C'THULU

Language Specification
Version 0'03

1. Introduction

C'thulu is a list-based functional language that borrows heavily from LISP , but also incorporates functionality from the Object-Oriented language such as dynamic typing, Python. C'thulu differs from Python and Lisp in that:

1. It eliminates the excessive amounts of parentheses used in lisp
2. Sigils (objects) are defined as being sophisticated forms of lists
3. Nothing in the language is case sensative, allowing programmers to not worry about a wrongly capitalized variable.
4. Everything is termed in the manner of the Lovecraftian tongue.

1.1 Hello world as an anonymous function:

```
cult of c'thulu
the stars are right
        shoggoth evoke("Hello World")
Summon C’thulu
>>Hello World
```

1.2 Program structure

The key organizational concepts in C'thulu are as follows:

1. Uses whitespace like in python, thus indentations must be used after the headings are completed
2. Programs that rely on cults must state the cults involved via a "Cult of" command
3. Programs may only begin execution if the stars are right and must finish before the rise of our tenticalled master.

This example:

```
Cult of C'thulu
        supernatural cult acolyte
                'title
                'purpose

                supernatural acolyte(name specialty)
                        mutate alchemist'title name
                        mutate alchemist'purpose specialty

                supernatural ritual call'cthulu()
```

```
        if alchemist'purpose = demonology
                acolyte'call'forth(c'thulu)
        else
                banish acolyte

supernatural ritual call'forth(elder)
        if elder = "C'thulu"
                evoke("Rise from the Depths O' Tenticalled Master!")
                summon elder
        else
                summon "The Stars are Not Right"
```

creates a class, called acolyte with two data members, title and purpose. Two functions, call'c'thulu and call'forth. The acolyte's data members are accessed individually by acolyte'title notation. Function calls are done in the same manner using rituals: acolyte'call'c'thulu.

1.3 Types and variables

C'thulu follows Lisp's example, using only list objects which may be empty or contain any set of values. Lists are passed by reference while their individual constituents are passed by value if called.

1.4 Statements Differing from Python and Lisp

Statement Example

Expression statement	Conjure x mutate x (1 2 3 4) Conjoin(x 23 42) Evoke(x) >>(1 2 3 4 23 42)
Looping	conjure x mutate x (1 2 3 4) Evoke(curse(elder x+1 spawn x)) >>(2 3 4 5)

1.5 Classes and objects

New cults (classes) are created using cult declarations.
The following is a declaration of a simple cult named `Point`:

```
public cult Point
      'x
      'y
      public Point(x y)
            mutate point'x x
            mutate point'y y
```

Instances of classes are created using the `conjure sigil` operators. Conjure typically just creates a new list, but when combined with sigil, a special "object" type is created which may have its individual properties manipulated easier.

```
conjure x sigil Point(4 2)
```

The memory occupied by an object is automatically reclaimed when an object goes out of scope. However it is good practice in C'thulu to always banish unused or unreferenced conjurations

1.5.1 Accessibility

Each member of a class has an associated accessibility, which controls the regions of program text that are able to access the member. There are four possible forms of accessibility. These are summarized in the following table.

Accessibility Meaning

Public	Access not limited
Private	Access limited to this class
Protected	Access limited to this class or classes derived from this class
Supernatural	Access is limited to other supernatural rituals or cults.

1.5.2 Fields

A property is a variable that is associated with a class or with an instance of a cult.

```
public cult Color
      conjure Black sigil Color(0, 0, 0)
      conjure White sigil Color(255 255 255)
      'r
      'g
      'b
      public Color(r g b)
            mutate color'r  r
            mutate color'g  g
            mutate color'b  b
```

1.5.3 Methods

A ***ritual*** is a member that implements a computation or action that can be performed by an object or class. An anonymous slave method is a Shoggoth, named for the slave race of the old ones, these are essentially one-shot rituals.

1.5.3.1 Constructors

C'thulu constructs new lists from the conjure command. Conjure x will create a new list denoted by the variable x. If sigil is appended, then it becomes a special class object with properties of that class that may be manipulated

1.5.3.2 Properties

Properties are a natural extension of fields. Both are named members with associated types, and the syntax for accessing fields and properties is the same. However, unlike fields, properties do not denote storage locations. Instead, properties have ***accessors*** that specify the statements to be executed when their values are read or
written.

1.5.3.3 Events

A **manipulation** is a member that enables a class or object to provide notifications.
Clients react to events through ***a manipulator***. Event handlers are attached using the `+=` operator and removed
using the `-=` operator. The following example attaches an event handler to
the `Changed` event of a `List<string>`.

```
Public cult Test
conjure changeCount


      ritual ListChanged(sender Arg)
      mutate changeCount  changeCount + 1
```

```
Cult of Cthulu
Cult of Test
The Stars are Right

public Cult Main
conjure names
names'Changed += new Manipulator(ListChanged);
names'Add("Christine")
names.Add("Luis")
names.Add("Anthony")
Evoke(changeCount)
```

1.6 Arrays

An ***array*** is a data structure that contains a number of variables that are accessed through computed indices. The variables contained in an array, also called the ***elements*** of the array, are all of the same type, and this type is called the ***element type*** of the array. The following example allocates a one-dimensional, a two-dimensional, and a three-dimensional array.

```
make'array(a 10)
make'array(a2 10 5)
make'array(a3 10 5 2)
```

The `a1` array contains 10 elements, the `a2` array contains 50 (10 × 5) elements, and the `a3` array contains 100 elements.

2. Lexical structure

2.1 Programs

A C'thulu ***program*** consists of one or more ***source files***. A source file is an ordered sequence of (probably 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.

2.2 Grammars

This specification presents the syntax of the C'thulu programming language where it differs from Python and Lisp.

2.2.1 Lexical grammar where different from Python and Lisp

Strings are akin to the Lisp language in that they are lists of characters, strings are treated as literals and must use specific operators to print or manipulate. otherwise C'thulu displays the literal. For example:

```
supernatural sigil student
  'first'name
  'last'name
  'gpa

conjure student stu   ---creates a new student object with empty lists for its properties
  mutate'stu'first'name "H.P."
  mutate'stu'last'name "Lovecraft"
  mutate'stu'gpa "3.4"

Summon stu
>>("H.P." "Lovecraft" "3.4")
```

```
Evoke'String(stu)
>>"H.P. Lovecraft 3.4"
Summon stu'first'name
>>(H . P .)
Evoke'String(stu'first'name)
>>"H.P."
```

by summoning the variable stu, it returns the list in its entirety. However, by using the sigil's (object's) properties, and using the Evoke'String() operator, a string version of that property is sent to the console. The same may be accomplished with the entire list by using Evoke'String(stu).

2.2.2 Syntactic ("parse") grammar where different from Python and Lisp

If-Then

If *boolean-exp* then *statement*

OR

If *boolean-exp*

statement

If-Else

If *boolean-exp*

statement

Else

statement

If-Else-If

If *boolean-exp*

statement

Else If *boolean-exp*

statement

Else

statement

While

whilst *boolean-exp*

statements

For

for *var* whilst *boolean-exp* do *parameter*

statements

2.2.3 Grammar notation

The lexical and syntactic grammars are presented using **BNF *grammar productions***. Each grammar production defines a non-terminal symbol and the possible expansions of that non-terminal symbol into sequences of nonterminal or terminal symbols. For example, the production:

while-statement:

`while` *boolean-expression*

embedded-statement

defines a *while-statement* to consist of the token `while`, followed by a whitespace and reading in the boolean statement (which is identified by a look-ahead that finds what is beyond the next whitespace)

When there is more than one possible expansion of a non-terminal symbol, the alternatives are listed on separate lines. For example, the production:
statement-list:
statement
statement-list statement
defines a *statement-list* to either consist of a *statement* or consist of a *statement-list* followed by a *statement*. In other words, the definition is recursive and specifies that a statement list consists of one or more statements.

2.3 Lexical analysis

2.3.1 Line terminators

Line terminators divide the characters of a C'thulu source file into lines.
new-line:
Carriage return character (U+000D)
Line feed character (U+000A)
Carriage return character (U+000D) followed by line feed character (U+000A)
Next line character (U+0085)
Line separator character (U+2028)
Paragraph separator character (U+2029)

2.3.2 Comments

Two forms of comments are supported: single-line comments and delimited comments. ***Single-line comments*** start with the character ` and extend to the end of the source line. ***Delimited comments*** start with the characters `! and end with the characters !`. Delimited comments may span multiple lines. Comments do not nest.

2.3.3 White space

White space is defined as any character with Unicode class Zs (which includes the space character) as well as the horizontal tab character, the vertical tab character, and the form feed character.
whitespace:
Any character with Unicode class Zs
Horizontal tab character (U+0009)
Vertical tab character (U+000B)
Form feed character (U+000C)

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.
token:
identifier
keyword
integer-literal: *digit-list*
real-literal: *digit-list . digit-list*
character-literal: *character*
string-literal: *" character-list "*
"character-list SPACE character-list"
operator-or-punctuator:
*Mathematical: +, - , mod, *, /, ^, +=, -=, =*
Logical: =, <=, >=, ~,
Punctuation: ( ) [ ] . " '

2.4.1 Keywords different from Python or Lisp

A ***keyword*** is an identifier-like sequence of characters that is reserved, and cannot be used as an identifier.
New keywords: one of
`supernatural, cult, ritual, conjure, summon, invoke, evoke, manipulator, manipulation, shoggoth, mutate, banish, conjoin, sigil, cast, spawn, elder, curse, whilst`
Removed keywords:
`do, goto, cdr, car, cons, int, float, while, print`

3. Basic concepts

3.1 Application Startup

Application startup occurs when the execution environment calls a designated method, which is referred to as the application's entry point. This entry point method is always named `The Stars are Right`. That is all. This language can only work if the stars are right, otherwise it is not the time to summon C'thulu's powerful application-building lists.

3.2 Application termination

Application termination returns control to the execution environment. The application will always end with ***Summon C'thulu***, the rise of our tenticalled Lord. The stars are right and therefore the end has come. Lord C'thulu shall rise up and end all, so the termination of your application is taken care of.

The only exception to this rule is if the user has made an error when working with the program. Bad data is unacceptable and as such the application will terminate, banishing all tainted variables.

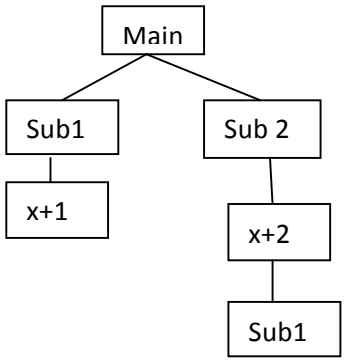
3.3 Scope

Like most implementations of Lisp, C'thulu uses dynamic scope, thus it relies on the order of the call stack to determine a variable's value. For example:

```
conjure x mutate 0

supernatural ritual sub1(var)
        mutate x x+1
supernatural ritual sub2(var)
        mutate x x+2
        summon sub1(x)

sub2(x)
>> 3
```



Since x is 0, as sub2 is called, it is mutated to 2, then sub1 is called, the value is mutated to 3.

3.4 Automatic memory management

Alan++ employs automatic memory management, which frees developers from manually allocating and freeing the memory occupied by objects. Automatic memory management policies are implemented by ***banishment***. It differs from Python and Lisp in the following ways:

1. A banish command may be called on any reference variable at anytime to destroy it as well as the pointers that referenced it.
2. If two pointers pointed to the same memory, both would be destroyed, rendering any aliasing inert.
3. However, C'thulu does not entrust humans to always do their banishments and will take care of it as a variable travels out of scope or at termination.

4. Types

C'thulu types are divided into two main categories: ***Value types*** and ***Reference types***.

4.1 Value types

In most languages, the types of int, float, boolean, or character would be placed here. While C'thulu recognizes these types, it does so dynamically for the purposes of computation and does not allow definition of these types. C'thulu works strictly with lists of values, thus a mutation of a variable will be converted to a list.

4.2 Reference types

The list is the most basic form of the reference type, though special lists may be formed into arrays or vectors. One specific type of list, denoted by the sigil keyword, is an "object" list, which contains properties as defined by object-oriented languages (similar to a struct in C).

5. Variables

Variables represent storage locations. Every variable has a type that determines what values can be stored in the variable. Lists may contain different sets of values, all dynamically typed akin to Python. However, it should be noted that arrays must contain the same variable type.

5.1 Variable categories

Integer:

Any value that is a whole number.
mutate x 0 => x = (0)

Float:

Any number that represents a fractional value that lies between two whole numbers.
mutate x 3.14 => x = ([3.14])

Character:

Any single character that may be represented by the Unicode table, be it a symbol, integer, or letter (with the exception of whitespace/newline characters) treated as a literal unless otherwise specified.
mutate x "c" => x = (c)

String:

Any combination of characters and whitespace characters (terminated by a newline character), treated as a literal, unless otherwise specified.
mutate x "Lord C'thulu waits in R'yleh" => (L o r d C ' t h u l u w a i t s i n R ' y l e h)

Boolean

A value that can be used to evaluated truth or falsity

mutate x true => (TRUE)
mutate x false => (NULL)

6. Parameter Passing

6.1 Method

C'thulu strictly uses Pass-by-Reference parameter passing (InOut) to pass a list's memory addresses to avoid copying the entirety of lists. This way an element may be passed and operated on as necessary given the necessary permissions.

6.2 Examples

```
conjure x sigil student("H.P." "Lovecraft" 3.14)
conjure y mutate 0.5
shoggoth (summon x'gpa + y)
>> 3.64
```

Local Variable x = ("HP" "Lovecraft" 3.14)
Parameters: x'gpa, y
Return Address

7. Conversions

A ***conversion*** enables an expression to be treated as being of a particluar type.

7.1 Implicit conversions

Most conversions in C'thulu are hands-off and are taken care of by the typing system. If a variable's behavior matches up with the function calls, all is well. Any misuse of varaibles or type-error detections are passed on to the user to be corrected via an explicit conversion if absolutely necessary.

7.2 Explicit conversions (Mutate some literals!)

The best example of an explicit type conversion in C'thulu is with the String-literals. Since all strings are treated as literals and will not be changed until explicitly stated by the user via a 'String function, this leaves input as pure as it was given. The list will be what is evaluated and operated on, however the string itself will only be outputted as a string if told specifically by the user. Otherwise, C'thulu recognizes the string as a list of characters and nothing more.

8. Statements

Examples of statements in C'thulu that differ from Python and Lisp, as noted in the introduction one:

1. Variables/lists are created by the conjure command. Objects are denoted by the sigil keyword.
2. C'thulu uses whitespace termination and indentation rather than slews of parentheses or braces.
3. Garbage collection is either manual or automatic.
4. C'thulu strictly operates using lists and objects, forming a unique functionally object-based language.
5. Case-sensitivity is eliminated, ridding users of pesky capitalization-errors within the code throwing errors.

9. Example Programs

1. Encryption / Decryption example:

```
Cult of C'thulu
The Stars are Right
`Here is the encryption function
public ritual encrypt(str num)
  invoke str
  evoke'String(tostring(shiftup (ascii str) num))

` And here is the decryption function
public ritual decrypt(str num)
   invoke str
   evoke'String(tostring(shiftdown (ascii str) num))

`function takes a list of characters and converts them into their numeric representations

private ritual ascii(str)
   `conditional: if the end of the list is reached, eval nil, return nothing
    if str'length = 0
            summon null

    `else: take the head of the list, convert it to int, and return the new list
    else
      curse (char'code(elder str) ascii (spawn str))


`used for encryption, shifts the number forewards through the ASCII table
`up to 126 and then loops back to 33

private ritual shiftup (str num)
  if str'length = 0
            summon null
  elseif (elder str + num) > 126
        `if the number + the current position is > 126, mod the total shift by 126
        `then use the smalled value to move up the difference from 32

              conjure shifted'val mutate (elder str + num) mod 126
                curse (shifted'val + 32 shiftup (spawn str) num)
  else
       `Otherwise, just move through the list and encrypt by the value given
            curse ((elder str) + num  shiftup((spawn str) num)


`used for decryption, behaves the opposite of shiftup
private ritual shiftdown (str num)
  if str'length = 0
     summon null
  else if (elder str) - num < 33
     conjure shifted'val mutate (elder str) - 33 - num
     curse (shifted'val - 127 shiftdown((spawn str) num)))
  else
     curse ((elder str) - num  shiftdown((spawn str) num)))
```

C'thulu Language Specification

```
`this is used to take the ascii ints and convert them into into chars -> string

private ritual tostring (str)
  if str'length = 0
     summon null
  else
     curse (code'char(elder str) tostring (spawn str))
summon C'thulu
```

2. Basic class creation

```
Public Cult Student
‘first
‘last
‘gpa
        Public sigil student(Name’first Name’last gpa)
                Mutate Student’first Name’first
                Mutate Student’last Name’last
                Mutate Student’gpa gpa

        Public sigil student()
                Mutate Student’first “”
                Mutate Student’last “”
                Mutate student’gpa 0

        Public ritual getStudent()
                Evoke’String(“Student Name:” Student’first Student’last)
                Evoke’String(“GPA:” student’gpa)

        Public Ritual setStudent(Name’first Name’last gpa)
                Mutate Student’first Name’first
                Mutate Student’last Name’last
                Mutate Student’gpa gpa
```

3. Simple Imparative Loops

```
Cult of C’thulu
The stars are right

Conjure x mutate 0
Conjure lista mutate x
Evoke lista

While lista’length < 10
        Mutate x x+1
        Lista’conjoin(x)
        Evoke lista
```

C’thulu Language Specification

Summon C’thulu

>>(0)
>> (0 1)
>>(0 1 2)
>>(0 1 2 3)
>>….
>> (0 1 2 3 4 5 6 7 8 9)

4. Looping via recursion

Cult of C’thulu
The Stars are Right

conjure lista mutate (0 1 2 3 4 5 6 7 8 9)

Shoggoth(evoke (curse ((elder lista ^ 2) spawn lista))

Summon C’thulu
>>(0 1 4 9 16 25 36 49 64 81)

10. Conclusion

Obviously any spawn of C'thulu is superior than the creations of any other lesser being that has come into existence since he was condemned beneath the sea by the stars that torment him and his followers so. This language, based on a god's likeness is no exception.

The concept behind the C'thulu language was to take two extremely powerful languages, Lisp and Python, and attempt to blend their features into a hybrid language that gives the same flexibility of an object-oriented language like Python, with a strictly functional language like Lisp.

Lisp is a notoriously powerful language, yet it is avoided by most programmers just due to its sheer complexity, not to mention all of those nasty parentheses. So the idea was to implement some of the more robust features of an object-oriented language as a crutch to wean programmers into functional programming which may benefit from, but not rely on the object-oriented paradigm.

My main reason for choosing Python is it is simple, direct, and easy to read. Combine that with the power and depth of a language like Lisp and you have a language fit to rival the most diabolical being to have ever slithered into our dimension.

After tidying up some of Lisp's functions (and giving them some theatrical flaire), all that needed to be done was compromise the scope and typing methods of the two languages to give it a more reliable and hands-off feeling that let the programmer just fill in the blanks and let the compiler deal with the type-

conversions. As it is, within functional programming the programmer should not rely on what the variable "should be", but rather should know that the program should not rely on something that could be so trivially broken. Thus it institutes a very firm reinforcement of good programming habits, along with the reward that, upon the successful completion of a program, our Lord and Master C'thulu shall be freed and the programmer will not be banished to the inter-dimensional void where the other garbage ends up.