# The Lucida Programming Language

Originated by Luc

Inspired by SICP

Motivated by Meling

# 1 What is Lucida?

Lucida is an scheme-like pure functional programming language built on CLR.

To be honest, it all comes by a sudden inspiration. **One day coding + Half day debugging + 4 Colas => Lucida**, so don't take it seriously.

Lucida could be a teaching language for those who did not program before, or did not use functional programming before.

Lucida use a **RPN(Reversed Polish Notation)**, this idea is borrowed from scheme/lisp. This may make some people very uncomfortable, but this also allow me build a tokenizer and a parser in 2 hours. Anyway, I think you could get used to it.

I would be glad to discuss with you, but please don't teach me lessons like C# is slower than C or CLR is fucked up, only if you could build another language by machine language just use 0/1 button in two days.

If you have any unstupid questions, contact me @ ***lunageek@gmail.com***.

# 2 Basic Type

## 2.1 Number

In order to use a negative number, you need to use - unary operator.

For example:

```
1 => 1
2 => 2
(+ 15) => 15
(- 20) => -20
```

## 2.2 Double

Use Double type if you need more than integer.

For example:

```
1.01 => 1.01
(- 2.5) => 2.5
(+ 3.5) => 3.5
```

## 2.3 Char

Use atoi and itoa to make conversion between digital char and digit.

For example:

```
'a' => 'a'
(itoa 3) => '3'
(atoi '3') => 3
```

# 2.4 Bool

Use Bool type in a condition expression:

For example:

```
true => True
(not true) => False
(< 1 2) => True
(= 'a' 'b') => False
(or true false false) => True
```

# 2.5 List

List could contain elements from different type.

Aslo, List could contain **nested** Lists.

Declare a list by using `list` keyword.

For example:

```
(list 1 2 3) => [1, 2, 3]
(list 1 'a' true) => [1, 'a', True]
(list (list 3 4) (list 1 2)) => [[3, 4], [1, 2]]
(list ) => []
```

# 2.6 String

String are made up of characters, but you could not apply list operation on String directly.

However, you could use `stol` to listify a string, and use `ltos` to stringify a list of characters.

So all list operation could be applied on String as well.

For example:

```
"123" => "123"
(stol "123") => ['1', '2', '3']
(ltos (list 'a' 'b' 'c')) => "abc"
(ltos (reverse (stol "lucida"))) => "adicul"
```

You could use `elem` to get character on a specified index of String.

And test String equality by `=` operator.

For example:

```
(= "abc" "abc") => True
(elem 3 "abcde") => 'd'
(map atoi (stol "1234")) => [1, 2, 3, 4]
```

# 2.7 Function

You could define a function like this:

```
(def (func-name para1 ... paraN) (func-body ...))
```

Then call a function like this:

```
(func-name arg1 ... argN)
```

Some advanced ways of using functions would be talked about later.

For example:

```
(def (add x y) (+ x y)) => Function(x, y)
(add 1 3) => 4
(add (list 1 2) (list 3)) => [1, 2, 3]
(def (sqr x) (* x x)) => Function(x)
(add (sqr 3) (sqr 4)) => 25
```

# 2.8 Named value

You could name instances from different type.

And functions could be assigned directly as well.

For example:

```
(def x 4)
x => 4

(def lst (list 1 2 3))
lst => [1, 2, 3]

(def lst1 (+ lst (list x)))
lst1 => [1, 2, 3, 4]

(def (f x) (+ x 1))
f => Function(x)
(f 3) => 4
(def g f)
g => Function(x)
(g 3) => 4
```

But you could not assign a named value twice.

For example:

```
(def x 4)
x => 4
(def x 5)
   => Variable x has already been declared
```

If you really need to "change" the value of a named variable. Do it like this in console

For example:

```
(def x 4)
x => 4
:del x => Variable x was deleted from current scope
(def x 5)
x => 5
```

# 3, Operators

## 3.1, Arithmetic Operator

Notice:

`+` operator could be used on List and Number type.

`-` operator is both unary and binary

Number of arguments on `+-*/` could be arbitrary **(but not zero**!)

Number and Double type could be used together, but the result is always the Double

For example:

```
(+ 3 4) => 7
(- 3) => -3
(* 1 2 3 4) => 24
(/ 6 3 2) => 1
(+ (list 1 2) (list 3 4)) => [1, 2, 3, 4]
(+ 1 0.0) => 1.0
```

## 3.2, Comparision Operator

Comparision operator could work on two Number, Char and Double, then return a Bool value.

For example:

```
(> 3 2) => True
(<= 4 3) => False
(= "abc" "abc") => True
(= 3 (+ 2 1)) => True
```

# 3.3, Logical Operator

In order to get more clarity, Lucida use `and`, `or`, `not` instead of `&&`, `||`, `!` which were used in many mainstream programming languages.

Notice:

`and`, `or` operator is **short-circuit** evaluated, it would return **asap**.

Number of arguments on and, or operator could be arbitrary (but **not zero**!)

Number of argument on not operator should be **EXACTLY** one.

For example:

```
(and true (= (/ 1 0) 1)) => Oops, an DivideByZero error
(or true (= (/ 1 0) 1)) => True (The latter expression won't be
evaluated due to short-circuit evaluation)
(not true) => False
(and (not false) (or false true) ) => True
```

# 3.4 Special eVil Operator

I designed a set of operators for common list operations. This would be talked about later. But I would remind you here, you don't want to use these only if you want to write some geeky but brain-fucking code.

For example, a factorial function could be written like this:

```
(def (_ n) ($ mul 1 (~ 1 n)))
```

# 4, Control Flow

## 4.1, Branch Statement

Use cond..else structure to express branch statement in a function.

```
(cond
    ((conditon1) (stmt1))
    ((conditon...) (stmt...))
    ((conditonN-1) (stmtN-1))
    (else (stmtN))
)
```

For example:

```
(def (iseven x)
    (cond
        ((= (% x 2) 0) true)
        (else false)
    )
)
(iseven 2) => True
(iseven (+ 2 1)) => False
```

Also notice the previous example could be written like this:

```
(def (iseven x) (= (% 2) 0))
```

Which is neater and more obvious.

Btw, you should use **HIGH-ORDER FUNCTION** if possible, for its briefness and clarity.

## 4.2, Oops, where were the loops?

Lucida does not provide a loop structure. You need to use **RECURSION** instead.

But again, you should use **HIGH-ORDER FUNCTION** if possible, for its briefness and clarity.

# 5, Recursion

I assume you have enough background on recursion.

If you have any doubt, go to wikipedia or ask stackoverflow.

# 5.1, A simple example

Say I need to get the sum from 1 to 100. And I am not smart enough to use a Gaussian method (`(1+100)*50/2`), all I know is to add these numbers one by one.

Of course you could write it down by press the number key for hundreds times:

```
(+ 1 2 3 4 5 .... 99 100)
```

But remember, the repetitive work should better be left to the dumb box instead of our brain.

Now assume we already get a magic function Sum. Call Sum(n), and you get the sum from 1 to n.

But how to design this function? Just follow these steps.

1, Consider the simple base case:

Sum(0) => 0

Sum(1) => 1

2, Write the inductive equation:

Sum(n) => Sum(n-1) + n

3, Then combine the base case and the inductive equation:

Sum(n)

=> 0 if n<=0

=> 1 if n=1

=> n+Sum(n-1) if n>1

4, Translate the equation to equivalent Lucida expression.

```
(def (Sum n)
    (cond
        ((<= n 0) 0)
        ((= n 1) 1)
        (else
            (+ n (Sum (- n 1))))
        )
    )
)
```

And you have already got it!

# 5.2, Recursions on List

### 5.2.1 Basic List operation

There are 4 basic operation on every List.(Ok, I must confess I copy the ideas from haskell, even the name.)

For example:

```
(def lst (list 1 2 3 4))
lst => [1, 2, 3, 4]
(head lst) => 1
(last lst) => 4
(init lst) => [1, 2, 3]
(tail lst) => [2, 3, 4]
```

Now it's time to show some basic recursion technique on List.

I have already illustrate the idea of recursion before, so I won't do too much talking latter.

If you have any doubts, just follow the previous 4 steps I mentioned.

### 5.2.2 Reverse a list

Inductive Equation:

Reverse(lst)

=> empty list if lst is empty

=> Reverse(rest part of lst) + first element of lst if lst is not empty

Translate it into Lucida:

```
(def (Reverse lst)
    (cond
        ((empty? lst) lst)
        (else
            (+ (Reverse (tail lst)) (list (head lst)))
        )
    )
)
```

### 5.2.3 Get length of a list

Inductive Equation:

Length(lst)

=> 0 if lst is empty

=> 1 + Length(rest part of lst) if lst is not empty

Translate it into Lucida:

```
(def (Length lst)
    (cond
        ((empty? lst) 0)
        (else
            (+ 1 (Length (tail lst)))
        )
    )
)
```

### 5.2.4 Sum a number list

Inductive Equation:

Sum(lst)

=> 0 if lst is empty

=> first element of lst + Sum(rest part of lst) if lst is not empty

Translate it into Lucida:

```
(def (Sum lst)
    (cond
        ((empty? lst) 0)
        (else
            (+ (head lst) (Sum (tail lst)))
        )
    )
)
```

### 5.2.5 List Comparison

Inductive Equation:

Equal(lst1, lst2)

=> True if lst1 is empty and lst2 is empty

=> False if lst1 is not empty and lst2 is empty

=> False if lst1 is empty and lst2 is not empty

=> True

if first letter of lst1 and lst2 are equal

and rest part of lst1 and lst2 are equal

=> False if all the above condition are not satisfied

Translate it into Lucida:

```
(def (listeq lst1 lst2)
    (cond
        ((and (empty? lst1) (empty? lst2)) true)
        ((or (empty? lst1) (empty? lst2)) false)
        ((= (head lst1) (head lst2))
            (listeq (tail lst1) (tail lst2))
        )
        (else false)
    )
)
```

# 6 High-ordered Function

## 6.1 Function as a value

Notice in Lucida(and all functional programming languages as well), function, like other type, could be passed as an argument, or be assigned to a left value.

For example:

```
(def (add x y)(+ x y))
(add 1 3) => 4
(def another-add add)
(another-add 1 3) => 4
(def (f op x y)(op x y))
(f add 1 3) => 4
```

And High-ordered function is, as you see, the function took function as arguments, and utilize the argument function to provide meaningful results.

## 6.2 Why it is useful?

Combining the small high-ordered function could get a concise, neat and powerful function.

I strongly recommend you to read "***Why Functional Programming Matters***" written by John Hughes, he made an excellent description of functional programming.

# 6.3 Three corner stone function of Lucida

### 6.3.1 map

Don't mess Mapreduce(the **cloudy** stuff) with `map`/`reduce` in functional programming, they may have some connections, but they are different stuff essentially.

These small functions would be used later in this section.

```
(def lst (list 1 2 3 4 5))
(def (sqr x) (* x x))
(def (inc x) (+ x 1))
(def (add x y) (+ x y))
(def (gt3 x)(> x 3))
(def (mul x y) (* x y))
```

The map function, which take a function as an argument, then apply this function on the second argument(which is a list).

Use map like this:

```
(map f lst)
```

This is equivalent to the Java-like code below:

```
List result=new List();
foreach(element in lst)
   result.add(f(element));
return result
```

For example:

```
(map sqr lst) => [1, 4, 9, 16, 25]
(map inc lst) => [2, 3, 4, 5, 6]
```

### 6.3.2 filter

The filter function, which take a predicate function as an argument, then get the elements from the second argument(which is a list) which satisfy the predicate.

Use filter like this:

```
(filter predicate lst)
```

This is equivalent to the Java-like code below:

```
List result=new List();
foreach(element in lst)
   if(predicate(element))
      result.add(element);
return result
```

For example:

```
(filter gt3 lst) => [4, 5]
```

### 6.3.3 reduce

The reduce function would be somehow weird, it took an aggregate function and an initial value an an argument, then feed the initial value to the aggregate function as first argument, and "wrap" the elements in the list to a single value.

If you still have any doubt, read John Hughes' paper I mentioned.

Use reduce like this:

```
(reduce aggregate_func init_value lst)
```

This is equivalent to the Java-like code below:

```
var result = init_value
foreach(element in lst)
    result = aggregate_func(result, element);
return result
```

For example:

```
(reduce add 0 lst) => 15
(reduce mul 1 lst) => 120
```

### 6.3.4 Combination of map, reduce and filter

First I have to introduce range function in Lucida, this idea is borrowed from Python.

Use range to generate a list quickly:

```
(range 3) => [0, 1, 2]
```

You could specify an interval:

```
(range 1 3) => [1, 2, 3]
```

And inc/dec step could be specified as well:

```
(range 1 3 2) => [1, 3]
(range 3 1 (- 1)) => [3, 2, 1]
```

Now let's redefine two sample function in the last chapter with map and reduce.

### 6.3.5 Sum from a to b:

1, First define a add function for aggregating.

```
(def (add a b) (+ a b))
```

2, First use range to generate a list of number from a to b

```
(range a b) => (list a ... b)
```

3, Then use reduce to aggregate this list, fairly easy, isn't it?

```
(def (sum a b) (reduce add 0 (range a b))
```

### 6.3.6 Length of a list:

1, First define a "to1" function, it simply returns 1 for every argument

```
(define (to1 x) 1)
```

2, Then "replace" every list element with number 1 use map

```
(map to1 lst) => (list 1 1 .... 1)
```

3, Then aggregate this "replaced" list with reduce

```
(def (length lst) (reduce add 0 (map to1 lst)))
```

As I mentioned, no loops, no conds, high-order function could solve the background complexity for you.

# 6.4 Anonymous function

You might found that to1 is very simple and only used once.In fact, you would declare many small functions in Lucida so you could feed these high-order functions. And these small function would cause some problems to the global scope.

In this condition, which a function is very small and only be used once, you could use an anonymous function( Anonymous function is often refered as lambda function, I don't use lambda because it only reminds me the brainfucking math, and it does no help to your understanding. At least, the so-called lambda function in my eye is just a function without name, that is it).

You could use `func` keyword(which is the **fifth and last** key word of Lucida) to define a anonymous function:

```
(func (para1 ... paraN) ( ...funcbody...))
```

You could invoke anonymous function directly, just like calling a named function.

For example:

```
((func (x) x) 1) => 1
((func (x y) (* x y)) 2 3) => 5
```

A anonymous function could be assigned to a named value. Then you could use this name to invoke this function.

For example:

```
(def sqr (func (x) (* x x)))
sqr => Func(x)
(sqr 3) => 9
```

After all, a anonymous function could be passed as an argument, and that's what we need.

For example:

```
(reduce (func (x y) (+ x y)) 0 (range 1 10)) => 45
(filter (func (x) (= (% x 2) 0)) (range 1 10)) => [2, 4, 6, 8, 10]
```

# 6.5 Curry function

**Haskell Curry** is a famous mathematician and logician, and this is where Curry comes. And this is the second and **LAST** time I mentioned math.

**Curry function**, in fact, is a **partially-valued function**, it's a bit weird, and to be honest, I don't like it.

Let's take add as an example:

```
(def (add x y) (+ x y))
```

Of course, `(add 2 3)` would return `5`, but what would happen if I just feed it one argument? Which means, what would `(add 2)` return?

In most mainstream programming languages, this would cause a argument-parameter mismatched error, but in Lucida (and most functional programming languages):

```
(add 2) => Func(y)
```

It would return a function with one parameter, and this function is **partially-valued**, I know this sounds pecuiliar, but think like this:

```
(def (common_add2 y) (add 2 y))
(def curry_add2 (add 2))

(common_add 3) => 5
(curry_add2 3) => 5
```

Just remember, `(add 2 3)` and `((add 2) 3)` is somewhat equivalent.

Although I do not like Curry function, but I have to say it could save you some work in some scenarios. Say you want to multiply every element in a list by 2. If you use anonymous function, things would be like this:

```
(map (func (x) (mul 2 x)) (range 3)) => [0, 2, 4]
```

Curry function would save you some typing, and the code seems more clear:

```
(map (mul 2) (range 3)) => [0, 2, 4]
```

# 6.6 Generated function

With Curry function and anonymous function, you could define a function which could returns a function.

For example:

```
(def mul3 (mul 3)) => Func(y)
(mul3 4) => 12

(def sqr (func x)) => Func(x)
(sqr 5) => 25
```

With this mechanism, you could define a function which "flip" the order of function's parameter. This "flip" function is very useful when you have to pass arguments in a reverse order.

```
(def (flip f) ((func (x y) (f y x))))
((flip sub) 3 1) => -2
(filter ((flip gt) 3) (range 1 5)) => [4, 5]
```

# 6.7 Sort for quick

It seems every functional programming language would take quick sort as an example, so does Lucida.

I won't spend too much time talking about what is quick sort, all you need to know is:

*Quick sort could sort a list of elements quickly, it select one element as a pivot, then move elements from the rest which is greater than the pivot to the right side, the smaller elements would be moved to left side. Recursively carry this procedure until a list with 1 or 0 element.*

For example:

```
[7, 4, 5, 8, 3, 9, 1]
   => [4, 5, 3, 1] + [7] + [8, 9]
   => [[3, 1] + [4] + [5]] + [7] + [[8] + [9]]
   => [[[1] + [3]] + [4] + [5]] + [7] + [[8] + [9]]
   => [1, 3, 4, 5, 7, 8, 9]
```

In imperative programming languages like C++ or Java, writing a qsort is often a messy job. Instead of focusing on the quick sort algorithm, you are forced to consider every possible off-by-one and index-out-of-range mistakes. What qort would be in functional programming languages?

Inductive Equation:

qsort(lst):

=> empty list if lst is empty

=> lst if lst contains only one element

=> otherwise

qsort(smaller rest part of lst) + first element of lst + qsort (bigger rest part of lst)

Translate it into Lucida:

```
(def (qsort lst)
   (cond
      ((or (empty? lst) (empty? (tail lst))) lst)
      (else
         (+
            (qsort (filter (gte (head lst)) (tail lst)))
            (list (head lst))
            (qsort (filter (lt (head lst)) (tail lst)))
         )
      )
   )
)
```

Quick and clean, isn't it?

# 7 A quick reference

# 7.1 Builtin function

### 7.1.1 add, sub, mul, div, mod

They are "functional" forms of corresponding arithmetic operator.

Lucida does not permit you pass an operator as a argument, but you could use them instead.

Remember all of them take EXACTly 2 arguments.

```
(add 1 2) => 3
(add (list 1 2) (list 3)) => [1 2 3]
(mod 3 2) => 1
(mul 4 5) => 20
```

### 7.1.2 gt, lt, gte, lte, eq

They are "functional" forms of corresponding comparison operator. gt stands for "greater than" and lt stands for "less than".

```
(gt 3 2) => True
(lte 3 4) => True
(eq 'b' 'c') => False
```

### 7.1.3 head, init, tail, last

Four fundamental list operation:

```
(def lst (range 4))
lst => [0, 1, 2, 3]
(head lst) => 0
(last lst) => 4
(init lst) => [0, 1, 2]
(tail lst) => [1, 2, 3]
```

### 7.1.4 map, filter, reduce

Three most useful high-order functions:

```
(def lst (range 1 4))
lst => [1, 2, 3, 4]
(map (func (x) (* x x x)) lst) => [1, 8, 27, 64]
(filter (func (x)(= (% x 2) 0)) lst) => [2, 4]
(reduce add 0 lst) => 10
(reduce mul 1 lst) => 24
```

### 7.1.5 range

For example:

```
(range 3) => [0, 1, 2]
(range 1 3) => [1, 2, 3]
(range 3 1) => []
(range 1 10 2) => [1, 3, 5 ,7 ,9]
(range 10 1 (- 2)) => [10, 8, 6, 4, 2]
(range 10 1 2) => []
```

### 7.1.6 aoi, itoa

Use them to make conversion between digit and characters:

```
(atoi '3') => 3
(itoa 9) => '9'
(map atoi (stol "123")) => [1, 2, 3]
(reduce
    (func (a b) (+ (* 10 a) b))
    0
    (map atoi (stol "123"))
) => 123
```

### 7.1.7 stol, ltos

Use them to listify a string, or stringify a list.

```
(stol "abc") => ['a', 'b', 'c']
(ltos (list 'L' 'u' 'c')) => "Luc"
(ltos (stol "This makes no sense")) => "This makes no sense"
```

### 7.1.8 len, elem

Function len and elem could be applied on String and List type:

```
(len "Meling") => 6
(elem 2 "Lucida") => 'c'
(len (list 1 4 2 8 5 7)) => 6
(elem 0 (list 'l' 'i' 'n')) => 'l'
```

### 7.1.9 number?, double?, char?, string?, function? list?, empty?

These function are used to judge a type of a given value, a Bool result would return.

The last function empty? is used to judge whether a list is empty.

```
(number? 1) => True
(double? 2) => False
(char? 'a') => True
(function? 1) => False
(list? (range 5)) => True
(empty? (range 2 1)) => True
```

### 7.1.10 error

You could regard this function as the Exception mechanism in Lucida, however, there won't be a catch.

```
(error "Catch me if you can") => Error: Catch me if you can
```

# 7.2 Some functions from standard library

You could infer the usage of these function easily by their name.

Almost all of them follow Haskell convention.(or LINQ convention or Python itertools convention, I wonder who copies who?)

These small functions would be used later in this section.

```
(def lst (list 1 2 3 4 5))
(def (sqr x) (* x x))
(def (inc x) (+ x 1))
(def (even? x) (= (% x 2) 0))
(def (add x y) (+ x y))
(def (gt3 x)(> x 3))
(def (mul x y) (* x y))
```

### 7.2.1 zip

For example:

```
(zip lst lst) => [[1, 1], [2, 2], [3, 3], [4, 4], [5, 5]]

(map
    (func (pair) (* (head pair) (last pair)))
    (zip lst lst)
) => [1, 4, 9, 16, 25]

(map sqr lst) => [1, 4, 9, 16, 25]
```

### 7.2.2 zipwith

For example:

```
(zipwith mul lst lst) => [1, 4, 9, 16, 25]
(def (palindrome? lst)
    (zipwith eq lst (reverse lst))
)

(palidrome (list 1 2 1)) => True
(palidrome (list 2 3 1)) => False
```

### 7.2.3 take

For example:

```
(take 3 lst) => [1, 2, 3]
(take 10 lst) => [1, 2, 3, 4, 5]
(take 1 (zip lst lst)) => [[1, 1]]
```

### 7.2.4 skip

For example:

```
(take 3 lst) => [4, 5]
(take 10 lst) => []
(skip 3 (zip lst lst)) => [[4, 4], [5, 5]]
```

### 7.2.5 takewhile

`takewhile` will **keep** taking elements **until** a element satisfy the predicate.

For example:

```
(takewhile even? (list 1 2 3)) => [1]
(takewhile (func (x) true) (list 1 2 3)) => []
```

### 7.2.6 skipwhile

Unlike `takewhile`, `skipwhile` will **begin** taking elements **when** a element satisfy the predicate.

For example:

```
(skipwhile even? (list 1 2 3)) => [2, 3]
(skipwhile (func (x) true) (list 1 2 3)) => [1, 2, 3]
```

### 7.2.7 any

`any` would return `True` if **any** element in the list satisfy the predicate.

For example:

(any even? (list 1 2 3)) => True

(any even? (list 1 3)) => False

### 7.2.8 all

`all` would return `True` if **all** elements in the list satisfy the predicate.

For example:

```
(all even? (list 1 2 3)) => False
(all even? (list 2 4)) => True
```

### 7.2.9 reverse

For example:

```
(reverse lst) => [5, 4, 3, 2, 1]
(reverse (list )) => []
(ltos (reverse (stol "Luc Meling"))) => "gnileM cuL"
```

### 7.2.10 flip

Use `flip` to **reverse** the order of parameters of a function.

For example:

```
(div 1.0 2.0) => 0.5
(flip div) => Func(y x)
((flip div) 1.0 2.0) => 2.0
```

### 7.2.11 sort

For example:

```
(def lst1 (list 2 3 1 4 5))
(sort lst1) => [1, 2, 3, 4, 5]
(sort (list )) => []
(sort 5) => Could not apply sort on Number type
```

# 7.3 Special eVil Operator

Before you look into this, I have to remind you again, these operators are EVIL. If you like clean and neat programs, just skip this section.

You might ask why I import these strange operators, the answer is simple: I don't like typing. And that's the reason I use def instead of define, func instead of lambda.

```
! => def
: => cond
; => else
\ => func
. => list

{ => head
} => last
[ => init
] => tail

@ => map
$ => reduce
? => filter
~ => range
```

Just look at the qsort function in the last chapter written with these operators.

```
(! (_ l)(:((or (e? l) (e? (] l))) l)(; (+(_ (? (gte ({ l)) (] l)))
(. ({ l)) (_ (? (lt ({ l)) (] l)))))))
```

I doubt if anyone **functional** could read this **functional** program, except a **functional** compiler.

# 8 Interpreter Command

All interpreter command starts with a colon.(Still, borrowed from haskell)

More advanced command would be added later.

## 8.1 Loading script

For example:

```
:load lib.luc => load the function in lib.luc to current scope
```

## 8.2 Displaying function and variable

Use :dir to display defined functions and variables in current environment.

For example:

```
(define x 1)
(define (f x) (+ x x))
(define (f1 x y) (* x y))
:dir
   =>
    x:1
    f:Func(x)
   f1:Func(x, y)
:dir f
   =>
    f:Func(x)
   f1:Func(x, y)
```

# 8.3 Removing defined function or variable

Use :del to delete defined functions and variables in current environment.

Use :reset to reset current environment to initial condition.

Remember :reset won't delete functions in standard library, but :del would kill'em all!

For example:

```
(def x 1)
:del x
   => Variable x has been deleted from current scope
(def (f x)(+ x 1))
(def x 2)
:del
   => Current variables and functions would be cleared, are you
sure? y
   => Current variables and functions were cleared
:dir
   => Nothing displayed
```

# 9 Further reading

## For functional programming:

- **How to design programs**
- **The Structure and Interpretation of Computer Programs**
- **Why Functional Programming Matters?**

## For interpreter constructing:

- **The Language Implementation Patterns**
- **Domain Specific Languages**

## For every programmer:

- **The C Programming Language**
- **The Practice of Programming**
- **Elements of Programming**