_expression '.' IDENTIFIER
    | postfix_expression '.' graph_property
    ;

primary_expression
    : attribute
    | IDENTIFIER
    | constant
```

```
    | STRING_LITERAL
    | '(' expression ')'
    ;

graph_property
    : ALL_VERTICES
    | ALL_EDGES
    ;

pipe_property
    : OUTCOMING_EDGES
    | INCOMING_EDGES
    | STARTING_VERTICES
    | ENDING_VERTICES
    ;

argument_expression_list
    : assignment_expression
    | argument_expression_list ',' assignment_expression
    ;

attribute
    : AT IDENTIFIER
    ;

constant
    : INTEGER_CONSTANT
    | FLOAT_CONSTANT
    | TRUE
    | FALSE
    ;

/*************************
 *   DECLARATION         *
 *************************/

function_literal_declaration
    : function_literal_type_sepcifier declarator '=' compound_statement ';'
  | function_literal_type_sepcifier declarator ':' declaration_specifiers '='
     compound_statement ';'
    ;

function_definition
    : declaration_specifiers declarator compound_statement
    ;

function_literal_type_sepcifier
  : FUNC_LITERAL
  ;

basic_type_specifier
    : VOID
  | BOOLEAN
    | INTEGER
```

```
    | FLOAT
    | STRING
    | LIST
    | VERTEX
    | EDGE
    | GRAPH
    ;

declaration
    : declaration_specifiers init_declarator_list ';'
    ;

declaration_specifiers
    : basic_type_specifier
    ;

init_declarator_list
    : init_declarator
    | init_declarator_list ',' init_declarator
    ;

init_declarator
    : declarator
    | declarator '=' initializer
    ;

declarator
    : direct_declarator
    ;

direct_declarator
  : IDENTIFIER
    | IDENTIFIER '(' parameter_list ')'
    | IDENTIFIER '(' ')'
    ;

parameter_list
    : parameter_declaration
    | parameter_list ',' parameter_declaration
    ;

parameter_declaration
    : declaration_specifiers IDENTIFIER
  | function_literal_type_sepcifier IDENTIFIER
    ;

initializer
    : assignment_expression
    | '[' initializer_list ']'
  | '[' ']'
    ;

initializer_list
    : initializer
```

```
    | initializer_list ',' initializer
    ;

%%

void yyerror(char *s) {
    printf("%s\n", s);
}

int main(int argc, char * const * argv) {
    if (argc<=1) { // missing file
        fprintf(stdout, "missing input file\n");
        exit(1);
    }
    yyin = fopen(argv[1], "r");
    yyparse();
    fclose(yyin);
    return 0;
}
```

../../NSBL/src/Parser.y