L++ Manual

Written by Dr. Daehwan Kim lab at the University of Texas Southwestern Medical Center
Updated: Dec 22, 2022

1 Introduction

L++ is a powerful biology-oriented, high-level programming language. It is designed to abstract away microscopic interactions at a programmer's level via familiar syntaxes while providing remarkable insight into intricate simulations when necessary.

The biological interpreter is written in C and C++, which parses and transcompiles L++ experimental code into Python to run efficient matrix-based, stoichiometric data calculations to run complex simulations.

This syntax tutorial aims to comprehensively introduce a variety of fundamental data types and structures virtually found in any simulation context, whether it's high-level at an experimental level or low-level at the molecular level.

2 Syntax

This chapter comprehensively covers all the types of syntaxes needed to get your first L++ program up and running to simulate experimental data. Many examples are provided along the way to help understand how and when each type of syntax is used.

Similar to other languages, L++ comments can be performed by `//`, for single line comments, or `/* <multi-line comment> */` , for block-style comments.

L++ is a typeless language, meaning it doesn't require data typing for more code efficiency. Data Types can be inferred, but types can be used to the programmer's desire as well. In this case, both statements shown below would compile and run.

```
int x = 1;
 x = 10;
```

In the example above, since `x` is initially declared as an integer (`int`), it can only be reassigned as another value of the same type. Reassigning it as any other data type will result in a compilation error. However, if `x` was never declared as an integer type, and rather just as

```
x = MoleculeA;
```

Its type depends on the context, and can therefore be reassigned to another experimental structure data type like so:

```
x = MoleculeB;
```

We'll cover this more later in section …

2.1 Basic Types

Experiments run entirely on numbers and the entities those numbers correspond to. These types compose the fundamental structure of the data that drives L++ programs.

Numbers

Integers and floating point numbers (decimals) are both supported in L++ for the respective use cases. As the language is largely based on the context of which the syntax is used in, the type of number used in a specific use case matters. For example, in cases such as chemical coefficients, the molarity of a chemical substance, or specifying a timestep may allow for integers or floating points, but the number of pipettes being used in an experiment must be a whole number. Using the wrong number type depending on the context will result in a compilation error.

```
int a = 5;
int b = 2.0;    // doesn't work
float c = 10.0;
float d = 6;    // works
```

Furthermore, units can follow numerical units to specify the experimental context. For example, the concentration of a solution would be given via molarity or molality. Here are a couple of examples; further references regarding units can be found in 4.7.

```
R = 5nM;
B = 0.1nM;
Am = 500nM;
```

Booleans

Similar to any other language, booleans can be expressed through `true` or `false`, or numbers through `1` or `0`. Nonzero numbers assigned to boolean types will be evaluated as `true`, while zero will be evaluated as `false`. They are commonly used within the context of control statements. Here are some examples:

```
bool is_true = true;
```

```
bool is_false = false;
x = true;
y = false;
```

Molecules and Chemical Substances

Molecules and chemical substances, both organic and inorganic, are the key components in simulating experiments. They're one of the most common data types that will be found and used across L++ code, especially within higher level abstractions and more advanced L++ data structures that will be explained further in section ___.

L++ supports chemical equation parsing, such as $10H_2 + 5O_2 \rightarrow 10H_2O$; as such, the language supports coefficient-optional chemical formulas in order to declare specific molecular substances. The number of atoms of each element within a formula does not require a subscript; they can be directly typed out after the element. It's implied that the leading number of a molecular formula will be its coefficient. Spaces between the coefficient, elements, and subscripts are optional, and will be omitted during the compilation of the code. Here are a couple examples of simple and complex chemical formulas:

Chemical Formula	L++
H_2O	`H2O`
$2C_6H_{12}O_6$	`2C6H12O6`
$[Cu(NH_3)_4]^{2+}$	`(Cu(NH3)4)^2+`

```
reaction CitrateCycle(acetyl_CoA + 3 NAD + FAD + ADP + Malate --> CoA_SH + 3 NADH +
FADH2 + ATP + Malate)
```

Higher-level simulations often require more complex substances that have long, precise formulas. It would be tedious to manually type these formulas out everytime they were used, so L++ comes with an imported database of millions of common chemical formulas along with their chemical synonyms. As long as the synonym is supported, typing out the synonym directly in replacement of its chemical formula is also supported. If the program fails to compile with an error at a chemical synonym, it is most likely not yet supported by the language yet. No additional libraries are required to be imported in order to use this functionality. Here's an example of ethanol combustion with chemical synonyms:

```
reaction combust(C2H5OH + 2O2 --> 2CO2 + 3H2O);
reaction combust(ethanol + 2oxygen --> 2 carbon dioxide + 3 water);
```

L++ determines the states of matter of any of the reactants and products based on the context of the simulation that most matches the real-world experiment. Therefore, states of matter don't need to be explicitly declared in the reaction.

To create more scalable simulations that are capable of handling molecular interactions at a protein, or even organism level in the future, declaring millions of chemical substances manually for each biochemical interaction is beyond cumbersome. Though further discussed later in section 4.5, it is worth prefacing that molecules and chemical substances are able to be encapsulated within more advanced data structures such as reactions, proteins, and pathways. The object-oriented approach enables significant scalability of the language in a variety of contexts.

Nucleotides

In genome-related experiments, L++ supports the expression of gene sequences in the form of DNA or RNA. On a basic level, the language supports the following syntax:

A - Adenine
G - Guanine
C - Cytosine
T - Thymine
U - Uracil
N - Any Nucleotide

In various biological scenarios nucleotide sequences are unknown at the beginning of the experiment; they are later determined by the context of the molecular interaction. As such, `N` can be used for positions in a specific gene sequence that can represent any nucleotide, which is later determined and replaced by one of `A, G, C, T, U,` during the simulation.

```
reaction RNAElongation(RNAP~Rna_{n} + c(NTP) + RNAP.ssDNA.N{n} --> RNAP~Rna_{n+1},
r=60.0);
```

2.2 Control Flow

As in any modern programming language, control flow is used to express and introduce rules within a program, such that certain pieces of code can be run for a number of times or only run under specific conditions.

Conditional Statements

If and else statements are used to control whether a piece of program should be executed under a certain condition. In many experimental contexts, molecular interactions should only occur if a specific concentration is reached, for example. The condition for an if statement must

be wrapped within parentheses following the `if` keyword, and code used within `if` or `else` statements must be wrapped within curly brackets following the condition. `if` statements can stand alone, while `else` statements must be paired following an `if` statement.

```
specification
{
   Ang = Membrane.phi;
   Pos = Membrane.position;
   cdist = 2.5e-5; // velocity from BergBrown_1972
   crot = rand(0, 2 * pi); // constant for rotation

   if (bRun)
   {
       Pos' = cdist;
       Ang' = 0;
   }
   else
   {
       Pos' = cdist / 3;
       Ang' = crot;
   }
}
```

Looping Statements

In many experimental cases, repetitive tasks are necessary, such as preparing a series of beakers with a particular solution. L++ supports familiar looping structures such as the `for` loop and the `while` loop for code efficiency, similar to how they work in other languages.

`for` Loop

A `for` loop should be used when a piece of code is designed to run a fixed number of times. An example may be wanting to run an Electrophoresis experiment 15 times, plotting all the results on a graph. The syntax for the `for` loop is similar to their counterparts in languages like C++ and Java, with the following structure:

```
for ([declaration]; [condition]; [increment])
{
    [code block]
}
```

1) **Declaration.** A variable assignment for a number.
2) **Condition.** Uses the previously declared variable and is often used with an inequality to state that while the condition holds true, the code within the for loop will execute.
3) **Increment.** The update of the declaration variable by a certain number, which can either take a positive or negative value.The declaration can “count down” by decrementing or “count up” by incrementing.

Here’s an example of how `for` loops can be used in L++ code. Note that the declaration, condition, and increment are all wrapped within parentheses, and each one is separated with a semicolon. Code to run in a loop must be wrapped within curly braces.

```
ecoli e;
for (i = 0; i < 3; i = i + 1)
{
   e(rand(10), rand(10), 0) = 1;
}
```

This generates *E. coli* at a random location three times. The random x, y location is drawn from a uniform distribution from 0 to 10, inclusive. The third parameter represents the z coordinate, which is 0 as the *E. coli* are generated on a 2D plane.

`while` Loop

The `while` loop should be used when a piece of code is designed to run until a certain condition is met, rather than a fixed number of times. This may mean that the piece of code can never run, run once, run infinitely, or anything in between. Here's the structure:

```
[declaration]
while ([condition])
{
      [code block]
      [increment]
}
```

If the variable’s value still satisfies the condition of the `while` loop, the code block will run. Every time the code block is run, it is the programmer’s responsibility to increment or decrement the value of the variable at some point inside the code block. Although it’s most common to perform this increment at the end of the code block, it’s up to the programmer to decide how and where the variable’s value is updated.

A common mistake for many programmers is forgetting to increment or decrement the variable, resulting in running the code block infinitely (infinite loop). It is further important to note that the declaration for the variable used in the condition must occur before the while loop itself. Otherwise, the variable used in the condition is unable to know what value it is referencing. Here's an example:

```
ecoli e;
int i = 0;
while (i < 3)
{
   e(rand(10), rand(10), 0) = 1;
   i++;        // ++i works as well
}
```

These looping structures will evidently become more useful upon further development, such as looping over an array of experimental structures in simulations involving numerous entities / trials.

2.3 Basic Operators and Keywords

Before we get into each set of operators, let's go over some syntax typing conventions. Typically, code involving any of the arithmetic operators, the logical operators (besides negation), and the assignment operator should have a space between the operator and the operands. Here's an example:

```
float x = (5 + (3 * 1)) % 2
bool x_is_even = (x % 2) == 0;
```

Though these statements may compile and run successfully without the spaces, many L++ identifiers are often molecular substances that contain similar symbols in their names. These counterexamples may obstruct code readability:

`Ca+2_ion`
`2-Butanol`

Arithmetic Operators

L++ mathematical operators follow the same set of rules as any other programming language, in terms of associativity, commutative, distributivity, and order of operations. Order can be further enforced through the use of parentheses. The following operators are available:

Addition	**+**
Subtraction	**-**
Multiplication	*****
Division	**/**
Modulo	**%**
Exponentiation	****** **e**

L++ includes new syntax for exponentiation. Similar to other languages, `**` is still supported for statements like `float x = 10**-5`, representing 10^{-5}. However, `e` can also be used to represent scientific notation with the format `<base>e<power>`. 10^{-5} can also be expressed as `float x = 10e-5`.

Increment Operators

Syntactic sugar to increment or decrement a variable, either by 1 or by `n`, is supported.

Increments or decrements performed to a variable by 1 can be performed via `++` or `--`. Addition and subtraction are supported, meaning that you can either add 1 to a variable or subtract 1 from a variable with both of these operators, respectively. The order of placing these operators before or after a variable matters.

Preceding a variable with `++` or `--` increments/decrements the variable's value first, then evaluates the variable. Following a variable with `++` or `--` evaluates the variable first, then increments/decrements the variable value. The difference in the order of these operations matters most when used in evaluating conditional statements and expressions. Here's an example.

```
// Case 1:
   x = 1;
   y = x++;        // x is 2, y is 1

// Case 2:
   x = 1;
   y = ++x;        // x is 2, y is 2
```

Increments or decrements performed to a variable by `n` wil can be performed by `+=, -=, *=, /= , or %=`. In any case, the operation is performed on the variable, and the result of the operation will be assigned to the variable. Here's the format:

```
x [o]= n;
```

`[o] - Operator.` Required. Can be one of `+`, `-`, `*`, `/`, or `%`.

Logical Operators

Logical operations result in booleans - either true or false. They are commonly used in program control flow, such as if-else statements to determine whether a piece of code should run based on a given condition.

Negation
Inverts the boolean.
`A = true;`
`!A = false;`
	!

Logical And
True if and only if all operands are true.
`A = true;`
`B = true;`
`C = false;`
`A && B = true;`
`A && B && C = false;`
`A && !C = true;`
	&&

Logical Or
True if any one of the operands are true.
`A = true;`
`B = false;`
`A || B = true;`
`!A || B = false;`
	||

Inequality Operators

These are operators used to evaluate two numerical expressions, returning a single boolean value indicating whether the statement is true or not. The following inequalities are supported:

Greater than
Returns true if the left hand side is greater than the right hand side. False otherwise.
 	
`x = 3 > 2;  // True`
`x is true`
	>

Greater than or equal to
Returns true if the left hand side is greater than or equal to the right hand side. False otherwise
 	
`x = 2 >= 2; // True`
`y = 3 >= 2; // True`
`// x and y are true`
	>=

Less than
Returns true if the left hand side is less than the right hand side. False otherwise.
 	
`x = 0 < 1;`
`// x is true`
	<

Less than or equal to
Returns true if the left hand side is less than or equal to the right hand episode. False otherwise.
 	
`x = 1 <= 2;`
`y = 2 <= 2;`
`// x and y are true`
	<=

Equal
Returns true if the left hand side is equal to the right hand side. False otherwise.
Does not assign a value to a variable.
 	
`x = 5 + 3 == 4 * 2;`
`// x is True`
	==

Not equals Returns true if the left hand side is not equal to the right hand side. False otherwise.	**!=**

`x = 1 != 2;  // x is True`	

These inequalities currently only support numerical comparisons. Operator overloading for customized comparison of experimental or self-created structures may be supported in future development.

Assignment Operator

Assignments are expressed through `=`, which is typically used to assign a value of any data type to a variable that can be manipulated later, such as in a function. The assignment operator has a different meaning for each type of data type that performs the assignment. Here's a basic list of each data type and the meaning of the assignment, along with several examples. Note that these examples are for structural demonstration purposes only; they don't have a particular meaning. Sections corresponding to each data type and structure will contain further examples and explanations.

1) **Basic Data Type**
   Assignment to a value of the corresponding data type. Integers - int and float - should be assigned numerical values like 5.0 and 3, along with an optional unit.

```
float x = 5.0;
```

2) **Parameter**
   Assignment to a value accepted by the corresponding parameter.

```
kcat = 1.4;
```

3) **Molecular Substance**
   Assignment to a numerical value to indicate a concentration.

```
h2o = 5nM;
glucose = 12uM;
```

4) **Experimental Structure**
   Assignment to another experimental structure as a reference. The example below assumes p1 and p2 are the same type.

```
protein p1 = p2;
reaction r1 = p1; // does NOT work
```

5) **Reference Variable**
   Assignment to alphanumeric strings depending on the parameter. Numeric values can be passed in for specific numeric-based parameters, such as page and equation.

```
reference Ingalls_2013
{
   title = “Mathematical Modelling in Systems Biology”;
   page = 151;
   figure = 6.2;
   equation = 6.0;
}
```

Reassignments of a variable that was initially, explicitly typed can only be possible if the data type (before and after) matches. If a variable’s type was not stated upon declaration, then its type will be inferred through the biological context of the simulation.

Positional Operator

When describing reactions involving macromolecules, precise referencing to specific locations of these structures may be needed. The curly operator enables positional specification of the building block along polymers, such as primary sequences of DNA, RNA and protein, and an individual protein in multimeric protein complexes. The `{}` operator is mainly used with DNA and RNA sequences in this format:

`<genetic-sequence-structure>[.N]{<position1> [: <position2>]}`

`[.N]` - Primary sequence. Optional. Access nucleotide at the nth position. Can only be used in specific contexts like RNAElongation. Can access a stretch of nucleotides if a range is used instead of a number for the index.

`[: <position2>]` - Index. Optional. Specify inclusive range from position 1 to position 2.

Here are a couple examples of how this operator can be used in the context of RNA translocation during transcription:

```
reaction Translocation(RNAPCore~ssDNA{n} --> RNAPCore~ssDNA{n+1}, r=10);
// binding site on ssDNA is changing in this reaction, resulting a translocation
```

In the above example, the RNAPCore binding to the nucleotide at the nth position of ssDNA has been replaced to the nucleotide at the n+1th position along ssDNA, resulting a translocation of RNAPCore on its bound ssDNA.

Length Operator

The `_{}` operator is used to access the length information, that is the quantity of building blocks, of polymers, such as DNA, RNA and protein, and multimeric protein complexes. The distinguishing feature of the length operator _{} from the positional operator is that the length operator does not concern the identity of the building blocks whatsoever. Here's the format:

`<polymer-structure>_{<length>}`

`<length>` - Index. Specifies …

Polymer elongation reaction is a type of reaction that can be efficiently described with the length operator. Here is an example of RNA polymerase extending the nascent RNA using ssDNA as a template:

```
complex RNAP // self = RNAPCore~RNA~ssDNA
{
      reaction Elongation(RNAP.RNA_{n} + c(NTP) + self.ssDNA.N{n} --> self.RNA_{n+1}
+ PPi + self.ssDNA.N{n}, r=60);
}
```

Membership Operator

The `.` operator is important in accessing specific data stored inside these abstractions. It functions the same as the dot operator in languages like Java, C, and C++. Let's walk through an example of how it can be used:

```
protein p
{
      reaction r1(...);
      reaction r2(...);
}
```

A `Protein` is instantiated in a code block, and the identifier for it is `p`. The identifier p is used to refer to this object in other places in the rest of the code. The protein takes in two reactions that describe `p`, which are the r1 and r2. These reactions may be accessed using the membership operator, e.g. `p.r1` and `p.r2`.

This operator is integral in a high-level language to describe simulations that can get large quickly. The membership operator can also be used to refer to specific structures located in imported files as well. Here's an example:

```
import Divisome; // .lpp file name, contains complex ZRing structure
ZRing = Divisome.ZRing; // accessing complex declared in Divisome.lpp
FtsZ_Recruitment = ZRing.FtsZ_MembraneRecruitment; // accessing reaction declared in
ZRing complex declaration
membrane Membrane;
Angle = Membrane.phi; // accessing intrinsic angle property of membrane variable
```

Indexing Operator

Square brackets, commonly used in array indexing in other languages, have a different connotation in L++. Though arrays will be developed in the near future, currently, indexing is primarily used in declaring the concentration of a substance at a given timestep. Square brackets can be used directly after an identifier of a substance, with the index being the absolute timestep (seconds by default), and the assignment being the concentration to set the substance at. Here's the format:

`<identifier-name>[t] = [c][u]`

`[t] - Absolute Time.` Required. Integer or floating point value.
`[c] - Concentration.` Required. Integer or floating point value.
`[u] - Unit.` Optional. Any SI prefix + unit is supported.

How is this useful? Initial concentrations are an example of a situation where concentrations must be declared in any given experimental context. Another example may be keeping certain substances at a fixed concentration throughout an experiment. The syntax for timesteps are similar to slicing in Python: an optional colon may be used to specify specific timestep ranges.

```
Molecule[0] = 5nM;  // initial concentration at time 0
Molecule[:] = 5nM;  // fixed concentration
Molecule[1:] = 5nM; // concentration from timestep 1 & onwards
Molecule [:3] = 5nM; // concentration up to 3rd last timestep
```

Here's a comprehensive example representing feedback inhibition of Hexokinase I.

```
protein HK1(Glucose --> G6P, kcat=30, KM=KM_list[i],
                              inhibitor=G6P, mode=allosteric, Ki=1.4, n=1.6);
protein E1(G6P --> X, k=5);

HK1[0] = 5; // count at time 0 (initial count)
Glucose[:] = 5; // fixed concentration
```

```
E1[0] = 1;
X[:] = 0; // optional
```

The Binding Operator

Molecular interactions often lead to non-transient binding between the molecules. L++ uses '~' (aka. a tilde) as the symbol to represent that two biochemical entities are bound together. These entities are commonly proteins, domains, and the other molecular structures used to represent biochemical abstractions. Here's an example:

```
reaction SigmaFactorBinding(RNAPCore + SigmaFactor --> RNAPHolo, Kd=1e-7);
```

As previously mentioned, other L++ keywords, like `self`, can be used in conjunction with the binding operator. A common example, thus, is such that a protein, reaction, or any other structure referred to by `self`, can be bound to another declared biological experimental structure to express a merged state between yourself and another entity. Here's an example:

```
reaction dsDNAUnwinding(self~dsDNA~RNA{n-10:n+10} --> self~ssDNA{n-10:n+10} +
ssDNA{n-10:n+10}, r=R);
```

Here's also an example of the multi-binding property of the operator:

```
reaction 70S_Dissociation(30S~50S~mRNA~tRNA~RRF --> 30S + 50S + mRNA + tRNA + RRF,
Kd=1e-9);
```

The Arrow Operator

Many experimental structures involve chemical reactions that describe the relationship between reactants and products. The `-->` operator is used to not only separate the reactants from the products, but indicate the formation of products from reactants (in a forward reaction), and vice versa (in a reversible reaction). Every type of reaction uses the `-->` operator; only the constants passed into a reaction are necessary to differentiate chemical reaction types.

`reaction RxnName(reactants --> products, constants);`

`self` Keyword

When using these experimental structures, another common, helpful operator is the `self` keyword, which functions as a way to express a structure's own identity. Other programming languages have their equivalent syntaxes: for C++ and Java, 'this' is used for an object to refer to itself, and for Python, `self` does the same.

`self` can be used in a variety of contexts and along with other operators, such as the membership operator as well. In the same way that it is commonly used in other languages,

`self` may be followed by a `.`, then followed by an identifier that belongs to that entity. The identifier can be a variable (which would be called a “member variable”, or even a function that belongs to that entity (which would be called a “method”). L++ supports similar functionality; however, as many of these structures don’t contain methods of their own, `self.<identifier-name>` is commonly used to refer to a specific variable declared within that structure. Here are a couple examples:

```
complex RNAP
{
   // In this context, self = RNAPCore~ssDNA;
   reaction Translocation(self.ssDNA{n} --> self.ssDNA{n+1}, r=10);
}
```

Note: we could technically use `.N{n}` to access the nucleotide at the position *n*, but the nucleotide information wouldn’t mean much in this translocation context.

Here, `self.ssDNA` assumes `self` is a complex that contains a single-stranded DNA component, which can either be explicitly or implicitly stated above. `self.ssDNA{n}` refers to position *n* of the `ssDNA`, which changes into `self.ssDNA{n+1}`, position *n+1* of the `ssDNA`. In a biological context, the reaction describes the 1 unit of procession of the binding site along the `ssDNA`.

`self` can also be used independently. This may seem a little odd, but it makes perfect sense in a variety of experimental contexts. For example, a protein complex may be described through a series of reactions. In these reactions, the complex may react with other molecules, modifying its state. Therefore, `self` can be inside of a reaction statement enclosed within a `complex` block to refer to that complex (itself). Here’s an example.

```
complex RhoComplex // Rho complex
{
   reaction RNABinding(self + rut_repeat -> self~rut_repeat, Kd=1e-7);
   ...
}
```

Another example of its flexibility is combining the `self` keyword with the binding operator `~`. We’ll cover that more in detail in section ____.

2.4 Advanced Molecular Structures

As L++ emphasizes the importance of encapsulation and abstraction, the language aims to support higher-level molecular structures that allow computational scientists to experiment with

larger, more complex structures easily without having the need to declare thousands of reactions on a microscopic level for every larger molecular entity, such as a protein complex. Here are a couple of fundamental, more advanced structures that encapsulate some of the basic structures that have been covered.

Reaction

A chemical reaction is a key component of many chemical pathways, and are the foundations of the mechanisms in which chemical substances and organisms interact with one another. A reaction is a statement on its own, and describes the stoichiometric relationship between different biochemical entities. Here's the basic syntax structure for declaring a reaction.

```
reaction <identifier-name>[([e], [op])][;]
{
      [e],
      [op],
}
```

`[e]` - Chemical Equation. Required. reactants --> products

`[op]` - Other Parameters. Optional. Includes constants, reaction rate, and more.

Reactions typically - though not enforced - contain the chemical equation as the first argumentof the declaration. They also often have at least one reaction rate constant, which can be any of the following: `k, krev, kcat, KM, Ki, Ka, Kd and nH.`The type of reactions can be specified by using appropriate set of constants. For instance, some parameters may be used alone (`k` or `Kd`), or conjugated (`k` and `krev`) or always meant to be coupled (`Ki` or `Ka` constants with Hill's coefficient `nH`; `kcat` and `KM`). Another parameter is the rate of reaction **`(r)`**, which describes the rate of reaction. Unless a specific unit is passed in, the default unit is 'per second'. Explanations for each parameter and their corresponding units can be found in sections 4.7 and 4.8.

Here are a couple of important notes on the reactions written in L++:

1. **Coefficients**
   Coefficients for both reactants and products can be expressed through integers and floating point numbers for precise reaction stoichiometry or alphabets for generalization. They should precede the molecular substance it is describing, which can be represented as a chemical formula or a L++-supported chemical synonym.

2. **Directionality.** All reaction forms will contain the reaction arrow pointing to the right, without requiring to indicate reversibility or inhibitory action. The type and directionality of the reaction solely follows the parameters and the corresponding reaction model. For

example, passing in the `krev` constant in `rate constants` directly indicates a reversible reaction.

```
reaction RxnName(reactants --> products, k and krev);
                    // Forwards reaction (k)
                  // Reversible reaction (krev)
```

3. **Inhibition and Activation.** The regulatory effect of the reaction also strictly follows the type of parameters passed in, without having to use an inhibitory arrow. For example, passing in the `Ki` constant in the regulatory reaction implies an inhibitory reaction of the regulator molecule on the target reaction.

```
reaction RxnName(regulator --> reaction identifier, Ki and nH);

                    // Inhibitory reaction
```

4. **Context Dependency.** For reactions enclosed within the protein or complex parentheses or code-blocks, the protein or complex (`self`) must be explicitly involved in the reaction or will act as an enzyme that facilitates the given reaction. In such enzymatic reactions, the enzyme concentration will be reflected onto the reaction rate.

L++ enables high-level abstractions of reactions without having to solely rely on biochemical algorithms, namely predefined reaction models, which are often designed to describe relatively low-level biochemical reactions involving law of mass action and kinetic parameters for enzymatic functions and molecular interaction. To do so, there are several customization options for each reaction. Each option can be expressed with a keyword and a matching code block placed underneath the reaction. As many customization blocks can be used for a given reaction. Here's the general format:

```
reaction <identifier-name>(...);
[k]
{
...
}
```

`[k]` - Customization Keyword. Optional. Any keywords from the following are allowed.

1. **`specification`.** Defines how the reaction statement should be executed without using a preset biochemical reaction model. Specifications are mathematical implementations of intended pathway designs, which are particularly useful when there is insufficient biochemical implementation available. They are numerical rules that can be used to determine the concentration of a chemical substance, for example, at any point of time in

any given reaction. Relationships between the quantities of substances are created through assignments to mathematical expressions referencing other molecular substance concentrations.

```
reaction Glycolysis(G6P + 2 ADP + 2 NAD --> 2 pyruvate + 2 NADH + 2 ATP)
specification
{
   float cg = 0.019M;
   ATP'  = (ADP / ATP) * cg;
   ADP'  = - ATP';
   NADH' = ATP';
   NAD' = - NADH';
   G6P'  = ATP' / 2;
   pyruvate' = G6P' * 2;
}
```

ADP/ATP represents the ADP to ATP ratio at a given time, which is used to calculate the amount of ATP generated at every time step. In future developments, the rate of reaction for only one of the participating molecules can be described, and the rest will be automatically adjusted according to the stoichiometry of the reaction.

Notice the apostrophes `'` following the naming of each reactant/product within the code block. This is used to describe the change of quantity in each simulation round, essentially a derivative where $ATP' = \frac{dATP}{dt}$. Furthermore, the order of each equation in a `specification{}` matters, as each following equation may depend on the concentration of a previously described relationship. Each equation can be separated via a semicolon, as they are each independently standing statements.

2. **`k_function / krev_function`.** Allows customization of each forward and reverse reaction rate calculations. The return values are used to update the molecular quantities of each reaction substrate.

```
reaction FtsZ_Recruitment(FtsA~(FtsZ_{n}) + FtsZ{in} --> FtsA~(FtsZ_{n+1})); // (n can
be 0, as well)
k_function
{
   return FtsA / max(1, MinC{in});
}
krev_function
```

```
{
   return 0.5 + MinC;
}
```

In the example above, the FtsZ_Recruitment reaction uses two different functions to calculate the forward and reverse reaction rates.

Protein

Proteins are a specific class of macromolecules made of amino acid residues. They include enzymes that can catalyze certain reactions and structural/functional components of a cell. As proteins are often composed by the series of reactions they facilitate or are involved in, `protein` code blocks mainly contain relevant reaction declarations. If they simply encompass a single reaction, they can be declared similarly to how reactions are declared as well. Here are the basic formats:

```
protein <identifier-name>[([r])][;]
[{
      [r]
      …
}]
```

[r] - Reaction. Optional. A protein involved in multiple reactions should be placed in a code-block rather than a single statement.

Here are a couple of examples of accepted protein declaration statements.

```
protein CheB;
protein CheAm(CheB --> CheBP, k=0.05e9, krev=0.005);
protein CheR
{
    reaction CheA_Methylation(CheA --> CheAm, kcat=1, KM=1e-10nM);
    reaction CheAL_Methylation(CheAL --> CheAmL, kcat=1, KM=1e-10nM);
}
```

Complex

Proteins often form together to create stable protein complexes via non-covalent molecular interaction. This provides a modular regulation of molecular processes. The ‘complex’ syntax is largely interchangeable with ‘protein’. However, this data structure allows for shortcut complexation reactions for forming itself by specifying the subunits of the complex along with the kinetic parameter, Kd.

```
protein RpoB, RpoC, RpoA, RpoZ;
complex RNAPCore(RpoB + RpoC + RpoA + RpoZ, Kd=1e-10);
complex RNAPHolo
{
    reaction SigmaFactorBinding(RNAPCore + SigmaFactor, Kd=1e-7);
    // '--> self' is assumed in the reaction argument
    reaction PromotorBinding(self + dsDNA.promoter --> self~dsDNA.promoter);
}
```

Complexes inherit information from their different protein subunits to access their domains (for proteins) and motifs (for DNA and RNA). If no reaction is passed in, the reaction arrow (`--> self`) is made as default.

Further development of L++ will enable additional, exclusive features that differentiate complexes further from proteins.

Nucleic Acids

Our goal to simulate cells and even entire organisms in the future is - and can be - largely expressed through genes. Though DNA is one of the major structures, L++ supports a variety of other genetic-related structures that help describe the functional behavior of these abstractions.

DNA

Deoxyribonucleic acid is a polymer - a naturally-occurring substance composed of macromolecules. In this case, the macromolecule refers to a nucleotide, structure made of a nucleobase (Adenine (A), Guanine (G), Cytosine (C), Thymine (T)), a five-carbon sugar, and a phosphate group. Sequences of nucleotides are formed to create a singular strand, sometimes referred to as a template strand. DNA's double-helix structure consists of the template strand, and a complementary strand containing opposing nucleotides for each nucleotide in the template strand. Here's a look at the structure of a sequence.

Template Strand: **CTGGCACNNN**
Complementary Strand: **GACCGTGNNN**

As seen in the example above, each strand simply can be expressed through the nucleobase that the nucleotide contains. Adenine and Thymine complement one another, while Cytosine and Guanine complement one another.

However, it's important to note that many biological reactions can often change the state of a sequence. Sometimes, and quite commonly, certain nucleotides in sequence cannot even be

determined until after a specific reaction occurs. To support this idea, L++ uses the letter `N` to express the idea of an unknown nucleotide. Here are two examples:

```
DNA Ori;
reaction OriBinding(DnaA + Ori --> DnaA~Ori, Kd=1e-9);
```

```
reaction OligoSynthesis(DNAP~ssDNA.N{n-10:n} + c(dNTP) --> DNAP~dsDNA, r=1000);
```

Note: In the second example, `N` in `ssDNA.N{n-10:n)` refers to the stretch of 10 nucleotides.

RNA

Though genetic information is encoded within DNA, another nucleic acid is frequently found in biological contexts for other purposes. Ribonucleic acid is again a polymer and basically shares the same structure as DNA; however, Uracil (U) replaces Thymine (T) as one of its possible nucleobases. Furthermore, RNA is single-stranded. It carries genetic information to perform its main purposes, which involve facilitating the coding of proteins via translation, for example. The three main types of RNA are messenger (mRNA), transfer (tRNA), and ribosomal (rRNA). Here's an example:

```
RNA tRNA;
complex AA_tRNA;
protein AA_tRNASynthetase(tRNA + AA --> AA_tRNA, kcat=1, KM=1e-9);
```

Similar to how amino acid sequences are not enforced to be declared for proteins to describe their functions, L++ doesn't require DNA or RNA to be assigned nucleotide sequences as long as they can be assigned functions. Assignments to specific sequences and relevant features will come with further development in the near future.

Pathway

Exactly synonymous with the biological term, a 'pathway' is an abstraction describing a series of biochemical interactions that lead to some product or effect within a particular context, most commonly a cell. Since these interactions are then most commonly described through a sequence of reactions, that's exactly what the syntax of a L++ pathway looks like. Here's an example:

```
pathway Transcription
{
   pathway Initiation
   {
       SigmaFactorBinding = RNAPHolo.SigmaFactorBinding;
       PromotorBinding = RNAPHolo.PromotorBinding;
       SigmaFactorRelease = RNAP.SigmaFactorRelease;
```

```
    }
    pathway Elongation
    {
        dsDNAUnwinding = RNAP.dsDNAUnwinding;
        DNATranslocation = RNAP.Elongation.DNATranslocation;
        RNAElongation = RNAP.Elongation.RNAElongation;
    }
    pathway Termination
    {
        IntrinsicTermination = RNAP.IntrinsicTermination;
    }
    SigmaFactor = 1500, RNAPCore = 3000, NTP[:] = 9mM;
}
```

Like previously stated, the syntax and rules for chemical reactions in general can be applied here as well. For example, one can assign and reassign chemical reactions declared within pathways to previously declared chemical reactions elsewhere in our code.

As seen above, pathways aren't simply limited to containing reactions, but may also include proteins, structural and even other pathways. This is why pathways can contain a majority of other L++ experimental structures. Assignments for other structures, from reactions to domains to other pathways, can be made within pathways as well.

Structural Components: Cytosol and Membrane

Living matters are physically and functionally compartmentalized. The simplest organization is to have an aqueous compartment, a.k.a cytosol, encapsulated by lipid bilayer, a.k.a. membrane, physically separated from the environment. L++ currently supports these two key structural components with syntax `cytosol` and `membrane`. Generally speaking, `cytosol` harbors large scale reactions driven by high abundance of molecules, such as metabolites. An example of a metabolic pathway occurring in the `cytosol` is glycolysis and protein biosynthesis. On the other hand, `membrane` is often used to elevate local concentration of low abundance molecules to facilitate their intermolecular interactions. These structural code blocks can contain any combination of sequence of reactions, pathways, or other molecular abstractions. Here's an example:

```
import GlucoseTransport;
import Glycolysis;
membrane CellMembrane
{
    GlucoseTransporter = GlucoseTransport.GlucoseTransport;
```

```
    cytosol Cytosol
    {
        Glycolysis = Glycolysis.Glycolysis;
        G6P = 8.8mM, ATP = 9.6mM, ADP = 0.56mM, NADH = 8.3mM, NAD = 2.6mM;
    }
}
```

By now, it's quite clear that there doesn't exist a clear top-down hierarchy of these molecular structures. With the complexity that comes in a biological cell, molecular structures can contain, and simultaneously be contained within, other molecular structures.

Notice how a membrane code block also contains molecular structures that can be found within a cell (ie. the cytosol). Just like how the cellular membrane wraps around cytosol to form the shape of a cell, the membrane in the code above acts as an outer scope, encapsulating the cytosol existing within the membrane.

The structural syntax substantially expand in the next few updates to accommodate high resolution spatial simulations.

2.5 Imports

As in any other programming language, common libraries can be imported in order to use specific data types and methods corresponding to those types that have already been implemented. Other L++ files can also be imported to access experimental structures and code located in other source files. Imports should typically be declared at the beginning of a program, so that the rest of the program can use any method and or data structure part of the library that was implemented. Here's the syntax for importing a library/L++ file:

```
import [Import];
```

`[Import]` - **Import.** Required. Can either be a L++ library or filename. Filename does not need to include .lpp; just the name of the file is sufficient.

Users do not need to worry about cyclic dependencies between files that contain the same imports, as the language handles this itself. L++ supports a growing number of libraries containing built-in functions. Further references can be found in sections …

2.6 Parameters

As L++ supports a variety of experimental data structures, it's imperative to note that different declarations take on a unique set of parameters. For example, the parameter 'temp' - for temperature - wouldn't make sense for a pathway declaration. Each section in this manual

describes the list of parameters it can take on, and this section aims to enumerate a complete list of possible parameters in L++ and their respective meanings.

Constants

Specific, non-changing numerical values may be used to describe the state of various molecular structures. For example, dissociation constants are almost always used in describing the behavior of a reaction. Here are a couple of constants and their syntax counterparts.

k	Forward reaction rate constant. Describes the proportional relationship between the concentrations of reactants and the rate of forward reaction.
krev	Reverse reaction rate constant. Describes the proportional relationship between changes of the concentrations of reactants and the rate of reverse reaction.
kcat	Catalytic conversion constant of substrate into product. Used together with **KM** in Michaelis Menten kinetics.
KM	Michaelis constant. Describes the amount of substrate needed for the enzyme to obtain ½ of the maximum rate of reaction. Used together with **kcat** in Michaelis Menten kinetics.
Ki	Inhibitory constant. Describes the concentration of an inhibitor required in order to half the maximal reaction rate by ½.
Ka	Activation constant. Describes the concentration of an inhibitor required in order to half the maximal reaction rate by ½.
nH	Hill's coefficient. Describes the degree of cooperativity in a binding process. Used together with **Ki** or **Ka**.
Kd	Dissociation constant. It is a type of equilibrium constant that describes the propensity of a molecular complex to separate reversibly into smaller components.
r	Rate of reaction. Directly used for the rate of reaction, instead of using the law of mass action.

Reference

Descriptions of biological pathways and experimental contexts can often be attributed to alternate authors and scientists. Instead of commenting, L++ directly supports a `reference{}` code-block type that encourages programmers to credit other sources when reference their work. Here's an example.

```
reference Ingalls_2013
```

```
{
   title = "Mathematical Modelling in Systems Biology";
   page = 49;
   equation = 3.2;     // Network
   page += 50;
   figure = 3.3;       // Plot
   equation += 3.3;    // Equation
}
```

The code-block takes in the following, expanding list of parameters:

1) **`Title`** - Accepts a string that names the title of the reference.

`title = “[s]”`

`[s]` - Source Name. Required.

2) **`DOI`** - Digital Object Identifier. A unique alphanumeric string that provides a persistent link to the original source on the internet.

`doi = “[d]10.[o]/[p]”`

`[d]` - URL Prefix. Optional. Protocol and resource names, that is `https://doi.org/`.

`[o]` - Organization Number. Required. Four-Digit Identifier assigned to organizations.

`[p]` - Publisher String. Required. Alphanumeric string assigned by publisher.

3) **`Date`** - Accepts an alphanumeric string given in any of the following formats:

`date = “[Month] [Date], [Year]”`

`date = “[MMDDYY]”`

`[Month]` - Month. Required. Alphabetic string for month.

`[Day]` - Day. Required. One or Two-Digit Integer value.

`[Year]` - Year. Required. Four-Digit Integer.

`[MMDDYY]` - 6 digit integer representation for MM (month), DD (day), YY (last two digits of year). Required.

4) **`Volume`** - Accepts a string corresponding to the volume of the reference. Does not require a specific format.

`vol = “[v]”`

`[v]` - Volume. Required. Alphanumeric String.

5) **`Journal`** - Accepts an alphanumeric string naming the reference's journal name.

`journal = "[k]"`

`[j]` - Journal Name. Required. Alphanumeric String.

6) **`Page`** - Accepts an integer numeric value corresponding to the page number in the reference. Can be further assigned an incremental assignment to append value to previously declared page #. Here are a couple formats:

`page = [p];`
`page += [p];`

`[p]` - Page Number. Alphanumeric String.

7) **`Figure` -** Accepts an integer or floating point numeric value corresponding to the figure number in the reference. Can include an optional delimiter to indicate a subfigure. Can be further assigned an incremental assignment to append value to previous declared figure #. Here are a couple formats:

`figure = [f].[s];`
`figure += [f].[s];`

`[f]` - Figure Number. Required. Integer or floating point value.
[s] - Subfigure. Optional. Alphanumeric String.

8) **`Equation` -** Accepts an integer or floating point numeric value corresponding to the equation number in the reference. Can be further assigned an incremental assignment to append value to previously declared equation #.

`equation = [e];`
`equation += [e];`

`[e]` - Equation Number. Required. Alphanumeric String.

Any number of parameters may be used in any given reference block. Parameters are optional and can be repeated. Each assignment within the code-block is a statement, so they must be separated by semicolons.

Reactions

1) **Equation**. Accepts a chemical equation composed of reactants, products, and the relationship between them via an arrow operator. The argument `eq =` can be explicitly written, or the equation can directly and implicitly be passed in. Typically the first parameter in a reaction declaration statement.

```
eq = <reactants> --> <products>
```

The type of reaction is determined by the type of reaction parameters provided with the equation.

2) **Reaction Parameters.** Accepts any one of the listed reaction constants or functions located in section ___. Depending on the reaction type, multiple reaction constants may be passed in.

3) **Rate of reaction `(r).`** Describes the rate of reaction through a numerical value. Unless a specific unit follows, the default unit is per second.

2.7 Prefixes and Units

In many experimental contexts, units must be used to describe specific characteristics such as concentration, volume, and more. L++ implements a two-part format of expressing a prefix and a unit.

Prefixes

Prefixes are the optional, first half of a unit. Our prefix system matches that of traditional SI Unit prefixes. The following prefixes are supported. Note that the capitalization of the prefix matters.

Yotta (Y) - 10^24
Zetta (Z) - 10^21
Exa (E) - 10^18
Peta (P) - 10^15
Tera (T) - 10^12
Giga (G) - 10^9
Mega (M) - 10^6
Kilo (k) - 10^3
Hector (h) - 10^2
Deka (da) - 10^1

Deci (d) - 10^-1
Centi (c) - 10^-2
Milli (m) - 10^-3
Micro (u) - 10^-6
Nano (n) - 10^-9
Pico (p) - 10^-12
Femto (f) - 10^-15
Atto (a) - 10^-18
Zepto (z) - 10^-21
Yocto (y) - 10^-24

Units

Prefixes are the required, second half of a unit. The following units are supported. Note that the capitalization of the units matters. Identifiers and parameters named after the units below will result in a syntax compiler error.

Volume	Electricity

Liter - **L**

Time

Second - **sec**

Minute - **min**

Hours - **hr**

Mass

Gram - **g**

Temperature

Celsius - **C**

Fahrenheit - **F**

Kelvin - **K**

Volt - **V**

Ampere - **A**

Concentration

Mol - **mol**

Molarity - **M**

Molality - **m**

Brightness

Candela - **cd**

Speed

RPM - **rpm**

G-Force - **G**

2.8 Built-in Functions

L++ supports an expanding list of importable, common functions and methods designed around our experimental data structures. Functions can be called and used by importing the corresponding libraries in a program's source file. Here's a breakdown of the list of libraries and the functions they contain.

`math` includes basic, built-in, commonly-used mathematical functions.

`max(value1, value2)`

Returns maximum between two values, and either if they are equal. Both parameters accept numerical values (integer or floating point).

`min(value1, value2)`

Returns minimum between two values, and either if they are equal. Both parameters accept numerical values (integer or floating point).

`rand(value1, value2)`

Returns a random value in the range value1 to value2, inclusively. Performed through a uniformly distributed random function.

`normal(value1, value2)`

Returns a random value in the range value1 to value2, inclusively. Performed through a normally (gaussian) distributed random function.

`log(value [,base])`

Returns the logarithm of `value` to `base` e, or if a second parameter is passed in, returns the logarithm of `value` to a specified `base`.

`c([,base])`

Returns the complementary nucleotide to the input nucleic acid sequence(s).

`rc([,base])`

Returns the reverse complementary nucleotide to the input nucleic acid sequence(s).

3 L++ Development Examples and Debugging

The most of L++ code development would be spent on the design and debugging process. The actual writing of the code is made as easy and general as possible to clearly express the biological phenomenon.

3.1 Overview

In L++ code development, first, the goal of the biological process, or L++ code must be set. Then, input and output and/or the behavior of the biological process must be specified in terms of analog signals, using quantities of molecular species or their states.

Now the L++ code for the biological process needs to be implemented. The system behavior is ideally accomplished solely using biochemical algorithms. However, it is often challenging due to many unknown parameters of the elementary chemical reactions taking place in the system. Often implementation employs some level of abstraction and model reduction, bypassing the need for every single parameter in the elementary chemical reactions in the system at the expense of the perturbability of those parameters. L++ syntax specification and k_function's are wonderful manifestations of such abstraction.

Fortunately, there are many cellular pathways, such as *E. coli* chemotaxis, the molecular mechanism of which has been identified, and the reduced mathematical models have been worked out. We will use it as an example to highlight the L++ code writing.

3.2 L++ code development example

Designing *E. coli* chemotaxis

The assumption is that E coli would be situated where the chemical gradient is relatively small across its own body, such that it is difficult to differentiate the chemical concentration at one end vs. the other. This contrasts much bigger cells, like mammalian cells (> 10 times bigger in body length), which gives a more likelihood to be able to sense and respond to the local environment at one end of the cell in a polarized fashion.

Instead, *E. coli* relies on its ability to sense the change in chemical concentration of interest at its position. *E. coli* compares the current level vs. the previous level to decide whether it would run straight or rotate itself using counter-clockwise and clockwise flagellar motor rotation, respectively. For example, when there is a large elevation in the chemical concentration, *E. coli* would run in a straight line with more assurance. Regardless of the low or high abundance of

the chemical, it would tumble. This adaptive behavior ensures E. coli to move up or down the gradient of chemical concentration.

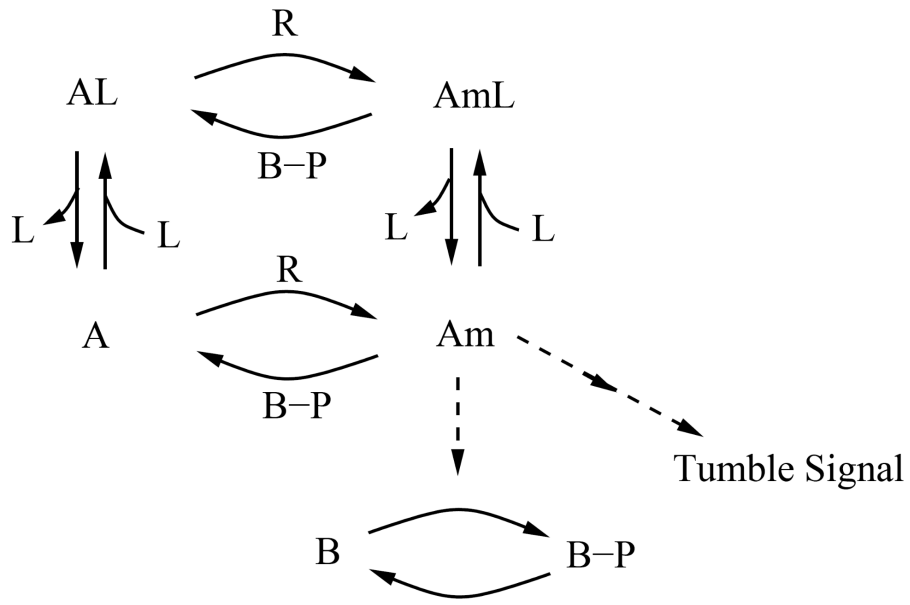
Implementing *E. coli* chemotaxis code

Now, how can one implement this using a biochemical algorithm? First, many molecular circuit designs may be considered and developed, then second, it needs to be expressed in an appropriate language, such as L++. The first part will be briefly discussed with a quote from the Ingalls 2013 textbook, and the manual will focus on how one would write out the L++ code.

The known molecular pathway can be described as follows:

> *E. coli cells use a signalling pathway to convert their measurements into an exploration strategy. This pathway, shown in Figure 6.12, transduces a signal from the transmembrane receptors (which bind attractant or repellent) to the flagellar motor. The receptors are complexed with a kinase called CheA ('Che' for chemotaxis). CheA phosphorylates the cytosolic protein CheY, which, when activated, binds to the flagellar motor and induces tumbling. Binding of chemoattractant to the receptor inhibits CheA activity; this reduces levels of active CheY, and thus inhibits tumbling. Binding of repellent activates CheA and so has the opposite effect.*
>
> *The pathway's ability to adapt to new environments is conferred by an added layer of complexity. On each receptor there are several sites that can bind methyl groups. Methylation is catalysed by an enzyme called CheR. Another enzyme, CheB, demethylates the receptors. Receptor methylation enhances CheA activity, so methylation induces tumbling. A feedback loop is provided by an additional catalytic function of CheA—it activates CheB (again by phosphorylation). This is a negative feedback on CheA activity. (When CheA is, for example, inhibited, CheB activity is reduced, which leads to more receptor methylation and hence increased CheA activity.) This feedback acts on a slower time-scale than the primary response, and causes the system to return to its nominal level of CheA activity (and thus motor behaviour) after a change in ligand concentration. This is the mechanism of adaptation.*
>
> (Ingalls, 2013, p161)

A reduced reaction network of the above text pathway description may be as follows:



(Ingalls, 2013, p163)

The model equation was described as follows:

$$
\begin{aligned}
\frac{d}{dt}[Am](t) &= k_{-1}[R] - \frac{k_1[B\text{-}P](t)\cdot[Am](t)}{k_{M1}+[Am](t)} - k_3[Am](t)\cdot[L] + k_{-3}[AmL](t) \\
\frac{d}{dt}[AmL](t) &= k_{-2}[R] - \frac{k_2[B\text{-}P](t)\cdot[AmL](t)}{k_{M2}+[AmL](t)} + k_3[Am](t)\cdot[L] - k_{-3}[AmL](t) \\
\frac{d}{dt}[A](t) &= -k_{-1}[R] + \frac{k_1[B\text{-}P](t)\cdot[Am](t)}{k_{M1}+[Am](t)} - k_4[A](t)\cdot[L] + k_{-4}[AL](t) \\
\frac{d}{dt}[AL](t) &= -k_{-2}[R] + \frac{k_2[B\text{-}P](t)\cdot[AmL](t)}{k_{M2}+[AmL](t)} + k_4[A](t)\cdot[L] - k_{-4}[AL](t) \\
\frac{d}{dt}[B](t) &= -k_5[Am](t)\cdot[B](t) + k_{-5}[B\text{-}P](t) \\
\frac{d}{dt}[B\text{-}P](t) &= k_5[Am](t)\cdot[B](t) - k_{-5}[B\text{-}P](t)
\end{aligned}
$$

(Ingalls, 2013, p162)

Instead, one can simply describe the biochemical reactions using L++ programming language. Together with the reaction parameters obtained from the reference, the corresponding L++ code can be written as below:

```
protein R
{
    reaction R1(A --> Am, kcat=1, KM=1e-10nM);
```

```
    reaction R2(AL --> AmL, kcat=1, kM=1e-10nM);
}

protein BP
{
    reaction R1(Am --> A, kcat=200, KM=1nM);
    reaction R2(AmL --> AL, kcat=1, KM=1nM);
}

protein Am
{
    reaction R(B --> BP, k=0.05e9, krev=0.005);
}

reaction R1(Am + L --> AmL, k=1e9, krev=1);
reaction R2(A + L --> AL, k=1e9, krev=1);

R = 5nM, B = 0.1nM, A = 500nM;
L[0:100] = 20nM, L[100:200] = 40nM, L[200:300] = 80nM;
```

No math, no particular user interface or input system needed. Just plain text-based coding using biologically intuitive syntax is all there is.

3.3 Debugging

There may be different types of bugs to debug in the code.

Compilation errors

Especially, for the first time users, it may take a bit of adaptation time. When there is a compilation error, the compiler error message and the location of the buggy code may be specified. The following compilation errors may be commonly encountered:

- Unsupported keywords, especially the identifiers.
- Insufficient information given to the reactions to determine the type of reaction.
- Incorrect import and inconsistent referencing.

```
proteins CheR; // instead of protein CheR;
```

When the above L++ code is compiled, the compiler will not be able to recognize CheR as a protein, but the ‘proteins’ will be assumed to be an identifier of a random molecule. indicate that incorrect keyword used here is no longer recognized as a keyword:

```
...
found identifier
Parsed Identifier: proteins
 ==> added to table
IdentifierNode <7, 0> : proteins (name), NON_FUNCTION (identifier
type),  (primitive type)
identifier printed. Now at: proteins
Assertion failed: cur, file
C:\Users\parkch\Projects\50-Omphalos\lcc.omphalos\src\lcc\ast.cxx,
line 1389
```

Biology errors

The simulation may not show the desired system behavior. Such issues often require simulation data assessment. A few common types of the biology errors would be the following:

- Missing or suboptimal object quantity
- Reaction model
- Reaction kinetic constant values
- Molecular circuitry
- Spatial context

Debugging is a key refinement and optimization step to match the desired system design at all levels of life programming.