easel

LANGUAGE REFERENCE MANUAL

Manager | Danielle Crosswell | dac2182
Language Guru | Tyrus Cukavac | thc2125
System Architect | Yuan-Chao Chou | yc3211
Tester | Xiaofei Chen | xc2364

Table of Contents

1. Introduction **3**
1.1. A Note on Notation and Conventions 3
2. Keywords, Comments, and Whitespace **3**
2.1. Keywords 3
2.2. Comments and Whitespace 3
3. Identifiers and Data types **4**
3.1. Names 4
3.2. Primitive data types 4
3.2.1 Primitive Variables 4
3.2.2. Primitive literals 4
3.2.2.1. Integer Literals 4
3.2.2.2. Floating Point Literals 5
3.2.2.3. Boolean Literals 5
3.2.2.4. Pix Literals 5
3.2.2.5. Unary operators on numeric values 5
3.3. Non-primitive data types 6
3.3.2. Non-primitive literals 6
3.3.2.1. Array and N-Ple Literals 6
3.3.2.2. Function Literals 6
4. Expressions **6**
4.1. Base Expressions 6
4.2. Postfix Expressions: array indices, function evaluation, and unary operators 7
4.3. Mathematical Expressions 7
4.4. Equivalence and Comparison 8
4.5. Logical Expressions 9
4.6. Assignment 9
5. Declarations **9**
5.1 Variable, Array, and N-Ple declaration 9
5.1.1 Simple Variable Declarations 9
5.1.2 Array Declarations 9
5.1.3. N-Ple Declaration 10
5.2. Scope 10
6. Statements **10**
6.1. Condition and Loop Statements 10
6.1.1 if statements 11
6.1.2 if else statements 11
6.1.3. Loop Statements 11
6.1.3.1. for statements 11
6.1.3.2. while statements 11
6.1.3.3. do while statements 12
7. Functions **12**
7.1. Function Declarations and Definitions 12
7.1.1 Function Declaration without Definition 12
7.2. Function Literals / Anonymous functions 13
7.3. Invoking functions 13
7.4. Built-in Functions 13

1. Introduction

The primary purpose of easel is to create art using mathematics. Our language provides a means for programmers to visualize mathematical data and functions. The programmer is able to create visual representations using colors represented as pixels to have more aesthetically pleasing and easily understandable visualizations. The goal of easel is to create a simple interface for mathematical data and functions to be represented with the use of primitive data types and standard library functions.

easel is capable of performing both basic and advanced mathematical expressions, such as trigonometric and logarithmic functions using built-in functions and simple operators. Because the creation of images is the central feature of our language, primitive data types include "canvas" and "pix" in addition to the array data type to ease the drawing of images.

In easel, functions are the core of the language. They can be passed as parameters, returned as values, and declared anonymously.

1.1. A Note on Notation and Conventions

Because of several syntactical parallels with C, much of the notation is similar to *The C Reference Manual* by Dennis M. Ritchie, particularly with regards to naming conventions and structures used in C's grammar.

2. Keywords, Comments, and Whitespace

2.1. Keywords

The following are keywords reserved by the language and are not permitted as Names.

bool	string	void	graph
false	true	do	view
float	else	for	red
int	function	while	green
null	if	draw	blue
pix	return	drawout	

2.2. Comments and Whitespace

Comments within a program are ignored and delineated by
`/*[text]*/`
Hence, all comments must have an opening "/*" and a closing "*/" which is then skipped by the parser.

There must be at least one space between tokens in order to separate them unless a separator (",", ".", or ";") is between them. A given token is therefore not allowed to have whitespace within it, lest it be interpreted as a different token. Otherwise, whitespace is ignored.

3. Identifiers and Data types

3.1. Names

```
name -> [a-zA-Z_][0-9a-zA-Z_]*
```

Names are used to identify variables of data types as well as functions. They are allowed to be a sequence of alphanumeric characters and underscores, excepting those that match with reserved keywords, which will be interpreted as keywords in accordance with the grammar rules to follow. A name must begin with a letter or underscore, and upper and lower-case letters are not considered equivalent

3.2. Primitive data types

3.2.1 Primitive Variables

Primitive variables can be of the following 4 data types:

`int`	32-bit (or default machine architecture size) 2's complement number values
`float`	32-bit
`bool`	1-bit true or false values. Can be represented using "true" or "false" keywords or simply "1" or "0" are acceptable
`pix`	24-bit integer values. A pix stores a color value from 0-16777216.

Integers may be used anywhere floats can be used and conversion is automatic. That said, a float value cannot necessarily be used anywhere an int value is expected.

3.2.2. Primitive literals

```
primitive-literal -> pix-literal
primitive-literal -> boolean-literal
primitive-literal -> float-literal
primitive-literal -> integer-literal
```

Primitive literals of the types listed above can take the following forms.

3.2.2.1. Integer Literals

```
integer-literal -> integer-literal-hex
integer-constant -> integer-literal-dec
integer-literal-dec -> number-dec
integer-literal-hex -> number-hex

number-hex -> #[0-9a-f]+
number-dec -> [0-9]+
```

Thus, Integers can take the form of decimal notation or hex notation, which is noted by the "#" symbol followed by a number.

3.2.2.2. Floating Point Literals

```
float-literal -> integer-literal . integer-literal float-literal-exponent
float-literal -> . integer-literal
float-literal -> . integer-literal float-literal-exponent
float-literal -> integer-literal float-literal-exponent
float-literal -> integer-literal

float-literal-exponent -> e integer-literal
float-literal-exponent -> e + integer-literal
float-literal-exponent -> e - integer-literal
```

Floating point constants in decimal form consist of an integer value followed by an optional decimal point and other integer value (representing the fraction) as well as an optional exponent. If the first integer value does not exist, then there must be a decimal point followed by an integer value. An exponent may follow either the first integer value, or the decimal and second integer value, or both. No hex values are permitted in this representation.

3.2.2.3. Boolean Literals

```
boolean-literal -> true
boolean-literal -> false
```

Boolean literals are simply true or false and correspond to integer values of 1 or 0. Boolean values in general will be “upgraded” to integer or float status if placed into contexts of mathematical expressions.

3.2.2.4. Pix Literals

```
pix-literal -> { integer-literal , integer-literal , integer-literal }
```

Integers between 0 and 16777216 (i.e. any unsigned 24-bit integer in either decimal or hex form) can be cast as pix literals. However, pixel literals are perhaps more easily expressed using “list notation” of three 8-bit integer values. For example: {255,0,0} or {#ff,#00,255}. Each item in the list represents a different color (Red, Green, and Blue).

3.2.2.5. Unary operators on numeric values

Integer and floating point values are able to be given specific signs, positive or negative, by preceding them with a unary operator + or -

```
+ numeric-expression
- numeric-expression
```

Additionally, pix and boolean values (including those arrived at from expressions) can utilize the unary operator “!” to flip their values (in the case of pixels, all of the bits are flipped).

```
unary-expression -> ! unary-expression
```

Note that the “not” operator when performed on an integer != 0 will return false and vice-versa.

3.3. Non-primitive data types

Variables storing a reference to a piece of data (rather than the data itself) and built using the above-mentioned primitive data types are created using the following types:

`array` a data structure to store a sequence of adjacent values of a given data type. Arrays can be nested, that is n-matrices are possible.
`function` a reference to a function instance.
`n-ple` a data structure storing various values that are added together to memory as they are defined.

3.3.2. Non-primitive literals

The compound types named above can also be found as literals within statements.

3.3.2.1. Array and N-Ple Literals

Both arrays and n-ple's can be initialized using an n-ple list:

```
n-ple-literal -> {parameter-list}
```

An n-ple literal is essentially a list of values that make up the parameters of a function. This can include all mathematical expressions, variable names, function names, and anonymous functions as well.

3.3.2.2. Function Literals

Functions are considered first-class objects (although not expressions) in easel and are therefore able to be referred to as literals in their own right:

```
function-literal -> function type (function-parameter-list) return-region
```

This is the same form as anonymous functions (see section 7.2). Function-literals can be used in many contexts, including as parameters to other functions or simply as statements to be executed immediately (both are discussed in the "Statements" section and the "Functions" sections respectively).

4. Expressions

Expressions can take many forms but at their essence they return mathematical values.

4.1. Base Expressions

```
base-expression -> primitive-literal
base-expression -> name
```

Base expression gives the value of primitive literals or variables of primitive data types.

Additionally, any parenthesized expressions are evaluated first.

```
base-expression -> ( base-expression )
```

4.2. Postfix Expressions: array indices, function evaluation, and unary operators

Once the base expression has been evaluated, any postfix operators are applied (including all of the unary operators). Of key importance for easel's functionality is the array subscript operator ("[ ]"):

```
postfix-expression -> postfix-expression [ base-expression ]
```

A key distinction between easel and other programming languages is that arrays are indexed using a Cartesian model, with an n-size array having a max index of ⌈n/2⌉ and a minimum index -⌊n/2⌋. For convenience, arrays and matrices also have a "." operator

```
postfix-expression -> postfix-expression .
```

This allows one to access an array's size (or total number of elements) and maximum and minimum index:

```
postfix-expression.size
postfix-expression.max
postfix-expression.min
```

n-ples have access to the "size" value, but are otherwise accessed using array notation starting from index 0.

Pixel values may also use the "dot" operator:

```
pixel_variable . color_key
```

where "color_key" is any value of the type "red", "green", or "blue". It returns an 8-bit integer value corresponding to the intensity of the color.

pix x= #008844
int z = x.red /* z=0 */
int y = x.blue /* y=68 or #44 in hex */

Moreover, named functions are called using parentheses and an optional parameter list:

```
postfix-expression -> postfix-expression ( parameter-list )
postfix-expression -> postfix-expression ( )
```

Functions and their parameters are covered in more depth in the Function section.

Other unary postfix-expressions include:

```
postfix-expression -> postfix-expression ++
postfix-expression -> postfix-expression --
postfix-expression -> postfix-expression //
postfix-expression -> postfix-expression **
postfix-expression -> postfix-expression ^^
```

Each of which is shorthand for a binary operation being performed on a variable with itself.

4.3. Mathematical Expressions

Mathematical expressions are evaluated in the standard order of operations: parentheses (seen earlier), exponents, multiplication, division, addition, and subtraction. These operators are left-to-right associative as in standard mathematics.

The exponential expression is taken to the power of the given unary-expression:
exponential-expression -> exponential-expression **^** *unary-expression*
exponential-expression -> unary-expression

Once any exponents have been taken into account, multiplication and/or division can occur:
multiplication-expression -> multiplication-expression ***** *exponential-expression*
multiplication-expression -> multiplication-expression / exponential-expression
multiplication-expression -> exponential-expression

These operations are then followed by addition and subtraction:
addition-expression -> addition-expression **+** *multiplication-expression*
addition-expression -> addition-expression **-** *multiplication-expression*
addition-expression -> multiplication-expression

Where do these rules leave matrices? A key feature of easel is to provide matrix multiplication. A matrix can be multiplied by a scalar; more importantly, matrices can be multiplied by other matrices assuming that they are 1st matrix column and 2nd matrix row compatible. Additionally, two same-sized matrices can be added to or subtracted from one another.

As for n-ples, provided a user is using the name of an n-ple rather than accessing a particular value, multiplication and division are inapplicable, but addition and subtraction allow for an expression to be concatenated to the end of the n-ple.

```
nple n = {1,2,3}
n = n+ 4; /* n={1,2,3,4} */
```

4.4. Equivalence and Comparison

Expressions are compared before being tested for equality and are evaluated left-to-right:
relational-expression -> relational-expression < addition-expression
relational-expression -> relational-expression > addition-expression
relational-expression -> relational-expression <= addition-expression
relational-expression -> relational-expression >= addition-expression
relational-expression -> addition-expression

equality-expression -> equality-expression == relational-expression
equality-expression -> equality-expression != relational-expression
equality-expression -> relational-expression

In the case of arrays (or matrices) the size of the array or outer array of a matrix are compared to determine results. Functions, whether anonymous or named, cannot be compared and will return an error (this is also invalid syntax as functions are not considered expressions in easel).

4.5. Logical Expressions

Boolean values can be evaluated using the logical operators AND (“&&”) and OR (“||”):

```
logical-AND-expression -> logical-AND-expression && equality-expression
logical-OR-expression -> logical-OR-expression || logical-AND-
expression
```

“&&” is given precedence. In the case of numbers as previously described, all values not equal to 0 are considered “TRUE” Boolean values and all values equal to 0 are considered “FALSE”.

4.6. Assignment

An assignment for base expressions occurs in the following manner:

```
assignment-expression -> unary-expression = logical-OR-expression
```

A variable can be set to the value of any of the mathematical expressions. Additionally, arrays and n-ples can be assigned values using n-ple literals:

```
assignment-expression -> unary-expression = n-ple-literal
```

Because n-ple literals take parameter-list values, they can also include nested n-ple literals. So an n-matrix can be defined using n-dimensional n-ple literals as well.

5. Declarations

5.1 Variable, Array, and N-Ple declaration

5.1.1 Simple Variable Declarations

```
type name;
```

int var; /* declares variable of type int called var */

Primitive variables are declared simply by providing the variable’s type followed by its name. Declaring a variable will not automatically allocate space for the variable-this only occurs for simple variables during the definition of the variable.

Definition can happen within a separate statement, or at the same time as declaration.

int var = 5;
is valid syntax and will allocate space and assign a value for the variable.

5.1.2 Array Declarations

Arrays are declared using a type followed by the name and bracket notation:

```
compound declaration[expression]
compound declaration -> type name
```

int array_example[3+5]
float matrix_example[8][3.2+8]

Values within brackets are numerical expressions but are cast as integers, rounded down if floating-point, by the compiler. An array must know its size at declaration in order for the declaration to be valid.

As with simple variables, space is not allocated for the array until at least one value of the array has been defined. In most cases, setting an array equal to an empty nple will suffice, as in:
array={}
matrix={}
is acceptable. All values found within the array or matrix will be initialized to 0.

5.1.3. N-Ple Declaration

nple n; /* declares a new n-ple n */

The n-ple is then defined using an n-ple literal, which may or may not be the empty set:
n = {}

5.2. Scope

Scope is best explained with a brief description of program structure. A given program consists of the global space, wherein any statement can be executed, or a function can be defined.

The scope of a variable in easel is defined to be within the function in which it is declared. Regions within control or loop statements have access to all other variables declared within the same function.

The global space is a relative no-man's-land of variables that can only be used by functions to whom they are passed as arguments. They are otherwise inaccessible to internally defined or called functions.

6. Statements

A basic statement is of the form:

```
declaration ;
expression ;
anonymous-function ;
```

A general statement can be a declaration (which may include an expression), an expression by itself, or, should an anonymous function (see section 7.3) be made as a statement, it will be executed immediately.

In addition to the general statements above, easel features both control statements and loop statements for iterative processes.

6.1. Condition and Loop Statements

```
statement -> condition-statement;
statement -> loop-statement;
```

The following types of control statements make use of a list of other statements called a region:

```
region -> { statement-list }
```

Regions are defined as having their own scope, and any variables declared within the region will be deleted upon completion of the execution of the statement list. They will, however, have access to any variables declared within their calling function.

Because regions are themselves lists of statements, nesting either the control or loop statements defined below is perfectly acceptable.

Note that both condition and loop statements must end in a semi-colon.

6.1.1 if statements

```
if ( expression ) region ;
```

If statements provide a condition which evaluates to true or false. If true, a sequence of statements in a region (bracketed on the left and right sides) will execute. If the expression is false, the region is skipped and the next statement will proceed to execute.

6.1.2 if else statements

```
if ( expression ) region else region ;
```

If-else statements offer an additional path following evaluation of the expression: if an expression evaluates to true, the first region is executed and the second is ignored. If the expression evaluates to false, the second region is executed and the first is ignored. Following either of these two events, the next statement following the if-else statement will proceed to execute.

6.1.3. Loop Statements

The following statements execute repeatedly until a given condition evaluates to false.

6.1.3.1. for statements

```
for ( expression ; expression ; expression ) region
```

For statements begin with an expression (meant to declare and initialize an iterator variable), run a particular region of statements, evaluate another expression (meant to be a form of iterator) and then check to see if a given condition is true (usually if the iterator has reached a certain threshold).

6.1.3.2. while statements

```
while ( expression ) region
```

While statements first check to see if a given expression evaluates to true and if so, executes the region. The expression is then evaluated at the conclusion of every iteration and if it continues to be true, the region continues to be executed. When the expression evaluates to false, the region is skipped and the following statement is executed.

6.1.3.3. do while statements

```
loop-statement -> do region while ( expression )
```

Do-while statements immediately execute a given region. The expression is then evaluated at the conclusion of every iteration and if it holds true, the region continues to be executed. When the expression evaluates to false, the region is skipped and the following statement is executed.

7. Functions

Functions form the foundation of easel's functionality. A few important things to note:

1. Elements are passed by *reference* not by value. This is meant to encourage the use of functions within easel as opposed to purely sequential programming, as a function will not otherwise have access to any outside variables.
2. Functions may return a void data type, but void is not a valid data type for any other value.
3. Functions may be passed as arguments to other functions, either as a function literal / anonymous function or simply as a named entity.
4. Functions may be called recursively, and can call any other functions defined in the global scope.
5. Functions may be overloaded. Multiple functions may have the same name provided that the function declaration (i.e. return type, parameters) is different.
6. If two functions within the global scope have the same name and declaration, the compiler will throw an error.

7.1. Function Declarations and Definitions

When declared as part of the normal statement flow of a program, functions must be defined at declaration.

```
function -> function-declaration return-region
```

Where a function declaration consists of**:**

```
function type name (function-parameter-list)
```

And a function-parameter-list is defined as:

```
function-parameter-list -> function-parameter-list, declaration
function-parameter-list -> declaration
```

Function parameters are essentially composed of a list of declarations, including declarations of other functions.

The return-region, which is the main element of a function, follows the form:

```
{ statement-list return expression ; }
```

A "return region" is a bracketed list of statements followed by a return expression.

7.1.1 Function Declaration without Definition

A function can only be declared without definition when the function itself is being passed as an argument to another function. This declaration follows from below:

```
function type name (function-parameter-list)
```

function void my_function (int x, int y, function int operate(int w, int z));

The “operate” function is thus declared as a parameter. Any function passed as the operate parameter must have a matching return type and argument list or the program will be rejected by the compiler.

7.2. Function Literals / Anonymous functions

Functions may also take the form of function literals. They follow the format:
function *type* **(***function-parameter-list***)** *return-region*

function pix (pix x) {return !x}

These functions differ from the standard form in that they are not named.

7.3. Invoking functions

Functions can be called anywhere during statement flow. A function is called by using its name and passing it parameters.
postfix-expression **(** *parameter-list* **)**

function int add (int x, int y){return x+y}
pix x = add(3,2) /* x=5 */

To pass another function to a function and execute, a function signature must be provided as a parameter:
type name **(***parameter-list***)**

function int im_a_parameter(int u, int v)={return u+v}
my_function(5, 2, int im_a_parameter(int, int)); /* value=7 */

Both the return type as well as the parameters must be provided in order to correctly identify the function being passed. If one of these values is missing or the function has not been defined, then the compiler will throw an error message.

If an anonymous function literal is written as a statement (with the requisite semicolon) it is an anonymous function and will be invoked immediately as part of normal execution. In this instance, the anonymous function must have an expression rather than declaration in its argument-list. (It uses the “parameter-list” reduction from the example above rather than the “function-parameter-list” used in section 7.2).

int x=18;
function pix(x) {return x-2}; /* Returns a pixel with value 16 */

7.4. Built-in Functions

Whereas many other languages have some mechanism of printing strings, easel’s built-in method of output is drawing a canvas to the monitor.

The signature of the function responsible for the drawing is:
draw(pix name[][], int x, int y);

The argument is a 2-dimensional pixel matrix as well as an x and y value that determines where the top left corner of the canvas window is placed. An easel program is ultimately about creating this matrix of pixels using various functions or coding prowess to create a visual representation on screen.

In the same vein, a filename may be provided to easel as a command line argument via stdin on unix platforms that will allow a user to use the "drawout" function as well as or instead of the draw function.
drawout(pix name[][]);

For an easel file drawout.es consisting of:

```
pix canvas[100][100] = {}
drawout(canvas);
```

Calling the compiled program drawout via the terminal and feeding it the desired filename would create a jpeg file consisting of a 100 x 100 white canvas of pixels.

```
$ echo "whitecanvas".jpeg | ./drawout
```

This will output the drawing as a jpg file for future use elsewhere.

Out of the box, easel also offers a number of mathematical functions. These include:
float tan(float x)
float sin(float x)
float cos(float x)
float log(float base, float value)
which allow the user to calculate more complicated geometric functions than the standard arithmetic operators.

Finally, easel has a random number generator, which generates a floating point value between 0 and 1 inclusive. This value can be multiplied, added, or subtracted to in order to provide a random value within a desired range.
float rand()