EZ-ASCII: Language Reference Manual

Dmitriy Gromov (dg2720), Feifei Zhong (fz2185),

Yilei Wang (yw2493), Xin Ye (xy2190), Joe Lee (jyl2157)

Table of Contents

1 Program Definition

This reference manual defines the language of EZ-ASCII.

The structure of an EZ-ASCII program source file consists of expression statements and functions. A `main()` function may be optionally specified to denote the main entry point of the program.

```
< global expressions >

< function declarations >

fun main() {
      <main program code>
}
```

2 Lexical Conventions

2.1 Tokens

There are six types of tokens: identifiers, keywords, constants, string literals, operators, and other separators. Blanks, horizontal, and vertical tabs, newlines, formfeeds, and comments as described below (collectively, "white space") are ignored except as they separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

2.2 Comments

When a `//` symbol is encountered, the `//` symbol and the rest of the line is considered a comment and is ignored by the compiler.

```
// This is a comment line
img[x1, y1] <- 1; // This is another comment
```

2.3 Identifiers

An identifier is a sequence of letters and digits. The first character must be a letter; the underscore _ counts as a letter. Upper and lower case letters are different.

2.4 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

`blank`	`load`
`else`	`main`
`false`	`map`
`Fun`	`out`
`for`	`if`
`return`	`true`
`include`	

2.5 Constants

2.5.1 Boolean Constants

A boolean constant is either `true` or `false` (case-sensitive).

2.5.2 Integer Constants

An integer constant consists of a sequence of digits.

```
i <- 213
```

There are 4 labeled integer constants to define shifting directions

```
SHIFT_UP, SHIFT_LEFT, SHIFT_DOWN, SHIFT_RIGHT
```

Their values are 0,1,2,3 respectively.

2.5.3 String Constants

A string constant consists of a sequence of characters enclosed in double quotes "". The following characters may be used with escape sequences:

Character	Escape Sequence
newline	\n
horizontal tab	\t
single quote	\’
double quote	\”
backslash	\\

2.5.4 Mapping Constants

An intensity mapping consists of a table mapping intensities to characters. A custom mapping can be defined using the keyword `map`:

```
map <- {I0:"C0", I1:"C1",...,IN:"CN"}
```

where each I is an intensity, and the corresponding C is the character mapped to that intensity. Any reference to the intensity mapping will refer to the most recent assignment of MAP or the default if none has been assigned.

The default mapping is a map of all printable ASCII characters ordered in ascending order based on how many pixels each character takes up in each character space.

2.6 Granularity and Intensity

A mapping must have at least two values and the granularity must be at least 2. The minimum intensity will be the least intense item in the map and the maximum will be the most intense. For intensities between `1` and `n - 1` where `n` is the size of the mapping the distance between each intensity is as close to even as possible. The formula for this is defined as follows:

```
diff = (n - 2) / ((g - 2) + 1)
```

where `n` is again the size of the map, and `g` is the granularity.

3 Meaning of Identifiers

Identifiers may refer to objects (locations in storage) or functions. A function and an object may not be referred to using the same identifier – the following is a syntax error:

```
foo <- 3
fun foo() {
      <function-body>
}
```

3.1 Types

There are four types:

- boolean
- integer
- string
- canvas

3.1.1 Boolean Type

A boolean stores one bit of information and may have the value `true` or `false`.

3.1.2 Integer Type

Integers can store 32-bits of data and are signed.

3.1.3 String Type

Strings are sequences of characters, and are bounded only by available memory.

3.1.4 Canvas Type

A canvas is the primary storage type in EZ-ASCII. All of the image modification happens on this type. Internally, it is represented as a two-dimensional array of integers referred to as intensities. This canvas can be loaded from an existing image file or it can be created manually. Additionally, a canvas has the following readable attributes: width and height in number of characters, and granularity.

There are two methods of creating a canvas in EZ-ASCII. The first is to load an existing image using the `load` built-in function, and the second is to use the `blank` built-in function (see built-in functions). In the case of loading an external image file, a custom intensity mapping may be specified to specify the granularity of the image, or the default will be used.

Various operations may be performed on canvases, including selection, movement, and masking.

4 Expressions

4.1 Unary Minus Operator

The operand of the unary – operator must have arithmetic type, and the result is the negative of its operand.

```
i <- -(1 + 4) // i assigned -5
```

4.2 Multiplicative Operators

The multiplicative operators *, `/`, and `%` group left-to-right and require their operands to be of the same primitive types.

If the operands are of integer type, then the result of the `*` operator is the product of the operands. The result of the `/` operator is the quotient of the operands and the result of the `%` is the remainder after integer division on the operands. The `/` operator results in an integer (fractions truncated).

If the operands are of any other type, a syntax error will occur.

```
i <- 2 * 3   // i assigned 6
i <- 3 / 2   // i assigned 1
i <- 2 / 3   // i assigned 0
i <- 3 % 6   // i assigned 0
i <- 5 % 3   // i assigned 2
```

4.3 Additive Operators

The additive operators + and – group left-to-right and require their operands to be of the same primitive types. The grammar is as follows:

If the operands are of integer type, then the result of the + operator is the sum of the operands, and the – operator is the difference of the operands.

If the operands are of string type, then the result of the + operator is the concatenation of the operands, and the – operator will result in a syntax error.

If the operands are of type canvas, then the result of the + operator is a new canvas where each intensity is the result of adding the two corresponding intensities from the operand canvases, truncated to the maximum mapped intensity. The result of the – operator is a new canvas where each intensity is the result of the difference between the two corresponding intensities from the operand canvases, truncated to the minimum intensity of 0.

If the operands are of boolean type, a syntax error will occur.

```
i <- 1 + 2 * 3 + 4                 // i assigned 11
j <- 5 – 3                         // j assigned 2
k <- "hello " + "world!"           // k assigned "hello world!"
k <- "hello " – "world!"           // syntax error
img3 <- img1[3:8, 2:4] + img2[,]   // img3 assigned additive layering
img4 <- img1[,] – img3             // img4 assigned difference layering
m <- k + img4                      // syntax error
n <- true + img4                   // syntax error
```

4.4 Relational Operators

The relational operators group left to right, i.e. `a < b < c` is parsed as `(a < b) < c`. The operators < (less), > (greater), <= (less than or equal), and >= (greater than or equal) all yield a boolean `true` or `false`. The two variables on either side of a relational operator must be of the same type.

4.5 Logical Negation Operator

The operand of the ~ operator must have boolean type, and the result is `true` if the value of its operand compares equal to `false`, and `false` otherwise.

```
b <- ~(3 > 2) // b assigned false
```

4.6 Equality Operators

The = (equal to) and ~= (not equal to) operators are analogous to the relational operators except for their lower precedence. For example, `a<b = c<d` is parsed as `(a<b) = (c<d)` and evaluates to `true` if `a<b` and `c<d` have the same truth-value.

4.7 Logical AND Operator

The `&&` operator groups left-to-right, returning `true` if both its operands compare unequal to `false`, and `false` otherwise. Both operands must be of boolean type, except in the case of boolean expressions used in a selection operator, in which case both operands must be of a boolean expression type that satisfies the selection operator (see selection operator).

4.8 Logical OR Operator

The || operator groups left-to-right, returning `true` if either of its operands compares unequal to `false`, and `false` otherwise. Both operands must be of boolean type, except in the case of boolean expressions used in a selection operator, in which case both operands must be of a boolean expression type that satisfies the selection operator (see selection operator).

4.9 Comma Operator

A pair of expressions separated by a comma "," is evaluated left to right.

4.10 Selection Operator

The selection operator `[]` denotes a selection on the canvas that it is applied to. When the selection operator is used on a canvas, the return value is a canvas of equal size which contains only the points of interest (rest are blank). There are multiple types of selections possible depending on different integer parameters for the selection operator, as follows:

4.10.1 Selection of a single point

identifier[x, y] – x and y are integer types which denote the x and y coordinates of a single point.

4.10.2 Selection of rectangles/slices

identifier[x1:x2, y1:y2] – x1:x2 denotes a range of rows (inclusive), and y1:y2 denotes a range of columns (inclusive).

identifier[x, y1:y2] – A horizontal slice in row x from columns y1 to y2 (inclusive).

identifier[x1:x2, y] – A vertical slice in column y from rows x1 to x2 (inclusive).

identifier[,] – Returns a new copy of the canvas (all rows and columns).

4.10.3 Selection by boolean expression

identifier[boolean expression] – selects elements with intensity that satisfy the boolean expression. Boolean expressions for the selection operator must be of the format *[condition][intensity]*, where *[condition]* may be either a relational or equality operator (`<, >, <=, >=, ~=, =`), and *[intensity]* is an integer value. Boolean expressions may be chained by a logical AND operator (`&&`) or logical OR operator (`||`).

```
img[>2 && <10] // selects intensities between 2 and 10 (non-inclusive)
               // from img as a new canvas
```

4.11 Mask Operator

The `&` operator groups left-to-right, operating on canvas types. It returns a new canvas where at any given position, the intensity is 0 if the corresponding intensity in the second operand is 0, and the intensity is the corresponding intensity in the first operand if the corresponding intensity in the second operand is greater than 0. In other words, it returns the first canvas operand, but where the corresponding areas in the second canvas are 0, the corresponding areas in the first canvas are "masked" out. Any operand type other than a canvas type is a syntax error.

```
img3 <- img1 & img2  // img3 is img1 with img2 applied as a mask
i <- 2 & 3           // syntax error
```

4.12 Arrow Operator

There are two arrow operators `<-` (left) and `->` (right), which are used for assignment and output, respectively.

4.12.1 Assignment

The `<-` left arrow operator assigns the value of the expression to its right to the variable to its left. If the variable is undefined, it is created. If the variable is already in memory, its contents are overwritten with the new value.

identifier `<-` *`expression`*

Examples:

```
canvas <- load('pic.jpg', 10); // the variable canvas holds image data
canvas <- 2;                   // the variable canvas holds an integer
for i <- 2 | i < 10 | i <- i + 1 {
...
}
```

4.12.2 Output

The `->` right arrow operator outputs the value of the variable or expression to its left to either a file specified by a filepath string to its right, or to standard output, specified by the keyword `out`. If the left operand is a variable, it must be have been assigned previously, otherwise a compiler error will result.

```
"output string" -> out;      // outputs "output string" to standard out
1 + 2 -> out;                // outputs "3" to standard out
```

If the left operand is a canvas, an intensity map may be optionally supplied to dynamically change the intensity mapping.

```
canvas -> out, render;               // outputs image canvas to standard out
canvas(map) -> "test2.txt", render; // outputs image canvas to file with new
mapping
```

`render` must be a boolean value that specifies whether or not the intensities should be converted to their corresponding characters before being printed. If render is false and it is printed to a file, the file is post pended with the extension .i. If render is true, then the file will map will be applied and the actual image will be printed.

4.13 Canvas Attribute Accessor (read-only)

The `$` operator may be appended to a canvas identifier along with one of [`w, h, g`] for width, height, and granularity, respectively, to read the attribute of interest from an existing canvas object as follows:

```
canvas <- load("test.jpg", map)
canvas$w -> [width-integer]
canvas$h -> [height-integer]
canvas$g -> [granularity-integer]
```

This puts the values of the canvas's width, height, and granularity into variables a, b, and c, respectively.

4.14 Function Calls

A function call moves program execution to the target function. The syntax of a function call is:

function-name (*identifier-list*$_{opt}$)

where *identifier-list* is defined as:

identifier

identifier-list , identifier

A function must be declared before the function call.

4.15 Include

The `include` keyword allows you to add functionality from another EZ-ASCII code file to the one you are currently working on and has the following syntax:

```
include [filepath]
```

filepath must be the location of another EZ-ASCII file. At compilation time the code included in the desired file will be copied into the file being compiled. Note that identically named global variables or function names in both files will cause compilation errors.

5 Declarations

5.1 Function Declarations

A function is declared as:

fun Function-name (*identifier-list*$_{opt}$) { *<function-body>* }

where *identifier-list* is defined as:

identifier

identifier-list , identifier

Functions act as blocks of code that can be called when desired. Functions can be optionally passed a list of input parameters which are passed by value, and the parameters will be copies of the inputs for the function body. Functions can also optionally return some value at the end of their execution. A function may also call itself recursively in its body.

```
fun foo(img) {
	tmpimg <- img[>3 || <6]
	tmpimg <- tmpimg[4:8, 3:6]
	return tmpimg
}


// recursive factorial
fun factorial(x) {
	if(x = 1) return 1;
	else return x * factorial(x - 1);
}
```

5.2 Variable Declarations

Variable declarations are declared as:

Variable-name <– *expression*

Type declarations are not required - variable types are inferred from the declaration. A variable may be set to a different value with a different type even if previously declared, e.g. the following will not result in an error:

```
i <- 3                    // i holds 3
i <- load(“test.jpg”, map) // i now holds a canvas
```

6 Statements

Except as described, statements in EZ-ASCII are executed in sequence. Statements are executed for their effect and do not have return values. They fall into several groups.

Statement:
expression-statement
selection-statement
for-statement

6.1 Expression Statement

Most statements in EZ-ASCII are expression statements, which are expressions ending with semicolons.

```
img(map) -> "test2.txt";
img[x1, y1] <- 1;
img2 <- shift(img, SHIFT_UP, 5);
```

6.2 Conditional Statement

Conditional statements allow for one of several flows of control. An `if` statement may be used with or without an `else` clause. The grammar is as follows:

`if` (*expression*) *statement*

`if` (*expression*) *statement* `else` *statement*

The expression in the `if` statement must be of boolean type, and if it evaluates to `true`, the first sub-statement is executed. In the second form, the second sub-statement is executed if the expression evaluates to `false`.

```
if(1 > 0) “true case” -> out; // “true case” is output to standard out
if(true) {
...
}

if(3 > 4) "three is greater than four" -> out;
else "the world is sane" -> out;
```

The `else` ambiguity is resolved by connecting an `else` with the last encountered `else`-less `if` at the same block nesting level.

```
if(2 ~= 2)
if(3 > 2)
else "this else binds to the second if" -> out;
```

6.3 For Statement

The `for` statement specifies looping.

`for` *expression*$_{opt}$ | *expression*$_{opt}$ | *expression*$_{opt}$ *statement*

In the `for` statement, the first expression is evaluated once, and thus specifies initialization for the loop. There is no restriction on its type. The second expression must be a boolean expression; it is evaluated before each iteration, and if it becomes `false`, the `for` is terminated. The third expression is evaluated after each iteration, and thus specifies a re-initialization for the loop. There is no restriction on its type. Any of the three expressions may be dropped. A missing second expression makes the implied test equivalent to testing a `true` constant.

```
for i <- 2 | i < 10 | i <- i + 1
{
      img[i, i] <- 3;
}
```

6.4 Return Statement

`return` *expression*$_{opt}$;

A function returns to its caller by the `return` statement. When return is followed by an expression, the value is returned to the caller of the function. A function without a return statement is equivalent to a return with no expression, and in both cases, the return value is undefined.

```
fun a(x) {
      return x + 1;
}

// boo has no return value
fun boo() {
      <body>
}
```

7 Scope and Linkage

7.1 Lexical Scope

7.1.1 Variable Scope

Parameters declared in function definitions and variables declared within function bodies have scope through the end of the function (local scope). Any identically-named identifiers declared previous to the function call are suspended until the end of the function call. If an identifier is referenced in a function body but has not been declared in the function, the identifier in global scope (all identifiers defined outside of functions) is used.

```
i <- 1

fun a() {
      i <- 3    // i is assigned the value 3 for the duration of the function
}
i -> out        // global scope – prints 1

fun b() {
      i -> out // i is undefined locally, so use global scope – prints 1
}
```

7.1.2 Function Scope

Functions have global scope. A function may not be referred to unless it has been previously declared.

```
fun a() {
      <body>
}
a()
b() // error – b undefined
```

8 System Functions

8.1 Blank

`blank` ([*width*], [*height*], [*granularity*])

`Blank` takes three integer input parameters (width and height in number of characters, and a granularity level), and outputs an empty canvas with attributes set accordingly. An empty canvas in EZ-ASCII is one such that all of the intensities are 0.

8.2 Load

```
load ( [filepath], [granularity] )
```

`load` takes a string filepath to an existing image file and an integer granularity level as inputs, loads the image file into memory, performs Floyd-Steinberg dithering and normalizes the values using granularity input, and finally returns a canvas corresponding to the image. If the file name contains the extension .i, it is assumed to be an EZ-ASCII intensity file. In this case, it will load the image directly without any processing. The latter case will throw an error if the intensities found in file are not compatible with the granularity specified.

8.3 Shift

```
shift( [canvas], [shift_dir], [dist] )
```

shift takes a valid canvas identifier canvas, a integer value `shift_dir` representing which direction to shift in and a distance to shift, `dist`. For simplicisty, shift_dir will accept only 4 possible values: SHIFT_UP, SHIFT_DOWN, SHIFT_LEFT, SHIFT_RIGHT. These values are described in section 2.5.2. The purpose of shift is to take all of the characters in on a canvas and shift them in the shift_dir direction, dist spaces.

The direction itself is implied by the names of the variables (e.g. SHIFT_LEFT means left). Dist must be greater than 0 and less than the width of the canvas if shifting left or right or less than the height if shifting up or down. The result of this function will be to return a representation of the canvas with everything shifted.

Please note, if the movement causes a character to go beyond the border of the canvas it will disappear. Shifting one column past the right edge will cause the right most edge to disappear and the second to right most column will take its place. Furthermore in this case, the left most column will be padded with intensities of 0. The analogous situation is true for all other shifting directions.