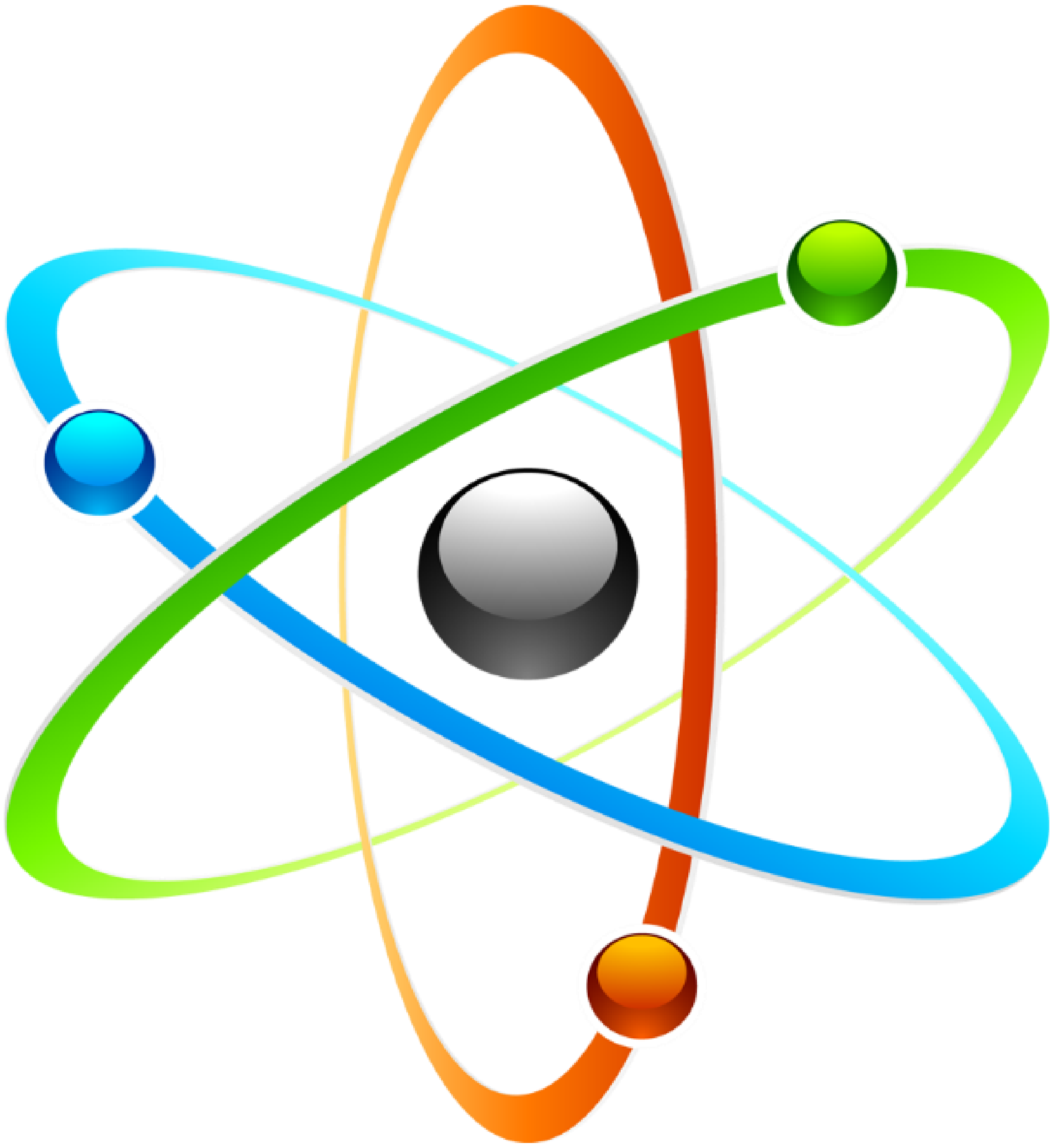
This is Kaeon FUSION

The First Ever Universal Programming Language



By Jesse Dinkin

Version 1

This is Kaeon FUSION

Use: Standard; Log Line: Open: Pitch.txt

In computer programming, it is a commonly held belief that it is inherently necessary to use multiple tools to cover the vast array of tasks a large project entails. However, we disagree with this sentiment.

The sheer, ever-growing scale of the toolset required by software developers is a burden that should have been dealt with long ago. To free developers from the bondage of incomplete tools, we have created Kaeon FUSION: the first ever universal programming language. In other words, it works effectively for all software development tasks.

Kaeon FUSION achieves its claim through its syntax which allows any hierarchy of data to function as code and through its unique ability to increase what it's capable of as it runs. At the same time, its minimalist design ensures that both newcomers and veterans can easily learn to use it.

The official online resource for Kaeon FUSION, which includes content not covered in this text, is located at the following address:

https://github.com/Gallery-of-Kaeon/Kaeon-FUSION/blob/master/README.md

Before starting with Kaeon FUSION, we recommend that you download the Kaeon Origin application from the following webpage:

https://tinyurl.com/ycpumr2p

The cover image was provided by www.freeiconspng.com at the following address:

http://www.freeiconspng.com/img/27349

For a general introduction to Kaeon FUSION, start on page 2. For the Kaeon FUSION documentation, skip to page 48.

For any questions or comments please email the author at kaeon.ace@gmail.com.

Kaeon is pronounced “KAI-on”.

A Guide to Kaeon FUSION

Preface

Computer Science – The Liberal Art

We cannot emphasize enough how important the art of computer programming is in the modern world, and how much you can benefit from having the ability to code.

Computer Science is a liberal art that allows us control over powerful machines, granting us immense power as these automatons become an extension of ourselves, and gradually freeing us from the burden of both physical and mental labor.

In calling computer science a liberal art, we use the term liberal art in the classical Greek sense, not the modern academic sense, in that it encompasses philosophy that transcends its application and that understanding it helps one grow as a person regardless of how they apply it.

What is a program?

Computer programming means giving a computer a set of instructions to perform. To write these instructions you need a programming language, which is sort of a compromise between something a person can understand and something a machine can understand. If you are indeed a newcomer to the world of programming, the good news is the options for your first language are vast.

Why Kaeon FUSION?

The bad news, is that the options for your first language are vast. Software development encompasses a vast array of domains, like web development, game programming, robotics, etc. Most programming languages are only good for small handful of them. Start with python and you'll be able to get a lot done at first but you'll hit a wall if you try to do anything heavyweight like write a game engine. Start with HTML and you'll feel helpless when you want to write code for the browser it runs on. This problem results from the fact that most languages have only so many things they allow you to do, and as a programmer you must work within these constraints. Most languages, except for Kaeon FUSION.

Instead of having a fixed set of rules built into the language, Kaeon FUSION allows functionality to be added to it while it's running using modules called interfaces. And instead of using a syntax which limits which symbols can do which things, Kaeon FUSION uses a syntax called ONE+, which allows any symbols to be arranged into a tree structure. This system not only makes Kaeon FUSION simple to learn, but allows it to encompass the needs of any software development task. Therefore, if you start with Kaeon FUSION, you may never need to touch another programming language.

A Quick Shout Out

Understanding the theory and history behind computer science always makes for a better developer. Though it is not required, we recommend you check out the web series Crash Course Computer Science if you are new to the field. We're not sponsored by them, we just think they're good at what they do. The series was created by the Green brothers and is sponsored by PBS. It is hosted by Carrie-Anne Philbin, director of the education mission for the Raspberry Pi Foundation.

Contents

Your First Program

A - 1 – Your First Program

It is tradition in the world of programming that when one either learns programming for the first time or learns a new language, they write a program called “Hello World” which displays the text “Hello, world!” on the screen.

A - 1.1 – Writing Hello World

Boot up the Kaeon Origin application and type the text in figure A - 1.1.1 into the upper text area.

A - 1.1.1

```
Use: Standard
Log Line: “Hello, world!”
```

NOTE: If this is indeed your first program, consider replacing the text between the quotation marks with something a bit more personal. Writing your first program is a special moment, so make it count!

Once you’ve done this, click the “Run” button. The text between the quotation marks should be displayed in the lower text area.

A - 1.2 – What just happened?

So, assuming you performed the above instructions correctly and the text you placed between the quotation marks appeared in the lower text area, you’re probably curious about what exactly went down when you clicked the run button.

A - 1.2.1 – ONE+

You may have taken note of the fact that bits of the text you wrote are separated by colons. As you wrote your program, you were writing in a syntax called ONE+.

Syntax refers to the system that governs how all of the symbols and words in a text document relate to each other. The first thing any programming language does when running a program is a process called tokenization, which means it uses special symbols in the document to cut the document into small chunks of text called strings. It then rearranges these strings into a tree, which is usually called an “abstract syntax tree”.

A - 1.2.1.1 – A tree? Like an apple tree?

In the context of computer science, the term “tree” refers to a certain type of graph. A graph is data structure made of little nodes. Each node stores miscellaneous data and connections to other nodes. A graph is a tree if none of the connections loop.

For example, we could have a tree with seven different nodes: A, B, C, D, E, F, and G. We’ll have A be connected to B and C, we’ll have B be connected to D and E, and we’ll have C be connected to F and G. In this example we could say that A is the parent of B and C, that B is the parent of D and E, and that C is the parent of F and G. Therefore, D and E would be the children of B, F and G would be the children of C, and B and C would be the children of A. However, if any of our connections formed a loop, we would no longer have a tree, just a graph.

Another thing to note is that the order in which the connections are stored within a node matters. For instance, if we assume A stores its connection to B before its connection to C, we could say that B is the older sibling of C, thereby making C the younger sibling of B.

Using this structure, we can interpret whatever data is stored in these nodes as a hierarchy.
In ONE+, every string created after tokenization is stored in a node called an element.

A - 1.2.1.2 – What’s special about ONE+?

Most languages have a very strict syntax that governs which chunks of text can be placed in certain spots on the abstract syntax tree. ONE+, on the other hand, allows you to define the tree however you want to.

In the case of our previous example, which for the sake of example we’ll assume you wrote “Hello, world!” between the quotation marks, the tokenization process cut the text into four chunks: “Use”, “Standard”, “Log Line”, and “”Hello, world!””.

Because “Standard” and “Hello, world!” were placed in front of “Use” and “Log Line” respectively using a colon, “Standard” became a child of “Use” and “”Hello, world”” became a child of “Log Line”. Because they were on separate lines, “Use” and “Log Line” became siblings of each other.

During the tokenization process, our ONE+ document was converted into a ONE document. ONE is a much stricter form of ONE+. ONE takes longer to write by hand than ONE+, but reading it makes the tree structure far more obvious. In the case of our ONE+ document, the corresponding ONE document looked like the text in figure A - 1.2.1.2.1.

A - 1.2.1.2.1

```
-
        Use
-
        -
                Standard
        -
-
        Log Line
-
        -
                “Hello, world!”
        -
```

NOTE: ONE code is still valid as ONE+. If you copy and paste the above text over your original code, it will still work!

It is important to note the importance of surrounding the text “Hello, world!” with quotation marks. If the text did not have a comma in it, we could have gotten away with not using quotation marks. However, commas in ONE+ normally cut text into separate siblings. If we had written our ONE+ like the text in figure A - 1.2.1.2.2, then our ONE document would have looked like the text in figure A - 1.2.1.2.3.

A - 1.2.1.2.2

```
Use: Standard
Log Line: Hello, world!
```

A - 1.2.1.2.3

```
-
        Use
-
        -
                Standard
        -
-
        Log Line
-
        -
                Hello
        -
        -
                world!
        -
```

So instead of displaying “Hello, world!”, our program would have displayed “Helloworld!”.

A - 1.2.2 – FUSION

After our document has been cut up and arranged into a tree, we want our program to do something based on the information the tree contains.

Every language has its own way of doing this. Kaeon FUSION governs how the tree is interpreted using a ruleset called the FUSION system.

FUSION will start at the first element in the document that has no parent. So for our example it starts at Use.

If the element FUSION is currently at has children, FUSION will shift focus to them. If not, FUSION will perform an operation based on the string inside the element. If the performed operation generates a value, FUSION will store this value in a list.

After performing an operation on an element, if the element has any younger siblings FUSION will shift focus to the next youngest sibling, and to the parent of the element if not. After shifting to a parent of an element it has already performed an operation on, it will perform an operation on the parent based on the string inside the parent and the values in its list. It will then clear the list.

Every time FUSION performs an operation, it may choose to jump to an element elsewhere in the document instead of proceeding to the nest sibling or parent.

Let’s use figure A - 1.2.2.1 as an example. The flow of the program according to the rules of FUSION will be as follows:

After starting at A the process trickles down to B and then to C. After performing an operation with C and generating a value, it returns that value to B. It then performs an operation with B and the value generated with C, and then moves on to D. After performing an operation and generating a value with D, it performs and operation with A and the value generated with B and D.

It of course then moves to JUMP, but after it performs an operation and generates a value, it jumps to LAND instead of moving to E.

After jumping to LAND, it trickles down to H, where it performs an operation and generates a value. It then goes back to LAND where it performs an operation and generates a value using the value generated with H. Finally, it performs an operation and generates a value with F using the value generated with LAND.

We can visualize the flow of the program using figure A - 1.2.2.2.

A - 1.2.2.1

```
-
    A
-
    -
        B
    -
        -
            C
        -
    -
        D
    -
-
    JUMP
-
-
    E
-
-
    F
-
    -
        G
    -
    -
        LAND
    -
        -
            H
        -
```

A - 1.2.2.2

C -> B(C) -> D -> A(B, D) -> JUMP -> H -> LAND(H) -> F(LAND)

A - 1.2.3 – The Use Command

The only word that Kaeon FUSION knows what to do with when it starts is “Use”, which is why we placed it first in our program.

The Use command looks at the string in each of its children, and searches the computer for an interface that matches them.

A Kaeon FUSION interface is code written in a language that only machines can understand: either machine code or byte code that teaches Kaeon FUSION what to do with certain words.

For example, the standard interface teaches Kaeon FUSION that "Log Line" means to display any value returned to it on the screen, and that any word it doesn't know should just be returned as a value minus any leading or trailing quotation marks. Such a value is called a literal.

A - 1.2.4 – Recap

So the ONE+ code we wrote in figure A - 1.1.1 got converted into the simpler ONE format as shown in figure A - 1.2.1.1.1. Then, according to the rules of FUSION, the program flowed as show in figure A - 1.2.4.1.

A - 1.2.4.1

```
        Standard -> Use (Standard) -> "Hello, world" -> Log Line ("Hello, world!")
```

Standard does not return a value, as Kaeon FUSION does not yet know what to do with anything other than Use.

Use detects that its child contains the string "Standard", so it activates the standard interface, which teaches Kaeon FUSION how to use Log Line and several other commands, and to treat any string it does not recognize as a literal.

The "Hello, world!" command, as a literal, returns its own string without the quotation marks.
Finally, Log Line takes the returned string and displays it on the screen.

NOTE

Understanding ONE+ is crucial to using Kaeon FUSION effectively. So unless you're a masochist who would prefer to write all of your Kaeon FUSION code in ONE, we recommend you read the ONE and ONE+ documentation from page 53 to page 65 before proceeding.

You may notice that Kaeon Origin has a "Show ONE" button next to the run button. Click it at any time to see how any ONE+ you've written in the upper text area translates to ONE.

Getting Comfortable

Contents

A - 2 - Getting Comfortable

Alright, you've made it this far! Give yourself a nice pat on the back.

Now that you understand the basics of how Kaeon FUSION works, you ought to know how to really take advantage of its power.

As such, this section will cover the how to effectively write code using the standard interface. We will also delve into programming topics that apply regardless of what language you use.

After each section, we recommend that you experiment with the subject the section covered before moving onto the next section.

A - 2 – 1 – Variables

A variable stores data for use later in the program. A variable has an alias, which is a string that it can be referenced by, and a value, which is a reference to data somewhere in the computer's memory.

The process of creating a new variable is called declaration. To declare a new variable, use an element containing a string that Kaeon FUSION does not recognize to serve as the alias (so you could name your variable x, but not Log Line), and nest within it a command that returns the value you want you variable to store.

NOTE: To nest an element means to make it a child of another element.

For example, we can declare a variable called "x" with a value of 5 using the code in figure A - 2 – 1.1.

A - 2 – 1.1

```
Use: Standard

x: 5
```

After the variable has been declared, you can get the value stored inside it by using an element containing its alias, and you can change the value stored inside it by nesting a command that returns the new value inside a command that contains the alias of the variable.

For example, running the code in figure A – 2 – 1.2 will display the text in figure A – 2 – 1.3.

A – 2 – 1.2

```
Use: Standard

Log Line: x

x: 5
Log Line: x

x: 10
Log Line: x
```

A - 2 – 1.3

```
x
5
10
```

NOTE: Kaeon FUSION allows you to assign any value to any variable. However, there are some languages that force you to give each variable a type, meaning that they can only store values that match that type. If a language does do this, it is called a statically typed language. If, like Kaeon FUSION, it does not, it is called a dynamically typed language.

A - 2 - 2 - Scope

Information stored within a variable can only be accessed by its siblings and children of said siblings. There is a command in the standard interface called "Scope". Scope performs no operations. It simply serves to establish a block of code isolated from the rest of the document. Any variable declaration nested within the scope command would have no effect on the rest of the document.

For example, running the code in figure A - 2 - 2.1 will display the text in figure A - 2 - 2.2.

A - 2 - 2.1

```
    Use: Standard

    x: 5

    Scope

        y: 10

        Scope

            z: 15

            Log Line: x
            Log Line: y
            Log Line: z

        Log Line: x
        Log Line: y
        Log Line: z

    Log Line: x
    Log Line: y
    Log Line: z
```

A - 2 - 2.2

```
5
10
15
5
10
z
5
y
z
```

A - 2 - 2 - 1 - Global Variables

Declaring a variable within a global command will prevent the variable from disappearing if the flow of the program moves outside the script, like when using a function, which we'll get to later.

An example of this is shown in figure A - 2 - 2 - 1.1.

A - 2 - 2 - 1.1

```
Use: Standard

Global: my variable: my value
```

A - 2 - 3 - Strings and Data Types

As discussed earlier, a command that does not match a command known by Kaeon FUSION will returns its own string as a value. Such a command is called a literal.

However, some characters have special effects if used in literals.

For example, as previously shown, quotation marks in literals block characters between them from have any effect on ONE+, but the quotation marks themselves disappear after that. In addition to being useful when your string contains commas or colons, it is also useful if you want to display something that would normally be identified as a command. Like if you wanted to display Log Line without running it as a command.

For example, running the code in figure A - 2 - 3.1 will display the text in figure A - 2 - 3.2.

A - 2 - 3.1

```
Use: Standard
Log Line: “Log Line”
```

A - 2 - 3.2

```
Log Line
```

Backslashes also allow certain characters to be placed into strings. A backslash followed by the letter 'n' will be interpreted as a new line, a backslash followed by the letter 't' will be interpreted as a tab, and a backslash followed by another backslash will be interpreted as a literal backslash.

For example, running the code in figure A - 2 - 3.3 will display the text in figure A - 2 - 3.4.

A - 2 - 3.3

```
Use: Standard
Log Line: abc\n\t\\
```

A- 2 - 3.4

```
abc
	\
```

A - 2 - 3 - 1 - String Types

Certain commands in Kaeon FUSION that take string values expect said values to be in certain formats.

For example, math commands expect the strings given to them to be in the form of numbers. A number string must consist entirely of digits, no more than one period, and no more the one minus which if present must be the first character in the string, and the last character in the string must be a digit.

Logic commands on the other hand, expect the strings given to them to be either the word "true", or "false". Letter case is irrelevant.

A - 2 - 3 - 2 - Null

The null value is a value used to sigify the absence of a value where there would otherwise be one. The null value can be returned using a command containing the string null.

A - 2 - 4 - Math

Basics math operations can be performed in the standard interface using its five basic math commands: add, subtract, multiply, divide, and modulus. Each math command has two child commands, each of which return a number. When activated, a math command will perform its respective operation on the two numbers returned to it by its children and return the result.

NOTE: A modulus operation takes the first number and divides it by the second number, but returns the remainder instead of the quotient.

For example, running the code in figure A - 2 - 4.1 will display the text in figure A - 2 - 4.2.

A - 2 - 4.1

```
Use: Standard

Log Line: Add: 1, 2
Log Line: Subtract: 3, 2
Log Line: Multiply: 2, 5
Log Line: 9, 3
Log Line: Modulus: 10, 4
```

A - 2 - 4.2

```
3
1
10
3
2
```

In addition, a random number between 0 and 1 can be generated by the random command. For example, running the code in figure A - 2 - 4.3 will display a random number between 5 and 10

A - 2 - 4.3

```
Use: Standard

min: 5: max: 10

random number: Add: min, Multiply: Random, Subtract: max, min

Log Line: random number
```

Furthermore, we can force the result to be a whole number using modulus commands, as seen in figure A - 2 - 4.4.

A - 2 - 4.4

```
Use: Standard

min: 5: max: 10

random number: Add: min, Multiply: Random, Subtract: max, min
random number: Subtract: random number, Modulus: random number, 1

Log Line: random number
```

NOTE:

While writing out math operations like this can be tedious, there is a ONE+ directive called Super Mode, which allows these operations to be written like normal math operations. Super Mode also comes in handy for dealing with variables and with the logic commands and flow control commands covered in the next sections.

You can read about Super Mode at the following webpage:

https://tinyurl.com/y88jtzp4

A - 2 - 5 - Logic

In programming, logic operations deal with values of true and false. These true or false values are called booleans.

Booleans are named after British mathematician George Boole, who created a system of processing true and false values called boolean algebra. This system laid the mathematical foundation for how computer circuitry works.

In boolean logic, which derives from boolean algebra, there are four primary functions: and, or, not, and exclusive or.

The and operation takes two boolean values and returns true if both of the boolean values it was given are true, and returns false otherwise.

The or operation takes two boolean values and returns true if at least one of the boolean values it was given are true, and returns false otherwise.

The not operation takes one boolean values and returns true if the boolean value it was given is false, and returns false if the boolean value it was given is true.

The exclusive or operation takes two boolean values and returns true if one but not both of the boolean values it was given are true, and returns false otherwise.

The Kaeon FUSION Standard Interface provides these four operations as commands.

For example, running the code in figure A - 2 - 5.1 will display the text in figure A - 2 - 5.2.

A - 2 - 5.1

```
Use: Standard

Log Line: And: True, True
Log Line: And: True, False

Log Line: Or: True, False
Log Line: Or: False, False

Log Line: Exclusive Or: True, False
Log Line: Exclusive Or: True, True

Log Line: Not: False
Log Line: Not: True
```

A - 2 - 5.2

```
True
False
True
False
True
False
True
False
```

A - 2 - 5 - 1 - Comparison

In computer science it is also important to have operations for comparison. Such operations take two values and return either a true or false boolean value depending on how the two values compare to one another.

Kaeon FUSION provides an equal command, which determines if two values are eqivilent, as well the following inequality operations, which only work with numbers: Greater, Less, Greater or Equal, and Less or Equal.

For example, running the code in figure A - 2 - 5 - 1.1 will display the text in figure A - 2 - 5 - 1.2.

A - 2 - 5 - 1.1

```
Use: Standard

Log Line: Equal: 10, Add: 5, 5

Log Line: Greater: 10, 10
Log Line: Less: 5, 10

Log Line: Greater or Equal: 10, 10
Log Line: Less or Equal: 15, 10
```

A - 2 - 5 - 1.2

```
True
False
True
True
False
```

A – 2 – 6 – Flow Control

In all of our programs written thus far, every command we place in our program will execute, and after they execute, they will not execute again.

You may recall that in the previous section: your first program, we discussed how certain commands could force the program to jump to another part of the program.

The Kaeon FUSION Standard Interface provides two such commands: loop and break, each of which have the option of having a child that returns a boolean value. In practice, they should almost always have such a child. If they have no child, they will perform their respective jump operation. If they do have a child, they will only jump if the child returns a true value.

The loop command jumps to the first child of its parent, and the break command jumps to the first sibling after its parent.

It is best practice to nest them within a Scope command.

For example, running the code in figure A – 2 – 6.1 will display the text in figure A – 2 – 6.2.

A – 2 – 6.1

```
Use: Standard

x: 5

Scope { Break: Less: x, 10 }
	Log Line: The variable x is less than 10.

Scope { Break: Greater or Equal: x, 10 }
	Log Line: The variable x is greater than or equal to 10.

i { 0 } Scope

	Log Line: i, " alligator"

	i: Add: i, 1
	Loop: Less: i, x
```

A - 2 - 6.2

```
The variable x is less than 10.
0 alligator
1 alligator
2 alligator
3 alligator
4 alligator
```

In addition, Kaeon FUSION also has a else command, which will only allow its children to execute if the most recently used break command had a child that returned false.

For example, running the code in figure A - 2 - 6.3 will display the code in figure A - 2 - 6.4.

A - 2 - 6.3

```
Use: Standard

x: 5

Scope { Break: Greater or Equal: x, 10 }
	Log Line: The variable x is greater than or equal to 10.

Else
	Log Line: The variable x is less than 10.
```

A - 2 - 6.4

```
The variable x is less than 10.
```

A - 2 - 7 - Input and Output

A typical program operates in a cycle of processing information and relaying said information to the user. A program is pointless if the user has no way to communicate with it.

For all of our examples thus far, we've been using the Log Line command to display information. The Kaeon FUSION Standard Interface provides another command called Log. The difference between Log and Log Line is that Log Line creates a new line after it displays the information given to it, whereas Log does not.

For example, running the code in figure A - 2 - 7.1 will display the text in figure A - 2 - 7.2.

A - 2 - 7.1

```
Use: Standard

Log: a
Log Line: b
Log Line: c
Log: d
Log: e
```

A - 2 - 7.2

```
ab
c
de
```

Of course, it is a given that some programs must take information from the user as well. The input command allows for this. It works like the log command, except after displaying the data it is given it will prompt the user to enter a string. It will return whatever the user enters.

For example, running the code in figure A - 2 - 7.3 will prompt the user for input using the text "Enter a word or phrase: ", and will display whatever input the user enters.

A - 2 - 7.3

```
Use: Standard

user input: Input: "Enter a word or phrase: "
Log Line: user input
```

A - 2 - 7 - 1 - Files

The Kaeon FUSION Standard Interface can read from and write to files using the open and save commands. For the sake of the following examples we'll assume you're running Kaeon FUSION from Kaeon Origin.

The open command has a child that returns a string. It will search the folder surrounding the environment Kaeon FUSION is running in for a file that matches the string. If it finds such a file it will return the contents of the file as a string. If it does not find the file in the local folder, it will check to see if the string is a URL. If it is, it will search the internet for the file.

For example, if you had a file called "My File.txt" in the local folder which contained the text shown in figure A - 2 - 7 - 1.1, then running the code shown in figure A - 2 - 7 - 1.2 would display the text shown in figure A - 2 - 7 - 1.1.

A - 2 - 7 - 1.1

```
	abc
	123
```

A - 2 - 7 - 1.2

```
	Use: Standard

	Log Line: Open: My File.txt
```

The save command will have two children that each return strings. It will create a file with the name of the second string and write the first string to it.

For example, running the code shown in figure A - 2 - 7 - 1.3 will generate a file in the local folder called “My File.txt” that contains the text in figure A - 2 - 7 - 1.1.

A - 2 - 7 - 1.3

```
	Use: Standard

	Save

		-
			abc
			123
		-

		My File.txt
```

A – 2 – 8 – Lists

So we know that we can use variables to store data, but what if we wanted to store multiple values in the same place? For that we can use lists.

A list is a value that can store other values. The Kaeon FUSION Standard Interface provides command both for creating lists and for manipulating values in them.

When displayed, a list will be presented as having all of the values inside of it placed between square brackets ans separated by commas.

The list command can be used for creating a list. It can have an indefinite number of children. When run it assembles all of the values returned to it into a list and returns the list.

For example, running the code in figure A – 2 – 8.1 will display the text in figure A – 2 – 8.2.

A – 2 – 8.1

```
Use: Standard

my list: List: a, b, c
Log Line: my list
```

A – 2 – 8.2

```
[a, b, c]
```

Additionally, it should be noted that lists can be stored inside of other lists.

For example, running the code in figure A – 2 – 8.3 will display the text in figure A – 2 – 8.4

A – 2 – 8.3

```
Use: Standard

my list: List: a, b, c, List: 1, 2, 3
Log Line: my list
```

A – 2 – 8.3

```
[a, b, c, [1, 2, 3]]
```

A - 2 - 8 - 1 - List Operations

You can use the at command to retrieve a specific value from within a list. The at command takes both a list and a number specifying the position, or the index, of the value you want to retrieve.

For example, running the code in figure A - 2 - 8 - 1.1 will display the text in figure A - 2 - 8 - 1.2.

A - 2 - 8 - 1.1

```
Use: Standard

my list: List: a, b, c

Log Line: At: my list, 1
Log Line: At: my list, 2
Log Line: At: my list, 3
```

A - 2 - 8 - 1.2

```
a
b
c
```

You can use the set command to replace a value in a list. The set command takes a list, a number specifying the position, or the index, of the value you want to change, and a value to place at that location.

For example, running the code in figure A - 2 - 8 - 1.3 will display the text in figure A - 2 - 8 - 1.4.

A - 2 - 8 - 1.3

```
Use: Standard

my list: List: a, b, c

Log Line: my list
Set: my list, 2, d
Log Line: my list
```

A - 2 - 8 - 1.4

```
[a, b, c]
[a, d, c]
```

You can use the insert command to place a value in a list without overriding anything. The set command takes a list, a number specifying the position, or the index, of where you want to insert your value, and a value to insert at that location.

For example, running the code in figure A - 2 - 8 - 1.5 will display the text in figure A - 2 - 8 - 1.6.

A - 2 - 8 - 1.5

```
Use: Standard

my list: List: a, b, c

Log Line: my list
Insert: my list, 2, d
Log Line: my list
```

A - 2 - 8 - 1.6

```
[a, b, c]
[a, d, b, c]
```

You can use the remove command to remove a value from a list. The remove command takes a list and a number specifying the position, or the index, of the value you want to remove.

For example, running the code in figure A - 2 - 8 - 1.7 will display the text in figure A - 2 - 8 - 1.8.

A - 2 - 8 - 1.7

```
Use: Standard

my list: List: a, b, c

Log Line: my list
Remove: my list, 2
Log Line: my list
```

A - 2 - 8 - 1.8

```
[a, b, c]
[a, c]
```

You can use the concatenate command to combine multiple list into a single list. The concatenate command takes an indefinite number of lists and returns them as a single list.

For example, running the code in figure A - 2 - 8 - 1.9 will display the text in figure A - 2 - 8 - 1.10.

A - 2 - 8 - 1.9

```
Use: Standard

Log Line: Concatenate: List { a, b, c }, List { 1, 2, 3 }
Log Line: Concatenate: abc, 123
```

A - 2 - 8 - 1.10

```
[a, b, c, 1, 2, 3]
abc123
```

You can use the crop command to cut a smaller list out of a larger one. The crop command take a list and two numbers, and returns the segment of the list from the first number to the second number.

For example, running the code in figure A - 2 - 8 - 1.11 will display the text in figure A - 2 - 8 - 1.12.

A - 2 - 8 - 1.11

```
Use: Standard

Log Line: Crop: List { 1, 2, 3, 4, 5 }, 2, 4
Log Line: Crop: List { 1, 2, 3, 4, 5 }, 5, 3
```

A - 2 - 8 - 1.12

```
[2, 3]
[5, 4]
```

Because strings and lists are more or less interchangeable in Kaeon FUSION, the standard interface provides the String to List command and the List to String command for converting between lists and strings.

For example, running the code in figure A - 2 - 8 - 1.13 will display the text in figure A - 2 - 8 - 1.14.

A - 2 - 8 - 1.13

```
Use: Standard

Log Line: List to String: List { 1, 2, 3 }
Log Line: String to List: 123
```

A - 2 - 8 - 1.14

```
123
[1, 2, 3]
```

NOTE: In Kaeon FUSION, list indexes start at one. However, in most other languages, they start at zero.

A - 2 - 9 - Functions

A function is a chunk of code that can be stored and reused throught the program. Every time you run a function, you can give it information called arguments, and after it finishes running the function can return a value. In other words, a making a function is like making a command out of other commands.

Functions are like variables, in that they have an alias to identify them. In addition, the same scope rules that apply to variables apply to functions.

The Kaeon FUSION Standard Interface allows you to create functions using the define command. Every child of the define command will be made into a function within the local scope, with said child being the alias of the function and the children of the child making up the function itself.

For example, we can define two functions: foo and bar, using the code in figure A - 2 - 9.1.

A - 2 - 9.1

```
Use: Standard

Define

	foo

		Log Line: abc
		Log Line: 123

	bar

		#[ Because what we're printing is the same as the name of our function,
		   we surround it in quotation marks. Otherwise, we'd be running the
		   function from within itself, which would create an infinite loop. ]#

	Log Line: "bar"
```

NOTE: When demonstrating functions to newcomers, it is tradition to name the functions "foo", "bar", and "baz".

Once a function is defined, it may be used by using its alias as a command.

For example, running the code in figure A - 2 - 9.3 would display the code shown in figure A - 2 - 9.4.

A – 2 – 9.3

```
        Use: Standard

        Define

                foo

                        Log Line: abc
                        Log Line: 123

                bar

                        Log Line: "bar"

        foo
        foo
        bar
```

A – 2 – 9.4

```
        abc
        123
        abc
        123
        bar
```

Any values returned to the alias by its children will be passed as arguments to the function. Said arguments may be accessed from within the function as a list using the arguments command.

For example, running the code in figure A – 2 – 9.5 will display the text in figure A – 2 – 9.6.

A – 2 – 9.5

```
        Use: Standard

        Define

                foo

                        Log Line: At { Arguments, 2 }, At { Arguments, 1 }

        foo: abc, 123
```

A - 2 - 9.6

```
123abc
```

You can make a function return a value by using the return command within it, and nesting any value you want to return beneath the return command. Once a return command is used within a function, the function will stop.

For example, running the code in figure A - 2 - 9.7 would display the code shown in figure A - 2 - 9.8.

A - 2 - 9.7

```
Use: Standard

Define

    foo

        Log Line: At { Arguments, 2 }, At { Arguments, 1 }

    bar
        Return: foobar

foo: abc, bar
```

A - 2 - 9.8

```
foobarabc
```

A - 2 - 10 - Objects

After running a function, the variable and other functions declared within it may be stored and reused. The stored state of the function is called an object.

Objects are a useful way of abstracting data. For example, we could have a function called "Dog" that declares variables analogous to the attributes of a dog, like name, age, weight, etc, and it could have functions within it that relect things that dogs do, like eat, bark, etc that can be affected by the variables.

If a function is called from within a new command, the new command will return its state which can be stored inside a variable. Its state can be brought into the local scope by nesting it within an in command. This should always be done within a scope command. The return command can be used after an in command to return something from within an object to the scope outside the object. If being used as an object, all variables in the function should be marked as global.

For example, running the code in figure A - 2 - 10.1 will display the text in figure A - 2 - 10.2.

A - 2 - 10.1

```
        Define: dog

                Global: name: At: Arguments, 1
                Global: age: 1
                Global: weight: 5

                Define: eat

                        weight: Add: weight, At: Arguments, 1

                Define: bark

                        Return: woof woof

        my dog: New: dog: Fiddo
        my other dog: New: dog: Fluffy

        food: 1

        Scope: In { my Dog } Log Line: name, " goes ", bark, .
        Scope: In { my Dog } eat: food
        Scope: In { my dog } Log Line: name, " weighs ", weight, " pounds."
        Scope: In { my dog } Log: name; Log: " is friends with "; In { my other dog } Log: name, .;
```

A - 2 - 10.2

```
Fiddo goes woof woof.
Fiddo weighs 6 pounds.
Fiddo is friends with Fluffy.
```

NOTE: Most programming languages don't allow you to make objects out of functions. They have an entirely seperate convention called classes that allow you to define objects.

SIDE NOTE: If a 5 pound dog could actually eat 1 pound of food at once, we'd be seriously concerned.

A - 2 - 11 - Errors

If you give a command data is isn't programmed to handle, something that would cause most other programming languages to crash, the Kaeon FUSION Standard Interface will stop running any commands after that, unless you use a catch command, which will allow the program to keep moving normally. Any commands nested within a catch command will only execute if the respective catch command was triggered.

For example, running the code in figure A - 2 - 11.1 will display the text in figure A - 2 - 11.2.

A - 2 - 11.1

```
Use: Standard

Log Line: Add: bad input
Log Line: abc

Catch: Log Line: Something went wrong.

Log Line: 123

Catch: Log Line: Something went wrong.
```

A - 2 - 12.1

```
Something went wrong.
123
```

Advanced Kaeon FUSION

Contents

A - 3 - 1 - Concurrency and Time

A program does one thing and then moves onto the next. A simple enough principle to understand, but often inconvenient. The fact is that there are several situation that can arise in programming that require the program to do multiple things at the same time.

The Kaeon FUSION Standard Interface allows you to split the flow of the program using the split command. Children of the split command will execute while the rest of moves on.

For example, running the code in figure A - 3 - 1.1 will display the text shown in figure A - 3 - 1.2.

A - 3 - 1.1

```
Use: Standard

Split

    i { 1 } Scope

        Log Line: i

        i: Add: i, 1
        Loop: Less or Equal: i, 5

i { 6 } Scope

    Log Line: i

    i: Add: i, 1
    Loop: Less or Equal: i, 10
```

A - 3 - 1.2

```
1
6
2
7
3
8
4
9
5
10
```

NOTE: When dealing with two things running at once, it's hard to pinpoint exactly what will display first, so what displays when running the code in figure A – 3 – 1.1 may vary slightly.

Additionally, it is sometimes necessary to pause the flow of a program. You can do this using the wait command. The wait command has a child that returns a number, and it will pause the program for the number of seconds that was returned to it.

For example, running the code in figure A – 3 – 1.3 will display the code in figure A – 3 – 1.4, but the program will pause for 1 second before displaying the second line.

A – 3 – 1.3

```
Use: Standard

Log Line: a
Wait: 1

Log Line: b
```

A – 3 – 1.4

```
a
b
```

It is also useful to use the time command to get the number of seconds on the computer's clock.

For example, running the code in figure A – 3 – 1.5 will display the text in figure A – 3 – 1.6 after pausing for 1 second.

A – 3 – 1.5

```
Use: Standard

start: Time
Wait: 1
end: Time

Log Line: Subtract: end, start
```

A – 3 – 1.6

```
1
```

A - 3 - 2 - Metaprogramming

If you make a string containing Kaeon FUSION code, you can run that code using the Kaeon FUSION Standard Interface's execute command.

For example, if you have a file called "My File.txt" containing the text in figure A - 3 - 2.1, then running the code in figure A - 3 - 2.2 will display the text in figure A - 3 - 2.3.

A - 3 - 2.1

```
Log Line: "Hello, world!"
```

A - 3 - 2.2

```
Use: Standard

Execute: Open: My File.txt
```

A - 3 - 2.3

```
Hello, world!
```

In addition, you can run command line arguments using the run command.

For example, if you run the code in figure A - 3 - 2.4, the Notepad application will open.

A - 3 - 2.4

```
Use: Standard

Run: Notepad
```

A – 3 – 3 – Building

Perhaps the most important thing you can do using the Kaeon FUSION Standard Interface is generate code in other languages without actually having to write any code in other languages.

Different dialects of ONE that correspond to different development domains can be provided by other interfaces, like the web interface. Using these dialects, you can write ONE+ code that doesn't do anything in Kaeon FUSION but can work with Kaeon FUSION to build something like a website. The first step is to define a function and write ONE+ code within the function that corresponds to the desired dialect. Then, use the build command. The build command should have one child specifying the name of the desired dialect, and this child should have children specifying the names of functions containing code in the given dialect.

For example, running the code shown in figure A – 3 – 3.1 will generate a file containing the HTML code shown in figure A – 3 – 3.2. If opened in a browser, this HTML file will display as a webpage with the title "My Website" and the text "Hello, world!" in the upper left corner.

A – 3 – 3.1

```
Use: Standard, Web

Define: My Website

	Head: Title: My Website
	Text: 'Hello, world!'

Build: Page: My Website
```

A – 3 – 3.2

```
<!DOCTYPE html>
<html>
	<head>
		<title>
			My Website
		</title>
	<head>
	<body>
		<p>
			Hello, world!
		</p>
	</body>
</html>
```

NOTE:

The web and machine interfaces, both of which come with Kaeon Origin, are not documented here. You can find information on them at the following webpage:

https://tinyurl.com/yab2sw9j

The Kaeon FUSION Documentation

Contents

The Foundations of Kaeon FUSION

Contents

B - 1 - The Foundations of Kaeon FUSION

Kaeon FUSION code is encoded in documents written in the ONE format, a markup format that encodes a document as a tree of strings.

The syntax of ONE is extremely minimalistic, but verbose at the same time, so ONE+, which is a syntactic abstraction of ONE, may be used when writing Kaeon FUSION code by hand.

The functionality of Kaeon FUSION is derived from the FUSION system, which allows any ONE or ONE+ document to act as code.

B - 1 - 1 - ONE

ONE is an extremely minimalistic markup language that allows the user to define a tree of string literals.

String literals may only be encoded in elements. Elements may be nested within one another.
An element is started with a minus sign followed by a new line and a tab. Every character from the first tab (exclusive) to the new line character (inclusive if and only if not the last line in the element) is encoded into the string. Every new line character must be followed either with a tab which continues the element, or by a minus sign which ends the element. There must be at least one line between the starting and ending lines.

If an element is nested within another element, every line of the element must be preceded by one tab for every level it is nested. No whitespace is permitted between elements.

A ONE file is referred to as a "document", and the proper file extension for a ONE file is ".one".

Note the following examples:

B – 1 – 1.1 – Document with one element that has no content

```
	-
	
	-
```

B – 1 – 1.2 – Document with one element that has content

```
	-
		Element
	-
```

B – 1 – 1.3 – Document with one element that has no content

```
	-
		Line 1
			Line 2
				Line 3
	-
```

NOTE: The above element encodes the following string: "Line 1\n\tLine 2\n\t\tLine3".

B – 1 – 1.4 – Document with two elements

```
-
        Element 1
-
-
        Element 2
-
```

B – 1 – 1.5 – Document with multiple elements, some of which have children

```
-
        Element 1
-
        -
                Child 1
        -
        -
                Child 2
        -
-
        Element 2
-
-
        Element 3
-
        -
                Child 1
        -
                -
                        Grand Child 1
                -
```

B - 1 - 2 - ONE+

While ONE is versatile and serves as a solid foundation for the FUSION system, it is quite cumbersome to write it by hand. ONE+ was created to compensate for this.

ONE+ provides users with more options for encoding and nesting elements. While there is only one way to express any given document in ONE, a single ONE document can be expressed in many different ways in ONE+. No matter how a ONE+ document is written, it will convert to ONE before being processed.

Because ONE+ is not only an abstraction of ONE but also a superset, any text that is valid in ONE is valid in ONE+.

However, unlike ONE, ONE+ does make use of token characters and thus requires escape sequences in certain situations.

The proper file extension for a ONE+ file is ".op".

B - 1 – 2.1 – Indentation

While ONE only allows the use of tabs for indentation, ONE+ allows either tabs or an arbitrary number of spaces. However, the use of indentation must be consistent throughout the file.

B - 1 – 2.2 – Element Definitions

ONE+ allows elements to be defined outside of element blocks. A non-blank line containing no token characters written will be converted into an element block. Any leading or trailing whitespace will be removed.

For example, the ONE+ in figure B – 1 – 2.2.1 is analogous to the ONE in figure B – 1 – 2.2.2.

B – 1 – 2.2.1

```
	abc
		xyz

	123
```

B – 1 – 2.2.2

```
-
    abc
-
    -
        xyz
    -
-
    123
-
```

B - 1 – 2.3 – Multiple Element Definitions

It is also possible to encode multiple elements into a single line. Such a line is called a "multiple element definition". Various token characters may be used to separate the elements. The token character used determines how the nest level is affected.
When nesting a child element beneath a multiple element line, the element that will become the parent is the most recent element at the nest level of the end of the line.

The token characters used in multiple element definitions are as follows:

, - Does not affect nest level
: - Increments nest level
; - Decrements nest level
(- Stores but does not affect nest level
) - Restores nest level to that stored by corresponding '('
{ - Stores and increments nest level
} - Restores nest level to that stored by corresponding '{'

Note the following examples:

The ONE+ in figure B – 1 – 2.3.1 is analogous to the ONE in figure B – 1 – 2.3.2.

B – 1 – 2.3.1

```
a, b, c
```

B - 1 - 2.3.2

```
    -
        a
    -
    -
        b
    -
    -
        c
    -
```

The ONE+ in figure B - 1 - 2.3.3 is analogous to the ONE in figure B - 1 - 2.3.4.

B - 1 - 2.3.3

```
    a: b, c
```

B - 1 - 2.3.4

```
    -
        a
    -
        -
            b
        -
        -
            c
        -
```

The ONE+ in figure B - 1 - 2.3.5 is analogous to the ONE in figure B - 1 - 2.3.6.

B - 1 - 2.3.5

```
    a: b; c
```

B – 1 – 2.3.6

```
-
          a
-
          -
                    b
          -
-
          c
-
```

The ONE+ in figure B – 1 – 2.3.7 is analogous to the ONE in figure B – 1 – 2.3.8.

B – 1 – 2.3.7

```
a ( b: c ) d
```

B – 1 – 2.3.8

```
-
          a
-
-
          b
-
          -
                    c
          -
-
          d
-
```

The ONE+ in figure B – 1 – 2.3.9 is analogous to the ONE in figure B – 1 – 2.3.10.

B – 1 – 2.3.9

```
a { b: c } d
```

B - 1 - 2.3.10

```
    -
        a
    -
        -
            b
        -
            -
                c
            -
    -
        d
    -
```

The ONE+ in figure B - 1 - 2.3.11 is analogous to the ONE in figure B - 1 - 2.3.12.

B - 1 - 2.3.11

```
    a: b
        c
```

B - 1 - 2.3.12

```
    -
        a
    -
        -
            b
        -
            -
                c
            -
```

The ONE+ in figure B - 1 - 2.3.13 is analogous to the ONE in figure B - 1 - 2.3.14.

B - 1 - 2.3.13

```
    a { b: c }
        d
```

B – 1 – 2.3.14

```
-
    a
-
    -
        b
    -
        -
            c
        -
    -
        d
    -
```

B - 1 – 2.4 – Escape Sequences

The effect of any token character can be negated by preceding it with a tilde. Likewise, the negating effect of a tilde can itself be negated with a preceding tilde. The negating tilde will not be encoded into the element.

If the letter n is preceded by a tilde, it will be encoded as a new line. If the letter t is preceded by a tilde, it will be encoded as a tab.

If a string is placed between two single quotes, the effects of all token characters between them will be negated. The single quotes will not be encoded into the string.

Double quotes have the same negating effect as single quotes, but double quotes will be encoded into the string.

In addition, no token character will take effect if placed inside an element block.

Note the following examples:

The ONE+ in figure B – 1 – 2.4.1 is analogous to the ONE in figure B – 1 – 2.4.2.

B – 1 – 2.4.1

```
a~: b
```

B – 1 – 2.4.2

```
-
    a: b
-
```

The ONE+ in figure B – 1 – 2.4.3 is analogous to the ONE in figure B – 1 – 2.4.4.

B – 1 – 2.4.3

```
a~~: b
```

B – 1 – 2.4.4

```
-
        a~
-
        -
                b
        -
```

The ONE+ in figure B – 1 – 2.4.5 is analogous to the ONE in figure B – 1 – 2.4.6.

B – 1 – 2.4.5

```
‘a: b’
```

B – 1 – 2.4.6

```
-
        a: b
-
```

The ONE+ in figure B – 1 – 2.4.7 is analogous to the ONE in figure B – 1 – 2.4.8.

B – 1 – 2.4.7

```
“a: b”
```

B – 1 – 2.4.8

```
-
        “a: b”
-
```

The ONE+ in figure B – 1 – 2.4.9 is analogous to the ONE in figure B – 1 – 2.4.10.

B – 1 – 2.4.9

```
hello~n~tworld
```

B – 1 – 2.4.10

```
-
        Hello
                world
-
```

The ONE+ in figure B – 1 – 2.4.11 is analogous to the ONE in figure B – 1 – 2.4.12

B – 1 – 2.4.11

```
-
        a: hello~n~tworld
-
```

B – 1 – 2.4.12

```
-
        a: hello~n~tworld
-
```

B - 1 – 2.5 – Comments

Any part of a line preceded by a pound sign will be commented out. A pound sign followed by an open square bracket dictates the start of a comment block. A comment block is closed with a closed square bracket followed by a pound sign.

Note the following examples:

The ONE+ in figure B – 1 – 2.5.1 is analogous to the ONE in figure B – 1 – 2.5.2.

B – 1 – 2.5.1

```
hello # world
```

B – 1 – 2.5.2

```
    -
            hello
-
```

The ONE+ in figure B – 1 – 2.5.3 is analogous to the ONE in figure B – 1 – 2.5.4.

B - 1 - 2.5.3

```
hello #[
abc
123
xyz ]#
world
```

B - 2 - 2.5.4

```
-
	hello
-
-
	world
-
```

B - 1 - 2.6 - Directives

Directives serve as preprocessor scripts for ONE+. They are written as multiple element lines encased in square brackets, and the directives available depend on the implementation of the ONE+ parser.

See figure B - 1 - 2.6.1 as an example.

B - 1 - 2.6.1

```
[DIRECTIVE NAME: DIRECTIVE ARGUMENT 1, DIRECTIVE ARGUMENT 2]
```

B - 1 - 2 - 1 - Standard ONE+ Directives

The directives usable when writing ONE+ vary by parser implementation, but the directives described here should be available in the majority of implementations.

B - 1 - 2 - 1.1 - DEFINE

The define directive allows a series of elements to be stored and reused.

The define directive begins with the element "DEFINE". Nested within this element is the name of the defined elements.

Nested beneath the directive are the defined elements.

See figure B - 1 - 2 - 1.1.1 as an example.

B – 1 – 2 – 1.1.1

```
[DEFINE: List]

        abc
        xyz
        123
```

B - 1 - 2 – 1.2 - USE

The use directive allows a series of elements defined within a using a define directive to be pasted elsewhere.

The use directive begins with the element "USE". Nested within this element is the name of the defined elements.

For example, the ONE+ code in figure B – 1 – 2 – 1.2.1 is analogous to the ONE code in figure B – 1 – 2 – 1.2.2.

B – 1 – 2 – 1.2.1

```
[DEFINE: List]

        abc
        xyz
        123

[USE: List]
```

B – 1 – 2 – 1.2.2

```
abc
xyz
123
abc
xyz
123
```

B - 1 - 2 – 1.3 – IMPORT

The import directive allows several series of elements defined in a ONE+ document using the define directive to be used in another document.

The define directive begins with the element "IMPORT". Nested within this element is the filepath of the referenced ONE+ document.

For example, if we have a file in the local folder called "My File.op" containing the ONE+ code in figure B – 1 – 2 – 1.3.1, then the ONE+ code in figure B – 1 – 2 – 1.3.2 is analogous to the ONE code in figure B – 1 – 2 – 1.3.3.

B – 1 – 2 – 1.3.1

```
[DEFINE: List]

        abc
        xyz
        123
```

B – 1 – 2 – 1.3.2

```
[IMPORT: My File.op]

[USE: List]
[USE: List]
```

B – 1 – 2 – 1.3.3

```
abc
xyz
123
abc
xyz
123
```

B - 1 - 3 - FUSION

FUSION is a system that allows any ONE document, and by extension any ONE+ document to serve as code.

B - 1 – 3.1 – Order of Operations

B - 1 – 3.1.1 – Step One

Based on the string encoded in the element, FUSION will perform an operation.

B - 1 – 3.1.2 – Step Two

If the element has no children, FUSION will skip to step three.

If the element does have children, then based on the string encoded in the element, FUSION will decide whether or not to trickle down to the element's children.

If it decides to do so, FUSION will shift focus the element's first child, increment the depth value by one, and jump back to step one. Otherwise, FUSION will move to step three.

B - 1 – 3.1.3 – Step Three

Based on the content of the element and any values generated by performing step three on the element's children, FUSION will perform an operation and generate a value.

B - 1 – 3.1.4 – Step Four

Based on the content of the element, FUSION will decide whether to modify the depth value or shift focus to another element.

B - 1 – 3.1.5 – Step Five

If the element is followed by a sibling element, FUSION will shift focus to the next sibling and loop back to step one.

Otherwise, if the element has a parent and if the depth value is greater than zero, FUSION will shift focus to the element's parent and decrement the depth value by one.

If the depth value is less than or equal to zero or if the element does not have a parent, FUSION will terminate.

B - 1 - 3.2 - Error Handling

If an error occurs during at any step, FUSION will skip over the step and proceed as if nothing happened.

Interfaces and the Use Command

B - 2 - Interfaces and the Use Command

Kaeon FUSION only has one command at runtime: the use command. The sole purpose of this command is to allow additional functionality to be added to the language at runtime through interfaces.

B - 2.1 – Using the Use Command

If an interface is available within the environment Kaeon FUSION is running in, it can be incorporated with an element whose content matches the name of the interface nested within an element containing the content "Use".

The figure in 2.1.1 shows a line of Kaeon FUSION code that incorporates the standard interface, if said interface is available

B - 2.1.1 – Incorporating the Standard interface

```
Use: Standard
```

The Standard Interface

Contents

B - 3 – The Standard Interface

The Kaeon FUSION Standard Interface establishes the basic functionality for writing Turing complete scripts and for integrating the features of other interfaces.

In doing so, it provides commands for console IO, arithmetic, logic, list operations, string operations, file IO, flow control, threading, metaprogramming, and object orientation.

As such, one might consider the functionality it provides to be the flagship functionality of Kaeon FUSION.

The Standard Interface can be activate using the code shown in figure B – 3.1

B – 3.1

```
	Use: Standard
```

B - 3 – 1 – Basics

B - 3 – 1.1 – Data Types

There is no limit on what data types can be generated by commands or stored by variables in Kaeon FUSION.

B - 3 – 1.2 – Literals

An element with no children that does not match the name of a command or variable will be interpreted as a string literal. In addition, any element whose content is encased in double quotes will return its content as a string literal, minus the double quotes.

The code in figure B – 3 – 1.2.1 and the code in figure B – 3 – 1.2.2 both return the literal string "Hello world".

B – 3 – 1.2.1

```
"Hello world"
```

B – 3 – 1.2.2

```
Hello world
```

B - 3 – 1.2 - 1 – Numbers

A number string must consist entirely of digits and no more than one period and no more than one minus sign. If the period is present, it must not be the last character in the string. If the minus sign is present, it must be the first character in the string.

B - 3 – 1.2 - 2 – Booleans

A boolean string may either take the form of "True" or "False". Letter case is irrelevant.

B - 3 – 1.2 - 3 – Null

Running the command "Null" returns a null value.

B - 3 – 1.3 – Variables

An element with one child that does not match the name of a command will assign the value returned by its child to a variable with the alias being the element's content.

For example, running the code in figure B – 3 – 1.3.1 will assign the string “Hello” to the variable x and running the code in figure B – 3 – 1.3.2 after running the code in figure B – 3 – 1.3.1 will overwrite the value stored in the variable x with the string “Goodbye”.

B – 3 – 1.3.1

```
x: Hello
```

B – 3 – 1.3.2

```
x: Goodbye
```

B - 3 – 1.4 – Function Definitions

A function definition begins with the "Define" command. Nested with the "Define" command is a string containing the name of the function. Nested underneath the name of the function is the content of the function.

See the code in figure B – 3 - 1.4.1 for an example.

B – 3 - 1.4.1

```
Define: foo

    Command 1: Command Argument 1
    Command 2: Command Argument 1, Command Argument 2
```

B - 3 – 1.5 – Function Calls

A defined function can be called by placing its name inside an element and nesting any arguments to be passed to the function within the element.

For example, running the code in figure B – 3 – 1.5.1 will run the function foo with an argument of the literal “Function Argument”.

B - 3 - 1.5.1

```
Define: foo

        Command 1: Command Argument 1
        Command 2: Command Argument 1, Command Argument 2

foo: Function Argument
```

B - 3 - 1.6 - Stored Function Calls

If a function call is nested within a "New" command., the "New" command will return the state of the function after it executes.

For example, running the code in figure B - 3 - 1.6.1 will run the function foo with an argument of the literal "Function Argument", and store the state of the function in the variable x.

B - 3 - 1.6.1

```
Define: foo

        Command 1: Command Argument 1
        Command 2: Command Argument 1, Command Argument 2

x: New: foo: Function Argument
```

B - 3 - 1.7 - State and Scope

Every time the Kaeon FUSION Standard Interface trickles down, it generates a new scope. Variables and functions generated by the children of a command are erased after the program bubbles back up to said command. If the flow of the program jumps, all scopes not shared by both the jumping and landing commands will be erased.

B - 3 - 1.8 - Indexes

Indexes for list and string operations in the Kaeon FUSION Standard Interface start at 1.

B - 3 - 1.9 - Strings as lists

Strings may be passed to List related commands, where they will be interpreted as lists for which each character is an element.

B - 3 - 2 - Core Commands

B - 3 - 2.1 - The Import Command

The Import command will import the function definitions from a specified Kaeon FUSION file.

For example, running the code in figure B - 3 - 2.1.1 will allow access to any functions defined in the root layer of the file "My Kaeon FUSION.op".

B - 3 - 2.1.1

```
Import: My Kaeon FUSION.op
```

B - 3 - 2.2 - The Global Command

The Global command will make a given variable alias globally accessible.

For example, running the code is both figure B - 3 - 2.2.1 and B - 3 - 2.2.2 will make the variable x globally accessible.

B - 3 - 2.2.1

```
Global: x
```

B - 3 - 2.2.2

```
Global: x: 5
```

B - 3 - 2.3 - The Catch Command

The Catch command will allow the program to instantly recover from a thrown exception. If at any point an exception is thrown, all subsequent commands will fail to activate until a catch command is reached. The children of a catch command will be activated if and only if it catches an exception.

For example, in figure B - 3 - 2.3.1, let's say that "Faulty Command" throws an exception due to "Bad Argument". "Normal Command 1" will then be passed over, and "Catch" will activate. As a result, both "Recover Command 1" and "Normal Command 1" will run. However, "Recover Command 2" will not run as the exception has already been caught.

B - 3 - 2.3.1

```
Faulty Command: Bad Argument
Normal Command 1
Catch: Recover Command 1
Normal Command 2
Catch: Recover Command 2
```

B - 3 - 2.4 - The This Command

The This command will return a pointer to the state of the current script, similar to how a function call returns a pointer to the function's state.

B - 3 - 2.5 - The Arguments Command

It is possible to execute a Kaeon FUSION script or a Kaeon FUSION function with arguments. The Arguments command returns a list containing the arguments that were passed to the script or function.

For example, running the code in figure B - 3 - 2.5.1 will display the text in figure B - 3 - 2.5.2.

B - 3 - 2.5.1

```
Use: Standard

Define: foo

	#[ The "Log Line" command comes from the standard interface,
	   and prints the values returned by its children to the console. ]#

	Log Line: Arguments

foo: 1, 2, 3
```

B - 2 - 2.5.2

```
[1, 2, 3]
```

B - 3 - 2.6 - The Return Command

It is possible for a Kaeon FUSION script or a Kaeon FUSION function to be called externally. The Return command will terminate the currently executing function or script and return the value returned by its first child to the caller.

For example, running the code in figure B – 3 – 2.6.1 will assign the literal string "y" to the variable x.

B – 3 – 2.6.1

```
Define: foo
    Return: y

x: foo
```

B – 3 – 2.7

Kaeon FUSION interfaces can provide the ability to code for various domains by specifying custom ONE dialects that meet the needs of those domains. Code written in these custom dialects can be nested within function definitions and cross compiled to other languages using the Build command.

The Build command takes one child that specifies the name of the custom ONE dialect. The children of this element specify functions containing code written in this dialect. The Build command may also take an indefinite number of other children as additional arguments.

For example, the code in figure B – 3 - 2.7.1 will cross compile the ONE markup nested within the Code function to another language.

B – 3 - 2.7.1

```
Define: Code
    # Code written in custom ONE dialect

Build: Name of Dialect { Code } Argument 1, Argument 2
```

B – 3 – 2.8 – The Meta Command

The Meta command can have an indefinite number of children, does not trickle down, and performs no operation. Its children can act as localized arguments when cross compiling a custom ONE dialect using the build command.

For example, in the code in figure B – 3 – 2.8.1, the Build command will take into account the arguments nested within the Meta commands when cross compiling.

B - 3 - 2.8.1

```
Define: Code

        Meta { #[ Arguments for custom ONE dialect ]# }
        # Code written in custom ONE dialect

        Meta { #[ Different arguments for custom ONE dialect ]# }
        # Code written in custom ONE dialect

Build: Name of Dialect { Code } Argument 1, Argument 2
```

B – 3 – 3 – Console IO Commands

B – 3 – 3.1 – Log

The Log command will print every value returned by its children to the console in order.

For example, running the code in figure B – 3 – 3.1.1 will display the text in figure B – 3 – 3.1.2.

B – 3 – 3.1.1

```
Log: a, b, c
Log: x, y, z
```

B – 3 – 3.1.2

```
abcxyz
```

B – 3 – 3.2 – Log Line

The Log Line command will print every value returned by its children to the console in order, plus a new line character.

For example, running the code in figure B – 3 – 3.2.1 will display the text in figure B – 3 – 3.2.2.

B – 3 – 3.2.1

```
Log Line: a, b, c
Log Line: x, y, z
```

B – 3 – 3.2.2

```
abc
xyz
```

B – 3 – 3.3 – Input

The Input command will print every value returned by its children to the console in order, prompt the user for string input, and return the given response.

For example, running the code in figure B - 3 - 3.3.1 will prompt the user for input and assigning whatever the user enters to the variable x.

B - 3 - 3.3.1

```
x: Input: "Enter a number: "
```

B - 3 - 4 - Arithmetic Commands

B - 3 - 4.1 - The Add Command

The Add command takes two string values in the form of numbers. It returns the sum of the two numbers as a string.

For example, the command in figure B - 3 - 4.1.1 will return 3.

B - 3- - 4.1.1

```
Add: 1, 2
```

B - 3 - 4.2 - The Subtract Command

The Subtract command takes two string values in the form of numbers. It returns the difference of the two numbers as a string.

For example, the command in figure B - 3 - 4.2.1 will return 1.

B - 3- - 4.2.1

```
Subtract: 2, 1
```

B - 3 - 4.3 - The Multiply Command

The Multiply command takes two string values in the form of numbers. It returns the product of the two numbers as a string.

For example, the command in figure B - 3 - 4.2.1 will return 10.

B - 3- - 4.3.1

```
Multiply: 5, 2
```

B - 3 - 4.4 - The Divide Command

The Divide command takes two string values in the form of numbers. It returns the quotient of the two numbers as a string.

For example, the command in figure B – 3 – 4.4.1 will return 5.

B – 3- - 4.4.1

```
        Subtract: 10, 2
```

B – 3 – 4.5 – The Modulus Command

The Add command takes two string values in the form of numbers. It returns the remainder of the two numbers as a string.

For example, the command in figure B – 3 – 4.5.1 will return 1.

B – 3- - 4.2.1

```
        Subtract: 10, 3
```

B – 3 – 5 – Logic Commands

B – 3 – 5.1 – The And Command

The And command takes two boolean strings and returns "True" if they are both "True", and returns "False" otherwise.

For example, the command in figure B – 3 – 5.1.1 will return “True”, and the command in figure B – 3 – 5.1.2 will return “False”.

B – 3- - 5.1.1

```
And: True, True
```

B – 3- - 5.1.2

```
And: True, False
```

B – 3 – 5.2 – The Or Command

The Or command takes two boolean strings and returns "True" if either of them are "True", and returns "False" otherwise.

For example, the command in figure B – 3 – 5.2.1 will return “True”, and the command in figure B – 3 – 5.2.2 will return “False”.

B – 3- - 5.2.1

```
Or: True, False
```

B – 3- - 5.2.2

```
Or: False, False
```

B – 3 – 5.3 – The Exclusive Or Command

The Or command takes two boolean strings and returns "True" if either of them are "True", and returns "False" otherwise.

For example, the command in figure B – 3 – 5.3.1 will return “True”, and the command in figure B – 3 – 5.3.2 will return “False”.

B - 3- - 5.3.1

Exclusive Or: True, False

B - 3- - 5.3.2

Exclusive Or: True, False

B - 3 - 5.3 - The Exclusive Or Command

The Exclusive Or command takes two boolean strings and returns "True" if only one of them is "True", and returns "False" otherwise.

For example, the command in figure B - 3 - 5.3.1 will return “True”, and the command in figure B - 3 - 5.3.2 will return “False”.

B - 3- - 5.3.1

Exclusive Or: True, False

B - 3- - 5.3.2

Exclusive Or: True, False

B - 3 - 5.4 - The Not Command

The Not command takes one boolean string and returns the opposite of its input.

For example, the command in figure B - 3 - 5.4.1 will return “True”, and the command in figure B - 3 - 5.4.2 will return “False”.

B - 3- - 5.4.1

Not: False

B - 3- - 5.4.2

Not: True

B - 3 - 5.5 - The Equal Command

The Equal command takes two strings and returns "True" if they are both the same, and returns "False" otherwise.

For example, the command in figure B - 3 - 5.5.1 will return “True”, and the command in figure B - 3 - 5.5.2 will return “False”.

B - 3- - 5.5.1

```
Equal: hello, hello
```

B - 3- - 5.5.2

```
Equal: hello, goodbye
```

B - 3 - 5.6 - The Greater Command

The Greater command takes two number strings and returns "True" if the first number is greater than the second, and returns "False" otherwise.

For example, the command in figure B - 3 - 5.6.1 will return “True”, and the command in figure B - 3 - 5.6.2 will return “False”.

B - 3- - 5.5.1

```
Greater: 2, 1
```

B - 3- - 5.5.2

```
Greater: 1, 2
```

B - 3 - 5.7 - The Greater or Equal Command

The Greater command takes two number strings and returns "True" if the first number is greater than or equal to the second, and returns "False" otherwise.

For example, the command in figure B - 3 - 5.7.1 will return “True”, and the command in figure B - 3 - 5.7.2 will return “False”.

B - 3- - 5.7.1

```
Greater or Equal: 2, 1
```

B - 3- - 5.7.2

```
Greater or Equal: 1, 2
```

B - 3 - 5.8 - The Less Command

The Less command takes two number strings and returns "True" if the first number is less than the second, and returns "False" otherwise.

For example, the command in figure B - 3 - 5.8.1 will return “True”, and the command in figure B - 3 - 5.8.2 will return “False”.

B - 3- - 5.8.1

```
Less: 1, 2
```

B - 3- - 5.8.2

```
Less: 2, 1
```

B - 3 - 5.9 - The Less or Equal Command

The Less command takes two number strings and returns "True" if the first number is less than or equal to the second, and returns "False" otherwise.

For example, the command in figure B - 3 - 5.9.1 will return “True”, and the command in figure B - 3 - 5.9.2 will return “False”.

B - 3- - 5.9.1

```
Less or Equal: 1, 2
```

B - 3- - 5.9.2

```
Less or Equal: 2, 1
```

B - 3 - 6 - List Operation Commands

B - 3 - 6.1 - The List Command

The List command places all of the values returned by its children into a list and returns it.

For example, the command in figure B - 3 - 6.1.1 will return the list [1, 2, 3].

B - 3 - 6.1.1

```
List: 1, 2, 3
```

B - 3 - 6.2 - The Size Command

The Size command takes a list and returns the size of it.

For example, the command in figure B - 3 - 6.2.1 will return 3.

B - 3 - 6.2.1

```
Size: List: 1, 2, 3
```

B - 3 - 6.3 - The At Command

The At command takes a list and an integer and returns the value in the list at the given integer.

For example, the command in figure B - 3 - 6.3.1 will return 5.

B - 3 - 6.3.1

```
At: List { 4, 5, 6 } 2
```

B - 3 - 6.4 - The Append Command

The Append command takes a list and a value and appends the value to the list.

For example, the code in figure B - 3 - 6.4.1 will append 4 to the list “my list”, making the content of “my list” [1, 2, 3, 4].

B - 3 - 6.4.1

```
my list: List: 1, 2, 3
Append: my list, 4
```

B - 3 - 6.5 - The Set Command

The Set command takes a list, an integer, and a value. It sets the value of the list at the index specified by the integer to the given value. If the index is greater than the length of the list, null values will be appended to the list until it reaches the required size.

For example, the code in figure B - 3 - 6.5.1 will replace 2 with 4 in the list “my list”, making the content of “my list” [1, 4, 3], and the code in figure B - 3 - 6.5.2 will append a null value followed by a value of 4 to the list “my list”, making the content of “my list” [1, 2, 3, null, 4],

B - 3 - 6.5.1

```
my list: List: 1, 2, 3
Set: my list, 2, 4
```

B - 3 - 6.5.2

```
my list: List: 1, 2, 3
Set: my list, 5, 4
```

B - 3 - 6.6 - The Insert Command

The Insert command takes a list, an integer, and a value. It appends the given value to the list at the index specified by the integer. If the index is greater than the length of the list, null values will be appended to the list until it reaches the required size.

For example, the code in figure B - 3 - 6.6.1 will insert a value of 4 at index 2 in the list “my list”, making the content of “my list” [1, 4, 2, 3], and the code in figure B - 3 - 6.6.2 will append a null value followed by a value of 4 to the list “my list”, making the content of “my list” [1, 2, 3, null, 4],

B - 3 - 6.6.1

```
my list: List: 1, 2, 3
Insert: my list, 2, 4
```

B - 3 - 6.6.2

```
my list: List: 1, 2, 3
Insert: my list, 5, 4
```

B - 3 - 6.7 - The Remove Command

The Remove command takes a list and an integer. It removes the value in the list at the specified integer and returns the value.

For example, the code in figure B - 3 - 6.7.1 will the value at index 2 from the list “my list” and assign said value to the variable x, making the content of “my list” [1, 3].

B - 3 - 6.7.1

```
my list: List: 1, 2, 3
x: Remove: my list, 2
```

B - 3 - 6.8 - The Concatenate Command

The Concatenate command takes an indefinite number of lists and returns a single list containing each element from each list it was passed. If the first list it was passed was a string, it will return the list in the form of a string.

For example, running the code in figure B- 3 - 6.8.1 will display the text in figure B - 3 - 6.8.2.

B - 3 - 6.8.1

```
Log Line: Concatenate: List { a, b, c }, List { 1, 2, 3 }
Log Line: Concatenate: "abc", "123"
```

B - 3 - 6.8.2

```
[a, b, c, 1, 2, 3]
abc123
```

B - 3 - 6.9 - The Crop Command

The crop command takes a list and two numbers. It returns a list containing the elements in the list from the first number to the second number.

For example, running the code in figure B- 3 – 6.9.1 will display the text in figure B – 3 – 6.9.2.

B – 3 – 6.9.1

```
Log Line: Crop: List { 1, 2, 3, 4, 5 }, 2, 4
Log Line: Crop: List { 1, 2, 3, 4, 5 }, 5, 3
```

B – 3 – 6.9.2

```
[2, 3]
[5, 4]
```

B – 3 – 6.10 – String to List

The List to String command takes a list and concatenates each element in the list to a string, which it returns.

B – 3 – 6.11 – List to String

The String to List command takes a string and places each character in the string into a list, which it returns.

B - 3 - 7 - File IO Commands

B - 3 - 7.1 - The Open Command

The Open command takes a string and returns the contents of the file located at the path specified by the string.

For example, if you have a file called "My File.txt" and it contains the text in figure B - 3 - 7.1.1, then the command in figure B - 3 - 7.1.2 will return the text in figure B - 3 - 7.1.1.

B - 3 - 7.1.1

```
abc
xyz
123
```

B - 3 - 7.1.2

```
Open: My File.txt
```

B - 3 - 7.2 - The Save Command

The Save command takes two string arguments. It writes the contents of the first during to the file at the path specified by the second string.

For example, the command in figure B - 3 - 7.2.1 will generate a file called “My File.txt” that contains the content “Hello”

B - 3 - 7.2

```
Save: hello, My File.txt
```

B - 3 - 8 - Flow Control Commands

B - 3 - 8.1 - The Scope Command

The Scope command performs no operations. It merely serves to establish an isolated scope in which its child commands proceed to execute.

See figure B - 3 - 8.1.1 for an example.

B - 3 - 8.1.1

```
Scope

    # Code
```

B - 3 - 8.2 - The Break Command

The Break command stops the execution of all commands in its scope and causes FUSION to bubble up. It has the option of take a boolean as an argument. If the boolean value is "False" the Break command will not take effect.

For example, the running the code in figures B - 3 - 8.2.1 and B - 3 - 8.2.2 will display nothing.

B - 3 - 8.2.1

```
Scope { Break }
    Log Line: Success
```

B - 3 - 8.2.2

```
x: hello

Scope { Break: Equal: x, hello }
    Log Line: Success
```

B - 3 - 8.3 - The Else Command

An Else command will only allow its child commands to execute if the most recently used Break command used since the most recently used Else command activated.

For example, the code in figure B – 3 – 8.3.1 will display “Success”, and running the code in figure B – 3 – 8.3.2 will display “Failure.

B – 3 – 8.3.1

```
x: hello

Scope { Break: Not: Equal: x, hello }
    Log Line: Success

Else
    Log Line: Failure
```

B – 3 – 8.3.2

```
x: hello

Scope { Break: Equal: x, hello }
    Log Line: Success

Else
    Log Line: Failure
```

B – 3 – 8.4 – The Loop Command

The Loop command jumps to the first child command of its parent command. It has the option of take a boolean as an argument. If the boolean value is "False" the Break command will not take effect.

For example, the running the code in figure B – 3 – 8.4.1 will display the text in figure B – 3 – 8.4.2.

B – 3 – 8.4.1

```
i { 0 } Scope

    Log Line: i

    i: Add: i, 1
    Loop: Less: i, 10
```

B - 3 - 8.4.2

```
0
1
2
3
4
5
6
7
8
9
```

B - 3 - 8.5 - The Throw Command

The Throw command will automatically throw an exception.

For example, the running the code in figure B - 3 - 8.5.1 will display the text in figure B - 3 - 8.5.2.

B - 3 - 8.5.1

```
Log Line: abc
Throw
Log Line: xyz
Catch: Log Line: Oops
Log Line: 123
```

B - 3 - 8.5.2

```
abc
Oops
123
```

B - 3 - 8.6 - The Exit Command

The Exit command will immediately stop the execution of Kaeon FUSION.

B – 3 – 9 – Threading Commands

B – 3 – 9.1 – The Split Command

The Split command will create execute its children on a new thread.

For example, running the code in figure B – 3 – 9.1.1 will, with the possibility of slight variations, display the text in figure B – 3 – 9.1.2.

B – 3 – 9.1.1

```
Split

        i { 0 } Scope

                Log Line: i

                i: Add: i, 1
                Loop: Less: i, 5

i { 5 } Scope

        Log Line: i

        i: Add: i, 1
        Loop: Less: i, 10
```

B – 3 – 9.1.2

```
0
5
1
6
2
7
2
8
4
9
```

B - 3 - 9.2 - The Wait Command

The wait command will take a number and pause the current thread for that many seconds.

B - 3 - 9.3 - The Time Command

The time command will return the value on the computer's clock in seconds.

B – 3 – 10 – Metaprogramming Commands

B – 3 – 10.1 – The Execute Command

The Execute command takes a string and executes it as Kaeon FUSION code.

For example, if we have a file in the local directory called “My Code.op” containing the text in figure B – 3 – 10.1.1, then running the code in figure B – 3 – 10.1.2 will display the text in figure B – 3 – 10.1.3.

B – 3 - 10.1.1

```
Log Line: Hello
```

B – 3 – 10.1.2

```
Execute: Open: My Code.op
```

B – 3 – 10.1.3

```
Hello
```

B – 3 – 10.2 – The Run Command

The Run command takes a string and passes it to the command line.

For example, running the code in figure B – 3 – 10.2.1 on a Windows operating system will open the Notepad application.

B – 3 – 10.2.1

```
Run: Notepad
```

B - 3 - 11 - Object Orientation

Kaeon FUSION does not support classes, but object orientation can be achieved by using functions as objects.

The state of the function can be stored within a variable after the function returns by nesting the function call beneath a "New" command, and assigning the value returned by the new command to a variable.

The In command will take the stored state of a function and execute all of the commands following it within the stored state. For this reason, the In command should always be nested inside another command. The In command will also return a value if the Return command is used within it.

For example, running the code in figure B - 3 - 11.1 will display the text in figure B - 3 - 11.2.

B - 3 - 11.1

```
Define: foo

	x: At: Arguments, 1
	y: 10

my foo: New: foo: 5

Scope { In: my foo }
	z: Add: x, y

Log Line: In { my foo } Return: z
```

B - 3 - 11.2

```
15
```