LAMBDA+

Laboratory Analysis & Mechanical Brogue for Data Application

Nicholas Fiore

Nicholas.Fiore1@Marist.edu

May 5, 2023

```
                ..;clodxxkxxxdoc:,.
            .;ok0KK00Okkkk00KK0Odc'.
         .,oOKKOdc;'.........',:ok0K0x:.
       ;x0K0d;.                   .,lkKKkc.
    .o0KOo'                          .:kKKx,
   'xKKx,                              .o0KO:
  'kKKd.                                .cOKO:
 .dKKx.                                   c0Kk'
 :0KO;                                    .xKKl
.oKKx.                                     lKKx.
.dKKd.           .lddddddddddddddddo;.     c0Kk.
.oKKx.           'kKKKKKKKKKKKKKKK0d'      cKKx.
 c0KO;           'kKKKKKKKKKKKKKKKKOc..dKKo.
 'xKKd.          'kKKKKKKKKKKKKKKKKK0xd0KO;
  ;OK0o.         'kKKKKKKKKKKKKKKKKKKKK0l.
   :OK0d:,,,,,,,lOKKKKKKKKKKKKKKKKKKKK0l.
   ,xKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKOc.
     .cOKKKKKKKKKKKKKKKKKKKKKKKKKKKKKK0d'
       .ck0KKKKKKKKKKKKKKKKKKKKKKKKOd,
         .,lx0KKKKKKKKKKKKKKKK0Od:.
            ..;codkOOOOOOOkxdl:,.
```

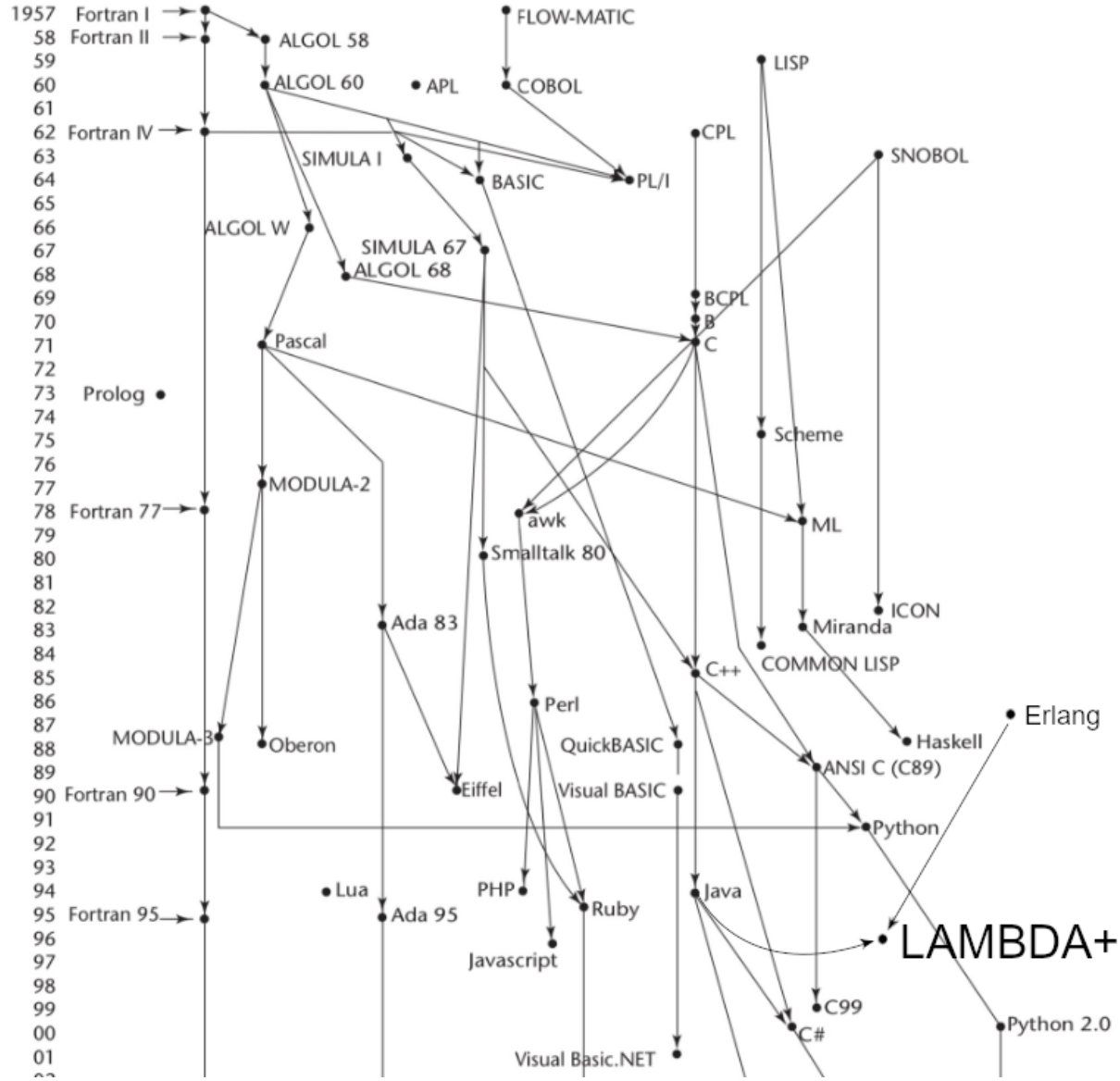
Contents

1 Introduction

LAMBDA+ (Laboratory Analysis & Mechanical Brogue for Data Application, plus) is a modern, strongly typed mixed structure programming language. By mixed, it can be both a procedural, object-oriented programming language as well as a functional language. LAMBDA+ is a proprietary language developed and owned by the Black Mesa research corporation, and is meant to be utilized for laboratory and other machines on site at The Black Mesa Research Facility located at **[DATA EXPUNGED]**, New Mexico. The language is designed around readability for our less tech-savvy team members. LAMBDA+ is based on Java and Erlang, with the following differences:

1. Containers (packages), experiments (classes), and methods/functions can be "declared" before their block is entered. They are given identifiers, and then once the programmer wishes to enter, they can use the "open/close" block structure for experiments and containers, and the "meth/close" or "func/close" block structure for methods/functions.
2. Distinction between Functions and Methods with separate declarations. Methods are procedures that are associated with a particular experiment/object, while functions are procedures that are not bound to a particular experiment/object.
3. Far more explicit wording and use of keywords, such as the use of "then" within if/else blocks. This is to help improve readability.
4. Only block structures are enclosed in curly braces. Common block structures include open/close, if/else, and loops.
5. Numbers have been simplified to two types: int and real. These two types have the highest precision or size equivalents in Java.
6. Procedural code can be compiled and run on the command line. Functional code, if not used by any procedural code, needs to be run on Lambda+'s custom REPL terminal: G-MAN (GNU-Manipulative Access Networker).

1.1 Genealogy



1.2 "Hello, World!"

```
/>
Dr. W. Breen
15:43:54 September 24, 2002
Example Program
</
container FirstProgram;
open FirstProgram {
    unrestricted experiment Hello;
    open Hello {
        unrestricted static def void begin(String[] args)
        def main {
            println("Hello,_world!");
        }
        close main;
    }
    close Hello;
}
close FirstProgram; /\this can also go on the same line as the '}' in all
   cases, however this may hamper readability
```

1.3 Program structure

The key organizational structure of LAMBDA+ is as follows:

1. All source files must be contained within an Container file structure, similar to Java packages. This is to keep the programs tidy, and prevent any unforeseen consequences.
2. When coding purely procedurally, all code must be contained within an experiment (class). Only methods can be declared within an experiment.
    - Additionally, every program (or container of source files) needs one method defined as `unrestricted static def void begin()`. This method functions similarly to the `main` method in Java. Having `(String[] args)` as the parameters is optional, but recommended.
3. File types are dependent on the type of code contained within the file.
    - Source files that contain procedural code (experiments) are saved in the `.lamb` format.
    - Source files that contain only functional code (no experiments) are saved in the `.lambmod` format (stands for "Lambda Module"). Files saved in this format must also have a module declared to allow importing of functions.
    - Source files with a mix of procedural code and functional code will also utilize the `.lamb` file format. These types of files do not need to have a module declared, however the functions will then only be accessible to classes or other functions within this file.
4. All blocks of code that are not alternation or repetition (meaning if/else, switch, and loop statements are excluded) must begin with an `open <identifier> {` and end with a `} close <identifier>;`.
5. All lines that finish a statement or close a block should end with a semicolon.

1.3.1 Example Program

The following program in Lambda+

```
/>
Dr. G. Freeman
08:58:22 May 16, 2003
Testing of crystal sample acquired from [REDACTED]. This is the purest sample yet.
</
unrestricted container AnomalousMaterials;
open AnomalousMaterials {
    unrestricted experiment CrystalSample;
    open CrystalTest {
        unrestricted static def void begin();
        open begin{
            Scientist gordon := new Scientist("Gordon Freeman", "Theoretical Physics", 100); /\ name, degree, suitHealth

            Crystal zenCrystal := new Crystal("yellow-orange", 10); /\ color, purity

            secret boolean success := gordon.test(zenCrystal);

            if (success)
            then println("Successful!");
            else println("Unforseen Consequences");

        } close begin

        unrestricted experiment Scientist;
        open Scientist {
            secret name;
            secret degree;
            secret suitHealth;

            unrestricted def Scientist(newName: String, newDegree: String, newHealth: int);
            open Scientist {
                name := newName;
                degree := newDegree;
                suitHealth := newHealth;
            } close Scientist;

            unrestricted def boolean test(sample: Crystal);
            open test {
                return (sample.getPurity < 10);
            } close test
        } close Scientist;

        unrestricted experiment Crystal;
        open Crystal {
            secret color;
```

```
46              secret purity;
47
48              unrestricted def Crystal(newColor: String, newPurity: int);
49              open Crystal {
50                  color := newColor;
51                  purity := newPurity;
52              } close Crystal;
53
54              unrestricted def int getPurity();
55              open getPurity {
56                  return purity;
57              } close getPurity;
58          } close Crystal;
59      } close CrystalTest;
60  } close AnomalousMaterials;
```

creates a new experiment called CrystalTest in the AnomalousMaterials container. In the experiment, a Scientist object is created with the name `"Gordon Freeman"`, degree `"Theoretical Physics"`, and suitHealth `100`. A crystal object is then created with the color `"yellow-orange"` and the purity `10`. The Scientist object then tests the Crystal object, which is an internal method to Scientist that checks the purity of a Crystal object and returns true if the crystal purity is <10, and false if $>=10$. This boolean determines the success of the theoretical object. The experiments for the Scientist and Crystal objects are also present in the file.

1.4 Types and Variables

There are two kinds of types in Lambda+: value types and reference types. Variables of value types directly contain their data whereas variables of reference types store references to their data, the latter being known as objects. With reference types, it is possible for two variables to reference the same object and thus possible for operations on one variable to affect the object referenced by the other variable. See Section 3 for details.

1.5 Visibility

There are three types of visibility for programs, classes, methods, and variables (collectively referred to as "items" from now on). They are defined as follows:

- **Unrestricted**: Unrestricted items have no restrictions on their access. Any other item that wishes to use a class, method, or variable is able to, even ones in different containers.
- **Secret**: Secret items are far more restrictive. Secret items can only be seen by the class they are a part of. This does not extend to classes that inherit classes that have secret methods or data.
- **Unlisted**: Unlisted items are similar to Secret items, in that they cannot be accessed outside the class they are present in, by default. However if a class inherits another class that contains unlisted items, the parent class will have access to the unlisted items. **By default, items not given an explicit visibility keyword will be considered Unlisted**.

1.6 Statements Differing from Java and Erlang

Statement	Example
Expression statements	`int i := 1;` `stack.push(i);`

if/else statements	`unrestricted static def void begin(String[] args);`
	`def main {`
	`  if (zenCrystal.isStable())`
	`  then {`
	`  println("Increasing power.");`
	`  photonLaser.increasePower(1);`
	`  }`
	`  else //curly braces are optional for single lines within a block`
	`  photonLaser.shutDown();`
	`}`
	`close main;`

Method declaration & initialization	`unrestricted def int double(input: int);`
	`open double {`
	`  return input * 2;`
	`} close double;`

Function declaration & initialization	`unrestricted fun int square(value: int);`
	`open square {`
	`  return value * value;`
	`} close square;`
	OR
	`unrestricted fun int square(value: int) -> {`
	`  return value * value;`
	`} close square;`

Matrix declaration	`unrestricted static def void begin(String[] args);`
	`open begin {`
	`  Matrix myMatrix1 := [2][4];`
	`  Matrix myMatrix2 := {{1,2,3},{4,5,6},{7,8,9}}`
	`}`

2 Lexical Structure

2.1 Programs

A Lambda+ ***program*** consists of one or more ***source files***. A source file is an ordered sequence of Unicode characters.

Conceptually speaking, a program is compiled using three steps:

1. Transformation, which converts a file from a particular character repertoire and encoding scheme into a sequence of Unicode characters.
2. Lexical analysis, which translates a stream of Unicode input characters into a stream of tokens.
3. Syntactic analysis, which translates the stream of tokens into executable code.
4. Code is then sent through the Code Analyzer AI located in the [REDACTED] for review. Any code that does not meet Black Mesa standards will be discarded and is grounds for a demerit.
   We're not amateurs here, folks. Amateurs work at Aperture.

2.2 Grammars

2.2.1 Lexical grammar where different from Java and Erlang

```
<Assignment operator>    -->  :=
<Mathematical operator>  -->  +  |  *  |  /  |  -
<String concatenation>   -->  ><
<Begin block grouping>   -->  {
<End block grouping>     -->  }
<num>                    -->  [0-9]
<alpha>                  -->  [A-Z]
                         -->  [a-z]
<Implicit func open>     -->  ->
<Single-line comment>    -->  /\
<Delimited comment>      -->  />  ...  <\
```

2.2.2 Syntactic grammar where different from Java and Erlang

```
<container declaration>   --> container <identifier>
<experiment declaration>  --> [<access modifier>] experiment <identifier>
<method declaration>      --> [<access modifier>] def <type> <identifier>
<function declaration>    --> [immutable] [<access modifier>] fun <type> <identifier>
<module declaration>      --> export module(<identifier>)
<module call>             --> <module identifier>:<function identifier>
<parameter list>          --> <parameter> <parameter list>
                          --> <parameter>
<parameter>               --> <identifier> : <type>
<block initialization>    --> open <identifier> {
<block closure>           --> } close <identifier>
<negative number>         --> ~<num>
```

2.3 Lexical Analysis

2.3.1 Comments

Two forms of comments are supported: single-line comments and delimited comments. ***Single-line comments*** start with the characters /\ and extend to the end of the source line. Its shape is inspired by the capital lambda. ***Delimited comments*** start with the characters /> and end with the characters <\. This is due to the fact that the arrows are closer to the / key and I hate holding right shift (don't put that in the document - *Breen*). Delimited comments may span multiple lines. Comments CAN nest, as this can be useful for commenting out code that may contain block comments already, which can aid in debugging.

2.4 Tokens

There are several kinds of tokens: identifiers, keywords, literals, operators, and punctuators. White space and comments are not tokens, though they act as separators for tokens where needed.

tokens:

identifier

keyword

integer-literal

real-literal

character-literal

string-literal

boolean-literal

operator-or-punctuator

2.4.1 Keywords different from Java or Erlang

New Keywords	**Removed Keywords**
unrestricted	public
secret	private
unlisted	[no access modifier]*
open	
close	
immutable	

*this isn't a keyword, but Java has a special type of visibility from having no modifier. Unlisted makes the implementation explicit while still allowing the modifier keyword to be optional.

3 Type System

Lambda+ uses a strongly static typed system. This is to improve readability and minimize the chance of errors occurring through user error or implicit conversions (remember: your mistype could cost someone's life!). Strong typing means that type errors are caught and expressed to the programmer during compilation. Static typing means early binding compile-time type checking.

3.1 Type Rules

Arithmetic:

$S \vdash e_1 : num$
$S \vdash e_2 : num$
num is a real or an int
$S \vdash e_1 + e_2$: num

OR

$S \vdash e_1 : num$
$S \vdash e_2 : num$
num is a real or an int
$S \vdash e_1 - e_2$: num

OR

$S \vdash e_1 : num$
$S \vdash e_2 : num$
num is a real or an int
$S \vdash e_1 * e_2$: num

OR

$S \vdash e_1 : num$
$S \vdash e_2 : num$
num is a real or an int
$S \vdash e_1/e_2$: num

Comparisons:

$S \vdash e_1 : T$
$S \vdash e_2 : T$
T is a primitive type
$S \vdash e_1 = e_2$: boolean

OR

$S \vdash e_1 : T$
$S \vdash e_2 : T$
T is a primitive type
$S \vdash e_1 != e_2$: boolean

OR

$S \vdash e_1 : T$
$S \vdash e_2 : T$
T is a primitive type
$S \vdash e_1 > e_2$: boolean

OR

$S \vdash e_1 : T$
$S \vdash e_2 : T$
T is a primitive type
$S \vdash e_1 < e_2$: boolean

String Concatenation:

$S \vdash e_1 : String$
$S \vdash e_2 : String$
$S \vdash e_1 >< e_2 : String$

OR

$S \vdash e_1 : String$
$S \vdash e_2 : T$
T is a primitive type
$S \vdash e_1 >< e_2 : String$

OR

$S \vdash e_1 : T$
$S \vdash e_2 : String$
T is a primitive type
$S \vdash e_1 >< e_2 : String$

Assignment:

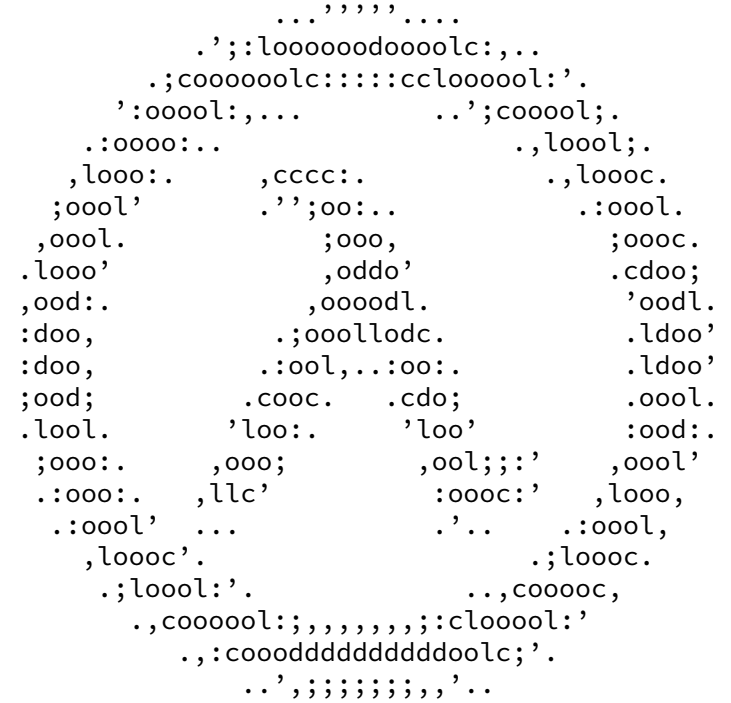
$S \vdash e_1 : T$
$S \vdash e_2 : T$
T is a primitive type
$S \vdash e_1 := e_2$: T

3.2 Value types (differing from Java or Erlang)

- **alpha** - a subset of char that only includes the Latin alphabet in both cases (A-Z & a-z). Utilizes a custom range different from ASCII for conversion (A-Z := 0-25; a-z := 26-51), and includes methods for casting between an int and an alpha as well as an alpha and a char.
- **int** - now the ONLY type of integer value. Long and Short type integers no longer exist. (We bought the whole memory, we're gonna use the whole memory!)
- **real** - now the ONLY type of floating point number, replacing double. Float no longer exists. (See previous statement on memory concerns)

3.3 Reference types (differing from Java or Erlang)

- **Matrix** - a new, explicit implementation of the double array. If a triple or larger multi-dimension array, they must be nested manually.
- **functional object** - an implementation of experiments (classes) in a functional manner that allows objects to be utilized functionally. An object constructor given the `immutable` property becomes functional: any object created from that constructor cannot have its values changed, only read.
- **module functions** - modules are specifically groups of functions that have been imported from a `.lambmod`. When a module file is imported, the functions contained within that file are committed to memory and given pointers. Whether a function is exported is determined by its accessibility modifer (only unrestricted functions are exported). To access these functions, the pointers are called using the syntactical format `<module identifier>:<function identifier>`, followed by any parameters the function may need enclosed in parentheses. Because these imported module functions are being called with their pointers, the pointers can also be passed into other functions. **(Warning! This can be dangerous. Use with caution. Black Mesa is not responsible for any world-ending events caused by a lost pointer)**



4 Example Programs

4.1 Caesar Cipher (Encrypt & Decrypt) [Procedural]

```
/>
A Caesar Cipher program that enables both encryption and decryption using a
string and a shift amount. This particular version is coded in a procedural
manner.
</
unrestricted container Ciphers;
open Ciphers {
    unrestricted experiment CaesarCipher;
    open CaesarCipher {


        unrestricted static def void begin();
        open begin() {
            /\ Data to be used in the cipher.
            secret String inputStr := "aBcDeFgHiJkLmNoPqRsTuVxYz";
            secret int shift := 3;

            /\ Code execution.
            String encrypted := encrypt(inputStr, shift);
            println("Encrypted: " >< encrypted);
            String decrypted := decrypt(encrypted, shift);
            println("Decrypted: " >< decrypted);
        }
        close begin;

        unrestricted def String encrypt(str: String, shift: int);
        open encrypt{
            alpha c;
            int ascii;
            String result := "";
            for (int i := 0; i < str.length(); i++) {
                if (str.charAt(i).isAlpha())
                then {
                    c := str.charAt(i);
                    ascii := c.toInt + shift;
                    if (ascii >= 0 && <= 25)
                    then {
                        if (ascii > 25)
                            then ascii := ascii - 26;
                        else if (ascii < 0)
                            then ascii := ascii + 26;
                    } else if (ascii >= 26 && <= 51)
                    then {
                        if (ascii > 51)
                            then ascii := ascii - 26;
                        else if (ascii < 26)
                            then ascii := ascii + 26;
                    }
```

```
                    c = ascii.toAlpha;
                    result := result >< c;
                }
            }
            return result;
        }
        close encrypt;

        unrestricted def String decrypt(str: String, shift: int);
        open decrypt{
            String result := encrypt(str, ~(shift));
            return result;
        }
        close decrypt;
    }
    close CaesarCipher;
}
close Ciphers;
```

4.2 Caesar Cipher (Encrypt & Decrypt) [Functional]

```
/>
A Caesar Cipher program that enables both encryption and decryption using a
string and a shift amount. This particular version is coded in a functional
manner.
</
unrestricted container FunctionalCiphers;
open FunctionalCiphers {
    export module(Caesar);

    unrestricted fun void begin;

    fun alpha shifter(c: char, shift: int) -> {
        if (c.isAlpha())
        then {
            const int ascii := c.toInt + shift;
            if (ascii >= 0 && <= 25)
            then {
                if (ascii > 25)
                    then return (ascii - 26).toAlpha;
                else if (ascii < 0)
                    then return (ascii + 26).toAlpha;
            } else if (ascii >= 26 && <= 51)
            then {
                if (ascii > 51)
                    then return (ascii - 26).toAlpha;
                else if (ascii < 26)
                    then return (ascii + 26).toAlpha;
            }
            return ascii.toAlpha;
        } else
```

```
            return c;
    }

    unrestricted fun String encrypt(input: String, shift: int) -> {
        return map(λ(X) -> {shifter(X, shift)}, input);
    } close encrypt;

    unrestricted fun String decrypt(input: String, shift: int) -> {
        return encrypt(input, ~(shift));
    }close decrypt;


    open begin { /\ initialization long after declaration
        const String inputStr = "aBcDeFgHiJkLmNoPqRsTuVxYz";
        const int shift = 3;

        const String encrypted = encrypt(inputStr, shift);
        println("Encrypted: " >< encrypted);

        println("Decrypted: " >< decrypt(encrypted, shift));
    } close begin;
}close FunctionalCiphers;
```

4.3 Factorial

```
unrestricted container MiscMath {
    export module(MathFuncs);

    unrestricted fun int factorial(val: int) -> {
        if (val = 0)
            then return 1;

        return val * factorial(val - 1);
    } close factorial;
} close MiscMath;
```

4.4 Insertion Sort

```
/>
An insertion sort implementation in Lambda+
</
unrestricted Sorts;
open Sorts {
    unrestricted experiment InsertionSort;
    open InsertionSort {
        unrestricted def void sort(int arr[]);
        begin sort {
            int n := arr.length;
            for (int i := 1; i < n; i := i + 1) {
                int key := arr[i];
                int j := i -1;

                while (j >= 0 && arr[j] > key) {
                    arr[j+1] := arr[j];
                    j := j - 1;
                }
                arr[j + 1] = key;
            }
        }
    } close InsertionSort;
} close Sorts;
```

4.5 QuickSort

```
/>
A quicksort implementation in Lambda+.
</
unrestricted container Sorts;
open Sorts {
    unrestricted experiment QuickSort;
    open RecursiveSort {
        unrestricted def void quickSort(arr: int[], first: int, last: int);
        open quickSort {
            if (first < last)
            then {
                int pivot := partition(arr, first, last);
                quickSort(arr, first, pivot-1);
                quickSort(arr, pivot+1, last);
            }
        } close quickSort;

        unrestricted def int partition(arr: String[], first: int, last: int);
        open partition {
            String pivotValue := arr[last];
            int pivotLocation := first - 1;
            int temp;
            for (int i:= first; i< last; i := i + 1) {
                if (arr[i] <)
                then {
                    pivotLocation := pivotLocation + 1;
                    temp := arr[pivotLocation];
                    arr[pivotLocation] := arr[i];
                    arr[i] := temp;
                }
            }
            temp := arr[pivotLocation + 1];
            arr[pivotLocation + 1] := arr[last];
            arr[last] := temp;

            return pivotLocation + 1;
        } close partition;
    } close RecursiveSort;
} close Sorts;
```

4.6 Stack

```
/>
A class representing a Stack object. This class assumes there is a valid Node
class within the same package. Also showcases the ability declare methods
before initializing them. This is nice for times where you want to lay out
all the methods a class will contain before you start implementation, sort
of like a roadmap.
</
```

```
unrestricted container DataStructs;
open DataStructs {
    unrestricted experiment Stack {
        secret Node myHead = null;

        /\Methods declared up here...
        unrestricted def Stack();

        unrestricted Node getMyHead();
        unrestricted void setMyHead();

        unrestricted void push(Node newNode);
        unrestricted Node pop();
        public boolean isEmpty();

        /\...and initialized down here.
        open Stack {} close Stack; /\empty constructor

        open getMyHead {
            return myHead;
        } close getMyHead;

        open setMyHead {
            this.myHead := newHead;
        } close setMyHead;

        open push {
            if (this.isEmpty)  /\functions with no parameters don't need
               parentheses.
            then this.setMyHead(newNode);
            else {
                newNode.setMyNext(this.getMyHead);
                this.setMyHead(newNode);
            }
        } close push;

        open pop {
            Node temp = this.getMyHead;
            this.setMyHead(temp.getMyNext);
            return temp;
        } close pop

        open isEmpty {
            if (this.getMyHead = null)
            then return true;
            else return false;
        } close isEmpty
    } close Stack;
} close DataStructs;
```

4.7 Lambda Functions

```
/>
Some examples of anonymous functions in Lamda+
</
unrestricted container Lambda;
open Lambda {
    export module(Combinators);

    unrestricted fun I -> {
        λ(x) -> {x}; /\ A basic λ function.
    } close I;

    unrestricted fun K -> {
        λ(x,y) -> {x}; /\ One way to call a λ function with multiple
            parameters.
    } close K;

    unrestricted fun KPrime -> {
        λ(x) -> {λ(y) -> {y}} /\ Another way to call a λ function with
            multiple parameters.
    } close KPrime;

    unrestricted fun S -> {
        λ(x,y,z) -> {
            xz(yz) /\ Can be expressed on multiple lines if needed.
        }
    } close S;

    />
    If functions are placed next to each other without a semicolon break or
    parentheses, the compiler will attempt to apply them (left associative).
    This ONLY works with functions, not methods.
    </
    println(S K K); /\ Should print "λ(x) -> {x}"

} close Lambda;
```

5 Appendix

5.1 Why Functional?

One might wonder, "why functional? Can't Lambda+ do everything it needs in a procedural manner?". And you would be right. However, there are times that functional programs may be beneficial. The greatest example being, modules. Since purely functional programs are saved in `.lambmod` files and exported as modules, they become almost lightweight libraries to be utilized within Lambda+. Say you have a project that has many source files that require very specialized functions by many different classes. You could use a utility class, but that could be bulky and memory intensive. A much smaller, cleaner, and simpler option is to use a module, which are lighter and faster than their OOP counterparts.

5.2 Method Overloading?

You may have noticed that with the way methods are declared and then initialized, there can be gaps between what a method is called and what it actually contains. Does this mean that methods cannot be overloaded? It does not. When a method is declared with the same identifier as another already declared method, the two methods are entered into a sort of Queue structure. Once initialization of those methods begin with an `open` statement, the first method is dequeued and initialized with the code that follows. This means that initialization of overloaded methods **must always** be in the same order that they were declared.