Language Reference Manual

1 Program source

1.1 Whitespace

Whitespace characters such as spaces and tabs are used to separate tokens in the input and are discarded during parsing.

1.2 Comments

Single line and multi-line comments are supported. Single line comments are denoted by the characters // and continue to the end of the line. These may be placed on the same line as the source code.

Multi-line comments begin with a /* and end with a */. They may not be embedded within each other.

1.2 Semicolons

Semicolons are used to terminate statements. Multiple statements may be placed on the same line by placing semicolons between them. Multiple consecutive semicolons are considered as a single statement terminator.

1.4 Identifiers

All identifiers begin with a letter and may be followed by zero or more letters, digits or underscores. Identifiers are case sensitive.

1.5 Literals

Literals in the language can be integers, characters, strings, booleans or floating point numbers.

1.6 Keywords

The following identifiers are reserved as keywords in the language:

char	int	float	string	boolean
for	while	if	else	break
continue	true	false	return	function
Attribute	Part	Composite		

1.7 Other tokens

The following characters have meaning in the language:

```
+     -     /     *     =     >     <     >=    <=    ==

!     !=    ||    &&    {     }     (     )     [     ]

,     .
```

2 Fundamental types

2.1 Integers

An integer consists of one or more consecutive digits. Integers are assumed to be base 10 and may not begin with a zero.

2.2 Floats

These are arbitrary precision decimals and are composed of three parts, all of which are optional:

- An integer part composed of digits
- A decimal part consisting of a period followed by digits
- An exponent part consisting of an e or E, followed by an optional + or -, followed by digits

Either the integer or the decimal part must be present.

2.3 Characters

Characters consist of a single character enclosed in single quotes

2.4 Strings

These are sequences of characters. String literals are sequences of characters enclosed in double quotes. The double quote character is escaped with a \

2.5 Booleans

The boolean constants true and false are language keywords and represent logical truth and falsehood

2.6 Attributes

Attributes are labels for primitives associated with parts. Attribute definitions must specify a primitive type.

Examples of attribute definitions are:

```
Attribute Sequence string;
Attribute Strength float;
```

3 Composite types

3.1 Arrays

These are one dimensional fixed length zero indexed arrays. Arrays are homogeneous ie. they may consist of objects of a single type. Array literals may be declared by enclosing a list of objects in square brackets and separating the elements by commas.

3.2 Parts

Part definitions define prototypes for standard biological parts. Parts can be instantiated only after the prototypes are defined. Parts have attributes which must be defined before parts can be defined.

```
Attribute Sequence string;
Attribute Name string ;
Attribute Compatibility string;
Attribute Strength float;

// define the prototype for promoters
Part Promoter(Name, Sequence, Compatibility, Strength);
```

3.3 Composites

Composites represent composites of biological parts or other composites. The same part or composite can be reused in a composite declaration.

```
// instantiate the BioBricks promoter BBa_I14018 based on the prototype
// declared above
Promoter Bba_I14018(“BBa_I14018”, “tgtaagtttatacataggcgagtactctgttatgg”,
“RFC21”, 0.5);
// instantiate the BioBricks RBS Bba_J63003
RBS Bba_J63003(“BBa_J63003”, “cccgccgccaccatggag”, “RFC21”);
// instantiate the BioBricks terminator BBa_B1002
Terminator Bba_B1002(“BBa_B1002”, “cgcaaaaaaccccgcttcggcggggttttttcgc”,
“RFC21”);

// create an assembly, represented by a composite; composites may contain other
//composites
Composite Assembly1( Bba_I14018, Bba_J63003, Bba_B1002);
```

4 Expressions

4.1 Arithmetic expressions

Arithmetic expressions consist of binary operators, the unary negation operator and parenthesis. Parentheses have the highest precedence, followed by the unary negation operator, followed by * and /, and then + and -. Arithmetic operations may be performed on integers as well as floats.

4.2 String concatenation

The + operator is overloaded for performing string concatenation.

4.3 Relational expressions

These consist of expressions involving >, <, >=, ==, != and <=. These operations may be performed on integers or floats, however operations on floats may not yield expected results.

4.4 Logical expressions

Logical expressions involve the !, || or && operators. These are performed on boolean operands and return the boolean values of true or false.

4.5 Attribute access expressions

Part attributes may be accessed using the dot operator as in:

```
if (promoter1.RelativeStrengh > 5.0) ….
```

4.6 Composite part access expressions

Parts within a composite may be accessed using the square bracket notation. Parts within a composite are zero indexed. Using the declarations in 3.3,

```
Promoter Bba_I14018 = Assembly1[0];
```

4.7 Array access expressions

Array are zero indexed, and their elements can be accessed using the square bracket notation.

4.8 Function calls

Functions are invoked by specifying the name of the function followed by a comma-separated list of parameters contained within “(“ and “)”. The parameter list is optional but the parentheses are not. Functions may have return types, in which case, the expression has the same type as the function.

4.9 Identifiers and literals

As specified in 1.4 and 1.5

5 Statements

Statements could be expressions or one or more of the statement types described below and terminated by a semicolon. If the statement consists of an expression only, its value is discarded. Statements may be grouped in blocks. Program flow proceeds from top to bottom, unless a conditional or iterative statement is encountered.

5.1 Statement blocks

Statements may be grouped between { and }. Each statement within the block must be terminated with a semicolon.

5.2 Prototype definitions

As defined in 3.2

5.3 Declarations

All variables must be declared before they can be used. Declaration may be accomplished in conjunction with assignment. The declaration syntax is:

<*type*> <*identifier*>;

To declare an array, use:

<*type*> <*identifier*>[<*size*>];

For declaration with assignment:

<*type*> <*identifier*> = <*expression*>;

examples:

```
int myVar;
int myVar2 = 10;
int myArray[5];

int myArray2[5] = [1,2,3,4,5];
```

5.4 Assignment

The assignment operator = is used to assign an expression to an identifier.

Identifiers must be declared before they can be assigned. Declaration and assignment may be performed in the same statement.

For the variable declared in 5.3, we have:

myVar = 10;

5.5 Conditional statements

The if statement may have an optional else part. The two forms of the if statement are:

if <*boolean expression*> <*statement*>
if <*boolean expression*> <*statement*> else <*statement*>

If-else semantics follow C semantics. <*statement*> may include multiple statements within { and }.

5.6 Iterative statements

These have two forms:

while (<*boolean expression*>) <*statement*>

for *(<expression-1>; <expression-2>; <expression-3>) <statement>*

These follow C semantics.

5.7 Function definitions

These follow the syntax:

<*type*> function_name <*variable list*> {
    <*statement*>
}

Functions may not be nested or recursive. The return type may also be an array or a user defined type. Functions may return void.

5.8 Return

The return statement occurs within function definitions and may return void or an expression.

5.9 Break

The break statement occurs in loops and returns control to the statement following the loop.

5.10 Continue

The continue statement occurs within loops and causes program flow to begin the next iteration.

5.11 Constraints

Constraints define rules for constructing assemblies. Composites are parsed and validated against constraints.

// examples of constraints
Start → <PlasmidBackbone><Prefix><Cassette><Suffix>
Cassette → <Promoter><Cistron><Terminator>
Cistron → <RBS><Gene>

In this manner the programmer is constrained to build only meaningful assemblies.

6 Predefined functions

6.1 validate()

The validate function validates composite assemblies against the constraints that have been defined.

6.2 printSequence()

Prints the genetic sequence of a part or an assembly of parts.

6.3 printDiagram()

Prints a schematic diagram of an assembly by associating an icon with each part.

6.4 generateMarkup()

Generates XML markup representing the assembly, which can be input to external simulation software.

Example usage of these function is as follows:

```
// validate the assembly against the declared constraints and then print sequence, print
// diagram and generate markup
if (Assembly1.validate()) {
    Assembly1.printSequence();
    Assembly1.printDiagram();
    Assembly1.generateMarkup("/usr/local/home/user1/assembly1markup.xml");

}
```