# EcoSL

## Economical Spreadsheet Language

Language Reference Manual

Somenath Das
Satish Srinivas
Lalit K Kanteti

# 1. INTRODUCTION

The EcoSL is designed to provide a platform for users to do simple and complex operations using a spreadsheet layout on different kinds of numerical and string data. It is platform independent and stresses on writing minimal code to execute arithmetic, logical or any aggregated operations over single or multiple cells in a spreadsheet. This manual defines the language EcoSL proposed earlier in a related white paper. Regular expression notation has been used to make productions more perspicuous.

# 2. LEXICAL CONVENTIONS

## 2.1 TOKENS

There are classes of tokens: identifiers, keywords, string literals, operators, and other separators. Spaces, tabs, newlines and comments separate tokens but are otherwise ignored.

## 2.2 Data Types

### Derived Data Types:

**cell**: Cell represents a basic addressable unit in the spreadsheet.. Data taken from the input or computed through user program is stored in cells.
**group**: A group is a combination of cells. All operations valid on cell are valid on group.

### Internal Data Types:

Each cell on a sheet is considered to be an identifier in itself. Type checking is done during run time. Each cell can store data of the following data-types:

INT: If compiler encounters with sequence of integers then it is mapped to int data type.
BOOLEAN: If any cell has true or false cell automatically be assigned Boolean data type.
FLOAT: Any alphanumeric value which starts with an integer part, a decimal point, a fraction part, an e, and an optionally signed integer exponent.
STRING: A sequence of characters surrounded by double quotes " ".

## 2.3 IDENTIFIERS:

EcoSL allows identifiers to be declared and used without specifying its type. Identifiers are associated with a type during runtime. Cell identifiers are aliases of the existing cells. These aliases are typically used for repetitive call of same cell in operations. Identifiers can be a sequence of letters and digits; the first character must be an alphabet or an underscore. Identifiers are not case sensitive

## 2.4 KEYWORDS:

Following identifiers are reserved for use as keywords in EcoSL:

| | |
|---|---|
| cell | group |
| if | else |
| for | forc |
| forr | do |
| while | break |
| continue | true |
| false | return |

## 2.5 BUILT IN FUNCTIONS:

Most of the EcoSL operations are carried out using following built in functions. Names of functions are self explanatory of their operation. These functions (except conout and displaysheet) are for group operations.

| | |
|---|---|
| max | min |
| avg | sum |
| count | scale |
| addcell | displaysheet |
| linegraph | conout |

## 2.6 CONSTANTS

There are two types of constants in EcoSL: numbers and strings.

**Numbers**: A number consists of a string of digits with an optional sign '-' and an optional decimal point '.'. All built-in mathematical operations performed on numbers assume base-10 format.

**String**: A string is a sequence of characters enclosed by double quotes: "string". A double quote inside the string is indicated by two consecutive double quotes. The second double quote is ignored in the final token.

## 2.7 COMMENTS

The characters /* start what may be a multi-line comment terminated by */, while the characters // start a single-line comment.

## 3. DECLARATION

Spreadsheet can be viewed as a collection of cells. A cell represents a basic addressable unit in the spreadsheet. Each cell is addressable by it's co-ordinates relative to an origin. A cell or a collection of cells can be declared as a group.

EcoSL does not mandate the user to declare an identifier before using it. User can use a variable without declaring it. Adding to the flexibility the user does not need to provide a type for any identifier he would use. Type checking for all identifiers is done at the run time.

The syntax to be followed is described by using a regular language below:

declaration:

*declaration-spec init-declaration-list[opt];*

declaration-spec:

cell | group

init-declaration-list:

declarator

| declarator, init-declarator-list

declarator:

co-ordinate initializer[opt]

initializer: constant-expression

co-ordinate: *identifier_alias*

|*identifier_alias [ udigit : udigit ]*

| *identifier_alias[udigit..udigit : udigit .. udigit]*

|*identifier_alias[udigit : udigit .. udigit]*

| *identifier_alias[udigit .. udigit : udigit]*

Constant-expression

*expression*

identifier_alias:

*identifier* | *ε*

udigit:

*[0-9]*$^{+}$*[0-9]**

# 4. EXPRESSIONS

## 4.1 PRIMARY EXPERSSION

Primary expression includes identifier, constant and function calls.

**Identifier**: An identifier itself is a left-value expression. It will be evaluated to some values bounded to this identifier.

**Constant**
A constant is a right-value expression, which will be evaluated to the constant itself.

**Function call**
A function call consists of a function identifier, followed by a list of arguments enclosed by ( ). The list of arguments contains zero or more arguments separated by a comma ",". Each argument is an expression. Function call is a right-value expression.

## 4.2 ARITHMETIC OPERATORS

Arithmetic operators take primary expressions as operands.

**Unary arithmetic operators**
Unary operators '+' and '-' can be prefixed to an expression. '+' operator returns the expression itself whereas the '-' operator returns the negative of the primary expression. They are applicable to all the identifiers except groups.
Example: –14 or +5

**Multiplicative operators**
Binary operators '*', '/', and '%' indicate multiplication, division, and modulo, respectively. They are grouped left to right. They are applicable to all the identifiers except groups.
Example: 18 * a or 12 / 3 or 63 % 5 (a is an identifier)

**Additive operators**
Binary operators '+' and '-' indicate addition and subtraction, respectively. They are grouped left to right. They are applicable to all the identifiers except groups.
Example 19 + 4 or 14 – b (b is an identifier)

**Relational operators**
Binary relational operators >=, <=, ==, !=, > and < indicate whether the first operand is greater than or equal to, less than or equal to, equal to, not equal to, greater than, or less than the second operand, respectively.
Example. a != 12 or b == 4 or c >= 5 (a, b, c are identifiers)

**Assignment operators**
Assignment operator in EcoSL is =. It is associative from right to left. Assignment operator requires a modifiable lvalue as their left operand. The type of an assignment expression is that of its unqualified left operand. The result is not an lvalue. Its value is the value stored in the left operand after the assignment.

**Logical operators**
Logical operators take relational expressions and numbers as operands. They have the lowest precedence. In EcoSL '= =' (is equal to), '&&' (and), '!=' (is not equal to) and '| |' (or) are logical operators.

### 4.3 Productions in Expressions

expression:

*primary*
*- expression*
*expression binop expression*
*lvalue asgnop expression*

primary:

*co-ordinate*
*constant*
*primary ( expressionlistopt )*

lvalue:

*co-ordinate*
*(lvalue)*

binop:

* / %
+ -
< > <= >=
== !=
&&
| |

asgnop:

=

# 5. STATEMENTS

A statement is a complete instruction to the computer. Statements are basically elements of a program. A sequence of statements will be executed sequentially, unless the flow-control statements indicate otherwise.

### 5.1 Statements in { and }

A group of zero or more statements can be surrounded by { and }, in which case they altogether are treated as a single statement. This is true of all function calls.

### 5.2 Assignments

An assignment is in this form:
expression = expression ;
where ; is the terminator of this assignment statement.

### 5.3 Conditional statements

A conditional statement is in form:
if (expression ) statement
or
if ( expression ) statement else statement

Conditional statements choose one of a set of statements to execute, based on the evaluation of the expression. The expression is the controlling expression. For both forms of the *if* statement, the first statement is executed if the controlling expression evaluates to nonzero. For the second form, the second statement is executed if the controlling expression evaluates to zero. An *else* clause that follows multiple sequential *else-less if* statements is associated with the most recent *if* statement in the same block.

### 5.4 Iterative statements

Iterative statements are basically loops. There are two kinds of loops, for and while.
*For statement*

The *for* statement has the following forms:

*for ( expression ; expression ; expression ) statement*

The first expression specifies initialization for the loop. The second expression is the controlling expression which is evaluated before each iteration. The third expression often specifies incrimination which is evaluated after each iteration.

*forc ( co-ordinate$_1$ asgnop co-ordinate$_2$ co-ordinate$_3$) statement*
This is specialized for loop with just one expression which is defining the range of cells in a column. Values of co-ordinate$_1$ are operated against values defined from starting cell of *co-ordinate$_2$* to destination cell *co-ordinate$_3$*

*forr ( co-ordinate asgnop co-ordinate co-ordinate) statement t*
This is again specialized loop similar to above one with exception that it runs across a row.

*While statement*
The while statement is a general iterative statement. It is in this form:

while ( expression ) statement

The expression is evaluated at the start of each loop. If the expression is evaluated to false, the loop will be skipped. An infinite loop will be introduced if the expression will always evaluate to true and a break statement is not used inside the loop.

*Break statement*
The break statement will break the inner-most or labeled iterative statement. This statement consists of keyword break then followed by a ';'.

*Continue statement*
The continue statement will end the current iteration of the innermost or labeled iterative statement and proceed to its next iteration. This statement consists of keyword continue then followed by ';'.

### 5.5 Function invocation and return

*Function call*
Different from other expressions, a function call followed by ';' can be a single statement.

*Return statement*
The return statement is used inside the function definition body, in order to return from the function at that point. It can be followed by an optional expression, for the return value, and ';'.

### 5.6 Productions in Statements

statement:

*expression ;*
*{ statementlist}*
*if ( expression ) statement*
*if ( expression ) statement else statement*
*for ( expression ; expression ; expression ) statement*
*forc ( co-ordinate asgnop co-ordinate co-ordinate) statement*
*forr ( co-ordinate asgnop co-ordinate co-ordinate) statement*
*while ( expression ) statement*
*break ;*
*continue ;*
*return ;*
*return ( expression ) ;*

statement-list:

*statement*
*statement statement-list*

## 6. INPUT

User should provide input data in a CSV file. Input can be either a single file or can be multiple files. The input file should consist of data on which the user intends to run his program and carry out operations

## 7. OUTPUT

As an output, user can opt for:

- A graph where the user data are matched and a line graph is drawn on the screen.
- A sheet where the data are displayed in tabular form.
- File where the output data are stored in a CSV file.

## 8. SCOPE AND NAMESPACE

All variables are global scope in EcoSL. This language has one namespace.

## 9. ERRORS

All kind of error messages will be displayed in console.

# Appendix I

**LEXER FOR ECOSL:**

DIGIT: [0-9]

LETTER: [a-z A-Z]

OPERATOR-CHARACTER: + | - | * | / | = | < | > |% | ! | <= | >= | == | <> |
|| | &&

WHITESPACE: space | tab | newline

GRAPHIC-CHARACTER: OPERATOR-CHARACTER | DIGIT | LETTER
| : | ; | [ | ] | _ | , | " |

COMMENTS: // (GRAPHIC-CHARACTER)*newline
| START_COMMENT (GRAPHIC-CHARACTER)* END_COMMENT

START_COMMENT: /*
END_COMMENT : */

KEYWORDS: displaysheet | group | forc | forr | for | cell | addcell | max | min | avg |
scale | linegraph | if | else | while | do | break| continue | true |
false | count | conout | forr | sum | return

# Appendix II

## SAMPLE PROGRAM FOR ECOSL

```
// each cell is initialized with the value on the RHS. It will automatically assign the data
// type

cell x1[5:5] = 203;
cell x2[2:3] = 223;
cell x3[3:4] = 45.93;
cell x3[1:1] = "Srinivas";

// another definition

cell [7:1] = 56;                // No identifier. Only the cell is defined with a data type and value


/* Regular arithmetic operations */

a = 34.00;
b = 99.00;
c = a + b;
d = b – a;


/* Cell operations */

group col;                      // definition of a group of cells


/* In the for loop below, we want to check if the value of each cell addressed by
 * [x:1], viz., a column vector, is greater than say a const 10. It will then be
 * grouped into the column vector g.
 */

forc ( [x:1] = [1:1] [20:1] ) {         // this loop will start from 1:1 and go till 20:1
      if ( [x:1] > 10 ) {               // [x:y] is the value of the cell at x,y
          addcell(col, [x:1]);
        }
 }

/*
 * To build a row vector we will do something similar as above
 */

group row;

forr ( [1:y] = [1:1] [1:20] ) {
     addcell(row, [i:y]);          // Function overloading for addcell
     }
 }
```

```
/*
 * Now let's plot a graph.
 * First it will check the dimension of the row, column vectors and then plot a line graph
 */

 linegraph ( row , col );


cell a[20:30] = max(col);          // Returns the cell which has the greatest value
cell b[20:31] = min(col);          // Returns the cell which has the least value
cell c[20:32] = avg(col);          // Returns the average
cell d[20:34] = scale(col,10);     // Scale the values of the cell in the group

displaysheet();                    // It will display the sheet in tabular form
```