LUCAS Language Specification Document
Team 3

JATIN TARACHANDANI
CS20BTECH11021

JULAKUNTLA MADHURI
CS20BTECH11023

PRASHANTH SRIRAM S
CS20BTECH11039

S GOUTHAM SAI
CS20BTECH11042

SHANTANU PANDEY
CS20BTECH11046

ARAVINDA KUMAR REDDY T
CS20BTECH11053

ANIRUDH SRINIVASAN
CS20BTECH11059

SRIVATSAN T
CS20BTECH11062

25th August 2022

Contents

1 Introduction

1.1 About the Language

1. "Lucas - Programming made simple" is a compiled, statically typed, object based language with calculus features and support for arbitrary precision integers as its USP.

2. It draws inspiration from some of the popular languages like C++, Java, Go, COOL, LaTeX and many more.

1.2 Why Lucas?

1. The need for our language is because of the unique combination of features that we support. We have designed this language for mathematicians, academicians, scientists, business analysts and for anyone who might need calculus support. The idea behind our language is to keep all relevant features bundled so that it's easy for the programmer to get the computational details that he needs, all at one place instead of depending on multiple languages in a technological stack.

2. We are planning to start with a support for calculus functionalities that can evaluate accurately the differentials at a point and integrals over an interval with certain level of user-defined precision level. We'll add upon relevant packages for differential equations and anymore related functionalities in our next release based on the feedback from the user community.

1.3 Design Goals

1. We want to make life easier for the programmer, so our focus is on bringing a balance between productivity and performance. Based on the use cases we could imagine of, we decided to go with an object based language that supports data encapsulation features.

2. The idea of having begin and end with labels for any block of code comes from LaTeX, which majority of the users, (mostly mathematicians) are already familiar with. And also, dangling if-else problem can be taken care of, with the help of proper labelling.

3. Also, we support multiple return types that can returned from a function. The programmer doesn't need to encapsulate the return values into a single structure before returning it from the function.

4. Arbitrary precision integers are represented as the new non-primitive datatype **bigint** that the language supports.

5. We support 32-bit int data type as well, which can be used if the programmer has the domain knowledge that the variables are going to be within the range supported for normal integers.

6. And necessary exceptions will be thrown, whenever the compiler feels that there might be a chance of overflows or if int might not be suitable for the operation.

2 Lexical Conventions

2.1 Comments

- In Lucas, the `//` symbol is used to denote an in-line comment. All the tokens after the `//` token are ignored until a newline is encountered.

```
//This is a single line comment
int i; // All tokens here are ignored

```

Listing 1: Single line Comments

- Moreover, the `/*` marks the beginning of a multi-line comment. The `*/` token denotes the end of a multi-line comment. All tokens in between the opening and closing tokens of a multi-line comment will be ignored.

```
bigint i = 0;
/* This is a multi-line comment or block comment.
   All characters here would be ignored. */

```

Listing 2: Block Comments

2.2 Whitespaces

- Whitespaces such as spaces, tabs and comments will be ignored except for those in between tokens and those inside quotes. For example:

```
begin function main() -> int
    bigint i = 21;
    bigint j = i*i;
end function main

```

and the function

```
 begin function main() -> int
    bigint i =
    21;
    bigint
    j = i*
    i;
 end function main

```

are both identical.

2.3 Punctuations

- All statements in the code must be terminated with a semicolon **;**
- Arguments and return values in a function have to be comma ',' separated.

3 Data Types

3.1 Primitive Data Types

The primitive datatypes supported by Lucas are: signed 32-bit integers, signed 64-bit floating point values, boolean variables, single characters from the ASCII encoding format, and strings of characters.

- Signed 32-bit integers, declared with the `int` keyword.
- Signed 64-bit floating point values, represented according to the IEEE 754 double precision specification. These are declared by the `double` keyword.
  - 1 bit for sign.
  - 12 bits for exponent.
  - 52 bits for storing the significand.
- Character values encoded as integers from 0 - 127, representing characters from the ASCII standard. These are declared with the `char` keyword.
- Boolean values that represent `true` and `false` values. These are declared with the `boolean` keyword.
- String values analogous to `string` in C/C++ as a sequence of characters. These are declared with the `charseq` keyword.

3.2 Non-Primitive Data Types

The non-primitive data types supported by Lucas are to implement arbitrary precision integers and to handle calculus calculations.

- **Arbitrary Precision Integers (bigint):** A data type to store integers with infinite precision. The keyword `bigint` is used to declare a variable of this type. `bigint` variables supports all the operations an `int` variable can support except for the bit-wise operations.

  1. The idea is to emulate what java does with its own BigInteger datatype, which is store an array of 32 bit integers instead of single digits. operations will then be done on an individual basis for the corresponding elements in the array, with appropriate carry overs implemented as additions into the next element of the operation.

2. Our plan was to begin with this at the implementation level and see the practical limits of such a solution. We will not impose an explicit limit, but if memory issues occur a runtime error or a special infinity value can be returned (something like NaN in python)

– **Integer Literals:** By default, all the integer literals are treated as `bigint` literals. To make an integer literal an `int` literal, the suffix `I` need to be added to the end of the literal.

For example:

```
    // Assume print is a library function
    // typeof is an operator, returns a string
    print(typeof(3)) //  prints bigint
    print(typeof(2I)) // prints int

```

Listing 3: Integer Literals

The default value of a `bigint` variable is 0.

– **Conversions between `int` and `bigint`:**

1. An `int` value (a variable or a literal) can be assigned to a `bigint` variable without any issues.
2. When a `bigint` value (a variable or a literal) is assigned to an `int` variable, It is checked in the run-time, whether this value is within the limits of an `int` variable or not and if it is within the limits, it will be stored without any issues. Else, a run-time error occurs.
3. In an operation involving a `bigint` value and a `int` value, the `int` value is promoted to a `bigint` value and the operation is performed.
4. The compiler will raise a warning if a `bigint` literal outside of the limits of an `int` is assigned to an `int` variable.

```
    bigint x = 3I; //  OK
    int y = 2; // OK
    bigint z = y; // OK
    int w = z; // 2 is within the limits of int, OK
    int l = 100000000000; // Warning raised by compiler +
    Run-time error occurs
    bigint bl = 1000000000000; // OK
    int m = bl; // Run-time error
    bigint n = y+z; // y is promoted to bigint and then +
    is performed

```

Listing 4: int - bigint conversions

Note: Invalid datatype conversion be handled by raising a runtime error stating the types that were attempted to convert to and from.

- **Classes:** Lucas is an object based language. As such, it supports classes to encapsulate the data members and member functions and supports `private` and `public` access modifiers to allow abstraction.

  Note: Some important features regarding classes are:

  - Lucas does not support Inheritance. So it isn't Object oriented language as such.
  - Classes are declared and defined using the `class` keyword.
  - Classes can be nested (One class defined inside another class).
  - Data members can be any of the primitive and non-primitive types, except for an object of the same class.
  - Class can be declared using the `decl` keyword.
  - Classes can be defined directly without declaring them beforehand.
  - The Default access modifier is private.

```
decl class class1;//declare class class1
begin class class1
    int X;
public:
    begin function getX() -> (int)
        return X;
    end function getX
    begin function setX(int x) // if no return type, ->() is optional
        X = x;
    end function setX
    begin function class1() // constructor - optional
        X = 2;
    end function class1
end class class1

```

Listing 5: Class

```
// Example of Nested Class:
begin class class2 // directly define without decl
private:
    class1 C1;
    begin class innerclass
    public:
        int z;
    end class innerclass
    innerclass C2;
public:
    int y;
    decl function getDetails() -> (int, class1);
    decl function getInnerZ() -> (int);
```

```
end class class2

```

Listing 6: Nested Classes

Note: The Syntax for member functions is exactly the same as the syntax for ordinary function (which is defined in Section 6), except that member functions can be defined outside the class too using the scope resolution operator `::` .

```
// For the above defined class class2,..
begin function class2::getDetails() -> (int, class1)
    return (z, C1);
end function getDetails

```

Listing 7: Defining member functions outside the class

Here, we are attempting to do what c++ does with being able to define class functions outside the body of the class.

Instantiation of Classes:

```
    // Assume class2 is defined
    class2 obj;

```

Listing 8: Instantiation of Classes

Note: Data members and member functions of an object can be called with the object using the `.` operator

```
    class2 obj1;
    obj1.C2.z = 3;
    print(obj1.getInnerZ());

```

Listing 9: Accessing members

Constructor: Constructors can be written to override the default constructor. The default constructor will be deleted if any user defined constructor for that class is written. Constructor should have the same name as the class and should not return any values.

```
begin class class3
public:
    bigint var1;
    begin function class3() // fn1
       var1 = 2;
    end function class3
    begin function class3(int val) // fn2
        var1 = val;
    end function class3
end class class3

```

Listing 10: Constructors

If such a class was defined, then,

```
class3 obj // implicit call to fn1
print(obj.var1); // prints 2
class3 obj2(3);//implicit call to fn2
class3 obj3 = class3(4);//explicit call to fn2
print(obj2); //prints 3
print(obj3); //prints 4

```

Listing 11: Implicit and Explicit calls to Constructor

Destructor: If a user defined destructor is written, the default destructor is deleted. Destructor takes no arguments and returns nothing. Destructor should have the name as a `~` appended to the name of the class.

```
begin class class4
public:
    bigint var1;
    begin function ~class4()
        print("Destroyed!");
    end function ~class4
end class class4

```

Listing 12: Destructor

If such a class was defined,

```
class4 obj1;

```

Listing 13: Destructor example

After this object goes out of scope and gets destoryed, Destroyed! will be printed.

- **Calculus Expressions**: A datatype to store mathematical expressions which we can use to find value of the function, derivative or integral at any point. The keyword `expr` will be used to declare a variable of this type.
  - We can use a `expr` variable to perform differentiation and integration of single variable functions.
  - We have to declare the expr before writing it and have to declare the variable to be used by the mathematical expression. For Example,

    ```
        expr e;
        e.var = 'x';
        e = 4*x^3; // This represents the expression 4x^3

    ```

    Listing 14: expr declaration

  - We can define expressions to be the derivatives of other expressions as follows:

```
expr e1.var = 'x';
e1 = 7x^4; // e1(x) = 7x^4
expr e2 = e1.diff(); // e2(x) = 28x^3

```

Listing 15: Derivatives of a function

– Moreover, we can pass arguments to the `diff()` function to find the value of the derivative at a particular point. For example,

```
expr f.var = 'x';
f = 4x^3; // f(x) = 4x^3
bigint i = f.diff(2) // i stores the value 48 in this case

```

Listing 16: Derivative at a point

– Similarly, we can define an expression to be the integral of another expression as follows:

```
expr f.var = 'x';
f = x; // f(x) is x
expr F = f.intg(); // F will be x^2/2

```

Listing 17: Integrals of a function

– Additionally, you can pass values to the `intg()` function to find the value of the definite integral. For example,

```
expr e.var = 'x';
e = x; // e(x) = x
bigint i = e.intg(1,3); // i has the value 4
```

Listing 18: Definite Integral

– We can find the value of an expression at a particular value by using the `eval` function on the expression. For example;

```
expr e.var = 't';
e = t^3; // e(x) = x^3
bigint i = e.eval(5); // i will have the value 125

```

Listing 19: Evaluating functions at a discrete point

- **Vectors:** Vectors are a homogeneous composition of elements that can dynamically resize itself like a dynamic array whenever elements are added or removed. It can support non-primitive data types. Some important features of this type are:

  – Vectors follow LIFO principle when it comes to insertion and deletion by default.

– The keyword `vector` has to be used for initialization of vectors. Since it is a homogeneous composition of elements, we have to define the type of the `vector` done with the < > operator. The type of elements of the `vector` have to be mentioned inside this angular brackets at the time of declaration.

  will be used for accessing elements of the vector with positions mentioned inside the square brackets. The numbering of the positions start from 0 like in C or C++.

– We can do the following operations with vectors:

  * Insert: This can be used to insert elements into the vector. By default, insertion occurs at the end. We can insert an element also at an arbitrary position within the length of the vector and push the appropriate elements to the right.
  * Delete: This can be used to delete elements from the vector. By default, deletion occurs at the end. We can delete an element also at an arbitrary position within the length of the vector and push the appropriate elements to the left.
  * Update: This can be used to update the value of a particular element of the vector within the length of the vector.
  * Search: This will return the the position of the element searched for. Will return -1 if the element doesn't exists

  Example:

```
1            vector <int> v = {1, 2, 3}; // v[1] has the
   value 2
2            v.insert(4); // updated v is {1, 2, 3, 4}
3            v.insert(5,2); // This will insert 5 at the 3
   rd position (as counting starts from 0). updated v is
   {1, 2, 5, 3, 4}
4            v.
5            v.delete(); // updated v is {1, 2, 5, 3}
6            v.delete(2); // updated v is {1, 2, 3}
7            v.update(10, 1); // updated v is {1, 10, 3}
8            v.search(3); // returns 2 as the position of 3
    in the vector is 2
9
10
```

Listing 20: Creating a variable of vector type and performing basic operations on it

3.3 Reserved Keywords

The keywords reserved by the Lucas compiler are :

bigint	boolean	char	charseq
double	else	end	false
for	if	int	expr
decl	true	while	begin
class	function	public	private
return	typeof	break	continue
vector			

4 Operators

4.1 Types

4.1.1 Arithmetic Operators

Operator	Input Operands	Output	Description
+	(numeric, numeric)	numeric	Returns sum of the 2 operands.
-	(numeric, numeric)	numeric	Returns difference of the 2 operands.
*	(numeric, numeric)	numeric	Returns product of the 2 operands.
/	(numeric, non-zero numeric)	numeric	Returns quotient when the first operand is divided by the second.
%	(numeric, non-zero numeric)	numeric	Returns the remainder when first operand is divided by the second.
∧	(numeric, integer)	numeric	Returns the result when the first operand is raised to the power of the second operand.

4.1.2 Relational Operators

Operator	Input Operands	Output	Description
<	numeric, char or string	boolean	Returns true if operand 1 is less than operand 2.
>	numeric, char or string	boolean	Returns true if operand 1 is greater than operand 2.
==	numeric, char or string	boolean	Returns true if operand 1 is equal to operand 2.
!=	numeric, char or string	boolean	Returns true if operand 1 is not equal to operand 2.
<=	numeric, char or string	boolean	Returns true if operand 1 is less than or equal to operand 2.
>=	numeric, char or string	boolean	Returns true if operand 1 is greater than or equal to operand 2.

4.1.3 Logical Operators

Operator	Input Operands	Output	Description
&&	boolean, boolean	boolean	Returns the logical AND of the 2 input boolean expressions.
‖	boolean, boolean	boolean	Returns the logical OR of the 2 input boolean expressions.
!	boolean	boolean	Returns the logical NOT of the input boolean expression.

4.1.4 Other Operators

Operator	Description
.	Member access operator. Returns the value of the desired field or the returned value of the called function.
=	Assignment operator. Used to assign the value of the expression on the right hand side to an identifier on the left hand side.
::	Operator used to refer to the functions of a specific class when they are being implemented by the programmer outside the class definition.
[	Enclose labels within brackets, Specify index to access in vectors
]	Enclose labels within brackets, Specify index to access in vectors
<	To specify the type of the vector
>	To specify the type of the vector

Note: "->" is also a reserved symbol, used to denote the set of values returned by a declared function.

4.2 Order of Precedence

It is similar to the order of precedence in C++. However, ^takes more precedence than () in C++. The associativity also remains the same in comparison to C++. You may see the order of precedence table for C++ here

5 Statements

5.1 Selection statements

Selection statements are used in decision-making situations where the flow of the program depends on the current value of the chosen constraint.
Lucas supports the following selection statements:

- `if`
  The if conditional is used in situations where a bunch of statements need to be executed given a constraint is true. Many programming languages use a pair of curly braces ({}) to denote the beginning and ending of a conditional block. But they often lead to dangling curly braces when they're not paired properly and may lead to compile errors. They are often difficult to debug as well. To counter this, Lucas uses optional parameters or labels for conditionals and iterative statements which can be used to terminate out of that block gracefully. The inclusion of labels in `end if` statement is also optional in which case it is matched with the nearest `begin if` statement.

  Below is the syntax of a typical `if` block.

```
        begin if[Label](condition)
            statements
        end if Label

```

Listing 21: If Statement

The above example shows an if statement with a label 'Label'. The approach is especially useful once we have nested if statements like shown below:

```
        begin if[label1](condition 1)
            begin if[label2](condition 2)
                Statements
            end if label2
        end if label1

```

Listing 22: Nested If Statement

- `if...else`
  The `if...else` conditional is used in cases where there are 2 statements and the control flow is decided by the truth value of the constraint.
  Below is the syntax for `if...else` statements.

```
        begin if[Label](condition)
            statement 1
        end if Label
        begin else
            statement 2
        end else

```

Listing 23: if...else Statement

- `if...else if...else`
  This conditional statement is used in cases where there are multiple constraints and corresponding blocks of code that needs to be executed provided that the constraint is true. The `if...else if...else` makes sure that the present block of code gets executed only when the current constraint is true and all the previous constraints are false. Thus only one among the n blocks of code is executed.
  Below is the syntax for `if...else if...else` statements.

```
        begin if[Label 1](Condition 1)
            Statement 1
        end if Label 1
        begin else if[Label 2](Condition 2)
            Statement 2
        end else if Label 2
        ...
        begin else
            Statement n
        end else

```

Listing 24: if...else if...else Statement

Note: If all the `if` and `else` statements use labels, the dangling else problem is entirely avoided. But, since the labels are optional, if no labels are used, the dangling else problem arises. It is resolved in the same way C++ resolves this ambiguity, that is, the unlabelled `else` is matched to the closest unmatched `if`.

5.2 Jump statements

Jump statements are used for unconditional control flow switches in the program. They can be used to change the flow of the program run to any other arbitrary point in the program.
Lucas has support for 3 types of jump statements They are as follows:

- `break`
  Break statement, when encountered inside an iterative block of code, terminates the executing of the loop and the control is passed onto to the next line after the loop.
  The following is the syntax for `break`:

```
            break label;
            break;

```

Listing 25: break statement

  Break statement when called along with a label terminates the loop with the provided label and returns the control flow to the line following the ending of that loop.
  When called without a label, the break statement simply breaks out of the closest loop to it.

- `continue`
  Continue statement is used to skip an iteration in a looping statement. The control is brought to the beginning of the block of looping statements after incrementing the looping variables.
  The following is the syntax for `continue`:

```
            continue label;
            continue;

```

Listing 26: continue statement

  When continue statement is called along with a label, the control is brought to the beginning of the loop denoted by the label after updating the looping variables.
  When invoked without a label, the continue statement skips one iteration of the closest loop.

- `return`
  The return statement terminates the execution of function and returns the control to the caller function. Unlike `break` and `continue`, which are optional and are used in specific cases, return statements typically accompany every function defined.
  The following is the syntax for `return`:

```
            return value;
            return (value1, value2, value3 ..., value n);

```

Listing 27: return statement

Lucas supports returning multiple values of heterogeneous types from a function. The first syntax above is to be used in cases where the function returns a single value. The second syntax is to be used when the function returns n values.

5.3 Iterative statements

Iterative statements are useful when statements are required to be executed zero or more times, subjective to looping criteria which are set according to the need. Lucas supports the nesting of Iterative statements and these statements are executed in order, except when they encounter `break` statement or `continue` statement.

Lucas supports 2 loop statements `for` and `While`. Many programming languages use a pair of curly braces ({}) to denote the beginning and end of an Iterative statement. In Lucas, we tried to avoid a common error that arises due to the use of curly braces by using `begin` and `end` statements, similar to that of selection statements. labels can also be used to identify each iterative statement cautiously to avoid errors.

Now, let's describe the two iterative statements that Lucas supports:

- `for`
  The `for` statements is used to execute a given statement rapidly until a certain given condition becomes false.Two jump statements : `begin` and `end` statements will be used along with optional use of labels.
  We use the `for` statement to construct loops that must execute a specified number of times.

  Below is the syntax of `for` statement:

```
        begin for[Label] ( init-expression ; cond-expression ;
    loop-expression)
         statement
        end for Label

```

Listing 28: for statement

Below is the syntax of a nested/compound `for` statement:

```
        begin for[Label1] ( init-expression ; cond-expression
; loop-expression)
        statement
                  begin for[Label2] ( init-expression ; cond-
expression ; loop-expression)
                     statement
                  end for Label2
        end for Label1

```

Listing 29: nested for statement

Below is the syntax of a nested/compound `for` statement with use of conditional statements and `break` and `continue`:

```
        begin for[Label1] ( init-expression ; cond-expression
    ; loop-expression)
         statement
                  begin for[Label2] ( init-expression ; cond-
    expression ; loop-expression)
                      statement

                           begin for[Label3] ( init-expression
    ; cond-expression ; loop-expression)
           statement
                            begin if[Lab](statement)
                                 break Label2;
                             end if Lab
                             begin else if[Lab2](statement)
                                 continue Label3;
                             end if Lab2
                  end for Label2
        end for Label1

```

Listing 30: nested for statement with use of break and continue statements

- `while`
  This iterative statement helps us execute a code zero or more times till a certain condition holds true. Similar to the `for` statement, two jump statements and labels are supported in the while statement.

  Below is the syntax of a `while` statement:

```
        begin while[Label1] (statement)
            statement
        end while Label1

```

Listing 31: while statement

Below is the syntax of a nested/compound with use of conditional statements and `break` and `continue` statement:

```
        begin while[Label1] (statement)
            statement
                begin while[Label2] (statement)
                    statement
                    begin while[Label3] (statement)
                        statement
                        begin if[Lab1](statement)
                            break Label3;
                        end if[Lab1]

                        begin else if[Lab2] (statement)
                            continue;
                        end else if

                end while Label2
```

```
16          end while Label1
17
```

Listing 32: while statement

6 Functions

6.1 Syntax

Declaration:

```
1 decl function fnname(argtype arg1, ..., argtype argn);
```

Listing 33: Function Declaration Syntax

Note, the names of the argument variables are optional in the declaration. The declaration itself is optional unless there is a need to call the function before the function is defined.

Definition:

```
1 begin function fnname(argtype arg1, argtype arg2, ..., argtype argn
      ) -> (rettype1, rettype2, ..., rettypem)
2     ....
3     return (ret1, ..., retm);
4 end function fnname
```

Listing 34: Function Definition syntax

In the return statement, the parantheses are optional. If no `-> (...)` are provided, it is assumed as a void function

6.2 Example:

```
1 decl function add(bigint x, bigint) -> (int);
2 begin function add(bigint a, bigint b) -> (int)
3     bigint c = a+b;
4     return c;
5 end function add
```

Listing 35: Example 1 for function

```
1  decl function sayHi();
2  begin function sayHi()
3      print("Hi");
4  end function sayHi
5  // ->() is optional if returning nothing
6  // Defining directly without declaring first...
7  begin function sayHello() ->()
8      print("Hello");
9      return (); // returns nothing
10 end function sayHello
```

Listing 36: Example 2 for function

6.3 Calling the function

Syntax:

```
// Assuming f, retvar1,...,retvarm is already declared
// arg1...,argn are either literals or variables/objects already declared
(retvar1, .., retvarm) = f(arg1, ..., argn);
// OR since the paranthesis for return values are optional
retvar1,...,retvarm = f(arg1,...,argn);
```

Listing 37: Function Calling syntax

Example:

```
// Assume f takes (bigint, char, int) as arguments and return (bigint, double, int)
bigint x;
double y;
int z;
char w;
x, y, z = f(3, w, 2I);
```

Listing 38: Calling Functions Example

7 Non Trivial Input Programs

7.1 Program-0:

```
        begin function main()->(int)

            charseq ch_seq="Welcome to Lucas";//Charseq is like string in cpp

            printf(ch_seq+"\n");

            return 0;

        end function main
```

Listing 39: A basic program

This is just a simple program. And through this program we wanted to represent that we use the word `charseq` instead of string.

7.2 Program-1:

```
        begin function main()->(int)

            bigint fact=1;

            int num=50;

            begin for(int i=1; i<=num; i++)

```

```
                fact*=i;

            end for

            print("Factorial(50)="+fact+"\n");//we cannot store the
     value of 50! in int data type but we can store it in bigint
    data type

            return 0;

        end function main
```

Listing 40: An application of bigint

In the above program we highlighted the usecase of `bigint`. 50! is a huge number that cannot be stored in `int`, and here comes the role of `bigint` to store that value.

7.3 Program-2:

```
        begin class cls
        public:
            begin function fun(int a, int b)->(int, int, int, int)

                int add=a+b;

                int sub=a-b;

                int mul=a*b;

                int div=a/b;

                return (add, sub, mul, div);

            end function fun

        end class cls

        begin function main()->(int)    //This is our main function

            int num1, num2;

            print("Enter two integers to get their sum, difference,
     product and ratio:\n");

            read(num1, num2);//taking input

            int sum, dif, prod, ratio;

            cls c;

            (sum, dif, prod, frac)=c.fun(num1, num2);

            print("sum="+sum+"::difference="+dif+"::product="+prod
    +"::ratio="+ratio+"\n");//printing the output

```

```
            return 0;

        end function main
```

Listing 41: Returning Multiple Values

The above program is just a representation of how our functions return multiple values and syntax of programs dealing with such functions. In the above program we are just returning the values of results of the 4 arithmetic operations when performed on the given 2 variables.

7.4 Program-3:

```
        begin function main()->(int)//This is our main function

            expr e;

            print("Enter the variable:\n");

            read(e.var);

            print("Enter the expression:\n");

            read(e);

            double x_val;

            print("Enter the value at which the value of expression
     is required:\n");

            read(x_val);

            e_val=e.eval(x_val);

            expr Int=e.intg();

            expr Dif=e.diff();

            double bounds[2];//lower bound=bounds[0], and upper
    bound=bounds[1]

            print("Enter the lower and upper bounds:\n");

            read(bounds[0], bounds[1]);

            double Def_Int=e.intg(bounds[0], bounds[1]);

            double x_of_point;

            print("Enter the x-coordinate at which the slope is
    required:\n");

            read(x_of_point);

            double D_val=e.diff(x_of_point);

```

```
41              print("Value of expression="+e_val+"\n");
42
43              print("Expression after integration:"+Int+"\n");
44
45              print("Expression after differentiation:"+Dif+"\n");
46
47              print("Value of Definite Integral="+Def_Int+"\n");
48
49              print("Slope at given point="+D_val+"\n");
50
51              return 0;
52
53          end function main
```

Listing 42: Some functions we provide

The above program represents the core purpose of Lucas-"supporting calculus". The `intg()` and `diff()` functions just returns the expressions, when no arguments were provided and they will return values when they were provided with arguments. The function `eval()`, just evaluates the expression at that particular value.