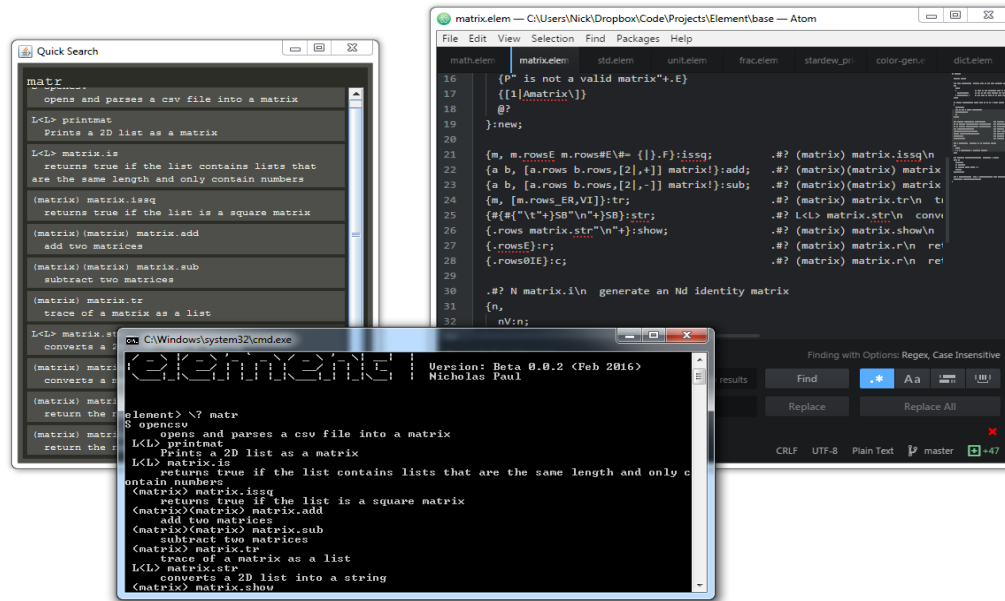


# Element

## An introduction to the Element programming language

20 June 2016



## Features

- Terse, yet readable syntax
- Modules with namespace like functionality
- Basic support for objects and data structures
- Fully loaded with a complete standard library
- macro-like pre-evaluation stack manipulation
- Functional feel: List comprehension etc.
- String Interpolation, Unicode, and special characters
- Comes pre-packaged with a feature packed GUI
- Built in plotting
- Interactive help and Documentation
- Colored printing and simple UI elements

## OVERVIEW

---

Element is a stack based programming language originally intended for code golf and programming puzzles. However, the language is very different from most golfing languages. Its support for user-defined types and macro-like function definitions allow for complex programs and data structures. It excels in cases where programs need to be written quickly.

Element comes fully-loaded with a standard library written entirely in element code. The standard library features types such as fractions, dictionaries, matrices, stacks, and more. It also features hundreds of functions for working on numerical computations, strings, plotting and file I/O.

Element also features a minimal GUI for easily writing code and working using the element language. The GUI features colored console printing, plotting, tab-completion for special characters, and most importantly, an interactive way to search QuickSearch help data.

## Table of Contents

Features.....	1
Overview.....	2
The Stack.....	6
Comments.....	6
Numbers.....	7
Booleans.....	8
Boolean operations.....	8
Comparisons.....	8
Characters and Strings.....	9
Characters.....	9
Special Characters.....	9
Unicode character literals.....	9
Named Characters.....	9
Binary and Hexadecimal Literals.....	9
Strings.....	10
String Interpolation.....	10
Long String Literals.....	11
Operators.....	13
Special Operators.....	13
The # Operator.....	13
The : operator.....	14
Assigning Variables.....	14
Assigning List Elements.....	14
Lists.....	15
Essential List Operations.....	15
Range Operator.....	15
List comprehension.....	16
Variables.....	18
Special Character Variables.....	18
Types.....	20
Verbose Type Names.....	20
Type Chart.....	21
Blocks.....	22
Block Header.....	22
Arguments.....	22
Argument Type Assertions.....	23
Local Declarations.....	23
Functions.....	24
Variable Scope.....	25
Ticks and Groups.....	26
Ticks.....	26
Groups.....	26

Modules.....	27
Namespaces.....	27
Access Variables.....	27
User Types.....	27
Object Creation.....	28
Constructor.....	28
Fields.....	29
Getting Field Data.....	29
Setting Field Data.....	29
The fields Function.....	29
String Conversion.....	30
Operator Overloading.....	31
Documenting Code.....	32
Common Practice.....	32
Functions.....	32
Modules and Types.....	32
The Standard Library.....	34
Function and Variable Definitions.....	34
Standard Functions (std.elem).....	34
Mathematical and Numerical Functions (math.elem).....	34
Golfing Functions (double_chars.elem).....	34
Units and Conversions (unit.elem).....	34
Types.....	34
Color (colors.elem).....	34
Complex Number (complex.elem).....	35
Table (dataframe.elem).....	36
Date (date.elem).....	36
Dictionary (dict.elem).....	36
Fraction (frac.elem).....	36
Matrix (matrix.elem).....	37
Queue (queue.elem).....	38
Set (set.elem).....	38
Stack (stack.elem).....	38
Other.....	39
Documentation (docs.elem).....	39
Sample Data (data.elem).....	39
Code Demo (demo.elem).....	39
The REPL.....	40
REPL Features.....	40
Time.....	40
QuickSearch.....	40
Examples.....	41
Examples.....	42
2D Vector Type Definition.....	42

Type Definition.....	42
Usage.....	42
Plot a sine wave.....	43
Explanation.....	43
Output.....	43
Plot a Fourier sine series with random coefficients.....	43
Output.....	43
Project Euler Golf.....	44
Problem 1 (15 Chars).....	44
Problem 6 (12 Chars).....	44
Problem 8 (31 Chars).....	44
Problem 34 (25 Chars).....	44
Problem 48 (10 Chars).....	45

# Element Manual

We begin with some essential knowledge regarding the Element programming language.

## THE STACK

---

Element is a stack based programming language. The stack is evaluated from left to right

```
In:  1 3 + 2 - 4 +  
Out:      4 2 - 4 +  
          2 4 +  
          6
```

## COMMENTS

---

Line comments start with a . #

Block comments are denoted using . { and . } They cannot be nested.

## NUMBERS

---

Element has three representations for numbers: integers, doubles, and BigDecimals. Numbers flow fluidly between types when needed. By default, number literals without a decimal are parsed to integers and number literals with decimals are parsed to BigDecimals.

Element uses standard mathematical operators.

```
3 4 + .# => 7
5 6 - .# => -1
2 0.5 * .# => 1.0
3 2 ^ .# => 9
```

Division converts integers to doubles.

```
6 4 / .# => 1.5
6 2 / .# => 3.0
```

- is never a unary operator unless it is [grouped](#) alone with a number literal. The group will be considered a negative number literal by the parser.

```
8 3 -1 .# is evaluated as (8 3-) 1 => 5 1
-1 .# ERROR: Empty stack at operator '-'
(-1) .# => -1
(-2.5) .# => -2.5
```

Generally, use the operator ! to negate numbers

```
1.5! .# => -1.5
```

## BOOLEANS

---

The capital letters T and F are used for true and false respectively.

### Boolean operations

Logical operators !, |, and & are defined as *not*, *or*, and *and* respectively.

```
T!      .# => false
F!      .# => true
T F &   .# => false
T F |   .# => true
```

### Comparisons

The = operator compares any two objects. If they are of the same type and have the same contents, = will return true. Append the ! Operator for a *not equals* comparison.

```
1 1 =      .# => true
1 1 !=     .# => false
"abc" "ABC" = .# => false
```

Use < for less than and > for greater than.

```
1 5 < .# => true
5 5 > .# => false
```

In order to remember the order of the arguments, imagine swapping the comparison operator and its first operand. 0 5 < is equivalent to 0 < 5 for infix parsers.

Use .< for *less than or equal to* and .> for *greater than or equal to*.

```
1 5 .< .# => true
5 5 .> .# => true
```



## CHARACTERS AND STRINGS

---

### Characters

Most characters do not need closing quotes.

```
'a      .# => 'a'  
'p'q    .# => 'p' 'q'
```

### *Special Characters*

Using a \ or a # after a single quote denotes a special character. Special characters **always need a closing single quote**.

### *Unicode character literals*

Unicode characters are written using a '\U\_\_\_\_' and need closing quotes. A space after the U is optional

```
'\U 00FF' .# => 'ÿ'  
'\U00A1' .# => '¡'
```

### *Named Characters*

Many characters have names. All names consist only of lowercase alphabetical characters. Named characters can be used like so within Element:

```
'\alpha' .# => 'α'  
'\pi'    .# => 'π'  
'\because' .# => '∵'  
'\n'     .# => <newline>  
'\t'     .# => <tab>
```

To add or override a named character from within element, use the Mk operator.

```
element> '\integral'  
SYNTAX ERROR: '\integral' is not a valid special character  
  
element> '\U222b' "integral" Mk  
  
element> '\integral'  
'∫'
```

## Binary and Hexadecimal Literals

Binary and hexadecimal integer literals are created using '#B\_\_' where B is the base (b => binary, h => hexadecimal). A space after the base id is optional. The literal does require a closing quote.

```
'#b 101001'    .# => 41  
'#h0F05'      .# => 3845
```

## Strings

Strings are created using the double quote character ".

```
"I am a string"  
"I am a string containing a newline character\n\t and a tab."
```

Strings may span multiple lines.

```
"I am a string containing a newline character  
and a tab."
```

Strings can contain special characters using \{ \_\_\_\_ }. Brackets can contain named characters or Unicode literals.

```
"Jack \{heart}s Jill"      .# => "Jack ♥s Jill"  
"sin(\{theta}) = \{alpha}" .# => "sin(θ) = α"  
"\{U00BF}Que tal?"        .# => "¿Que tal?"
```

Strings are essentially a list of characters, so any list operator that can be used on lists can be used on strings.

```
"Hello " "world!" K      .# => "Hello world!"  
['s't'r'i'n'g]          .# => "string"  
"abcde" 2 I              .# => 'c'
```

## String Interpolation

Use the \$ character within a string to evaluate the variable or statement following it.

If used with a variable name, evaluate the variable name.

```
element> 5:num;          .# Assign the number 5 to the variable `num`  
element> "I have $num apples"  
"I have 5 apples"
```

If used with a `()`, evaluate the [group](#).

```
element> "I have $(1 num +) bananas"
"I have 6 bananas"
```

If there are more than one item left on the stack, element dumps the stack inside square brackets.

```
element> 123:playera;
element> 116:playerb;
element> "The final scores are $(playera playerb)!"
"The final scores are [ 123 116 ]!"
```

If used after a `\`, keep the `$` char.

```
element> 10:dollars;
element> "I have \$$dollars."
"I have $10"
```

If used with anything else, keep the `$`.

```
element> "Each apple is worth $0.50"
"Each apple is worth $0.50"
```

Here are some additional examples:

```
element> 5:num;
element> 0.75:price;
element> "I sold $num apples for \$$price each and I made \$$ (num price*)"
"I sold 5 apples for $0.75 each and I made $3.75"

element> "Inner $(\"strings\")"
"Inner strings"

element> "Inner $(\"$a\") interpolation requires backslashes"
"Inner 1 interpolation requires backslashes"

element> "Inner-$(\"$(\\\"inner\\\")\") interpolation can be messy"
"Inner-inner interpolation can be messy"
```

## ***Long String Literals***

Long strings are entered using triple quotes. No characters are escaped within long strings. In the following code...

```
"""<div id="my_div">
    <h1>\n: the newline character</h1>
    <p>\{alpha}<p>
    <p>$interpolate</p>
</div>"""
```

...no escape characters are parsed in the output:

```
"<div id="my_div">
    <h1>\n: the newline character</h1>
    <p>\{alpha}<p>
    <p>$interpolate</p>
</div>"
```

## OPERATORS

---

Operators come in a few forms:

- Most non-alphabetical characters
- Uppercase letters
- Any character (except digits and lowercase letters) directly following a .
- Any character directly following an M. These characters are usually **math** or **misc.** related.

Element programs can be so short because operators are overloaded to perform several different tasks based on the argument types. The table below contains a few operators to illustrate how they work. The definition of every operator is omitted from this manual. To browse the definition of operators, use the [QuickSearch](#) feature of element.

Arg Types	Operator	Function	Input	Output
Any	—	duplicate the item on the top of the stack	1 —	1 1
Any	;	remove the item on the top of the stack	1 2 ;	1
Number	!	negate	4 !	-4
List	!	reverse	"abc" !	"cba"
Bool	!	boolean not	T !	F
Number	R	range	3 R	[1 2 3]
Any	.T	prints the type identifier (as a char) for the item	3 .t	'N
Number	Ms	trigonometric sine	pi Ms	0.0
Number	Mq	square root	16 Mq	4.0

## Special Operators

### *The # Operator*

# is a very powerful infix operator. It's primary function is map. It takes the arguments from its right side and maps them to the list on the left side.

```
[1 2 3] # {1 +} .# => [2 3 4]
```

If a block is not given on the right side, # will collect items until an operator or variable is encountered.

```
.# Same as the previous example  
[1 2 3] # 1 + .# => [2 3 4]
```

# will also collect items on its left side until a list is hit. It will add these items to the front of the block being mapped to.

```
.# Also the same as the previous line  
[1 2 3] 1 # + .# => [2 3 4]
```

Below are some more examples of how to use #

## ***The : operator***

The : operator is generally used for assignments.

## ***Assigning Variables***

(see [variable](#) section)

## ***Assigning List Elements***

List assignments have the form `list index : new_value` or `list : new_value`. Below are some examples.

```
element> 6R:list  
[ 1 2 3 4 5 6 ]  
  
element> (list 2):100  
[ 1 2 100 4 5 6 ]  
  
.# Copy the jth element of list into the ith location of list  
(list i : (list j I)) : list;
```

If no index is provided, map the RHS to the LHS. **NOTE:** This syntax is hardly ever used. Use the # operator (which has the same mapping functionality) instead.

```
element> list:{1 +}  
[ 2 3 101 5 6 7 ]  
  
element> list:45  
[ 45 45 45 45 45 45 ]
```

## LISTS

---

List literals are created using square brackets and do not need commas. Literals are first evaluated as their own stack. The results remaining on the stack become the list items.

```
[1 2 3 4 5] .# Do not use commas
[1 2 + 7 2 - 3!] .# => [3 5 -3]
```

List literals can grab items from the outer stack using the format `... [num| ...]` where `num` is an integer literal.

```
element> 1 2 3 4 5 [3| 6 7 8]
1 2 [3 4 5 6 7 8]

element> 'h 'e [2|'l 'l 'o]
"hello"

element> "a" "b" [2|]
["a" "b"]
```

## Essential List Operations

Joining 2 lists

```
[1 2 3] [4 5 6] K .# => [1 2 3 4 5 6]
```

Lists can be indexed using the `I` or `. I` operators. The `. I` operator will leave both the list and the element on the stack.

Arg Type	Function	Input	Output
Number	choose the nth item from the list (starting from 0)	[1 2 3] 1 I	2
List	use each item in the second list to index the first	"abc" [1 1 2] I	"bbc"
Block	filter the list. Take all items that satisfy the block	[1 1 2 2] {1=} I	[1 1]

## Range Operator

**One item:** create a range from 1 (or 'a') to that number.

```
10 R      .# => [1 2 3 4 5 6 7 8 9 10]
'd R      .# => "abcd"
```

**Two items:** create a range from the first to the second.

```
[5 10] R      .# => [5 6 7 8 9 10]
['z 'w] R      .# => "zyxw"
```

**Three items:** create a range from the first to the third using the second as a step.

```
[0 0.5 2] R      .# => [0 0.5 1.0 1.5 2.0]
```

## List comprehension

When commas are used inside of a list literal, the list is created using list comprehension. List comprehension follows the format `[range, map, filter1, filter2, ..., filterK]`. The range section is evaluated like the R operator. When the list is evaluated, the sections are evaluated from left to right; first create the range, then map the block to the values, then apply the filters. All filters must be satisfied for an item to be added to the list.

If the map section is left empty, the list is evaluated as a basic range.

```
element> [10 ,]
[1 2 3 4 5 6 7 8 9 10]

element> ['\U00A3' '\U00B0' ,]
"£ ¤ ¥ ¦ § ¨ © ª « ¬ ® ¯ °"

element> [0 3 15 , !]
[0 -3 -6 -9 -12 -15]
```

Here are some examples using map and filter.

```
element> [10, 2*]
[2 4 6 8 10 12 14 16 18 20]

element> [10, 2*, 5<]
[2 4]

element> [10, 2*, 5<, 4=!]
[2]

.# Can grab from stack
element> 3 [1| 6 18, 2*]
[ 6 12 18 24 30 36 ]
```



If a list literal is used as the first section of a list comprehension, the list comprehension is simply applied to the inner list.

```
element> [ [1 2 3 4 5], 2*, 7<]  
[2 4 6]
```

If there are **two or more** lists used as the first argument of a list comprehension, and each list is the same length, **all** respective elements of each list will be added to the stack when applying the map and filter sections.

```
element> [ [1 2 3][4 5 6], +]  
[5 7 9]  
  
element> [ "hello" "world", K]  
[ "hw" "eo" "lr" "ll" "od" ]
```

## VARIABLES

---

Variables may only contain 12 or less lower case letters. They are assigned using the colon (:) operator. The value is left on the stack after the assignment has occurred.

```
1 :a      .# => 1
3:b a +   .# => 4
```

Variables may only be 12 characters long. If a variable is more than 12 characters long, the parser will print a warning and only use the first 12 characters of the variable. This means that two seemingly different variable names that share the same first 12 characters will be considered the same variable.

```
thisisaverylongvariablename
|---used---|--ignored-->
```

If the first 12 characters are the same, the rest is ignored. The parser evaluates them as the same variable:

```
element> "hello":abcdefghijklmnop;
element> abcdefghijklmnop
"hello"

.# The last 4 letters of this var are different
element> abcdefghijklwxyz
"hello"
```

## Special Character Variables

Any of the [named special characters](#) can be used as a variable. Since variables can only contain lowercase letters, the parser translates the special character into its name and uses the name as the variable internally. This means that internally, the character and the name of the character are **the same** variable. Unicode characters that do not have a defined name cannot be used as variables. Variables containing special characters may only be **one symbol long**.

```
element> 0.05:α;
element> 100 α *
5.0
```

```
.# the number 1 is only assigned to β
element> 1:βα
1 0.05

element> αβ
0.05 1
```

Internally, character names are the same variable as the character.

```
element> 2:alpha;  
element> 4:β;  
element> α beta *  
8
```

Variable scope is discussed in the [Variable Scope](#) section of this document.

## TYPES

---

The `.T` operator returns an object's type as a character.

```
1 .T      .# => 'I'
'1 .T     .# => 'C'
1.t.T     .# => 'C'
```

The library function `isa` takes a type and an item and returns true if the item matches the type.

```
5 'I isa      .# => true
[1 2 3 4] 'L isa .# => true
```

'N' represents any number type (Int, Double, BigDecimal).

```
[ 1 1.0 1.0Ms ] #.T      .# => "IFD"
[ 1 1.0 1.0Ms ] 'N #isa  .# => [ true true true ]
```

'A' represents any type.

```
[ 'c "c" 1 [] ] 'A #isa  .# => [ true true true true ]
```

Strings are lists.

```
"hello" 'L isa      .# => true
```

Lists are only strings if all items are characters

```
[1 2 3] 'S isa      .# => false
['a 'b 'c] 'S isa   .# => true
```

## Verbose Type Names

Verbose type names can be used to get type names for user types and module names, To get a verbose type name, use the `Mw` operator.

When using `Mw` with modules, the module name is returned with a colon. Here we retrieve the type of the `frac` module. A colon is used to denote that the object is a module.

```
element> frac Mw
":frac"
```

When using `Mw` with a user type, the module name is returned. A period is used to denote that the object is an instance of a user type.

```
element> 2z3 Mw  
".frac"
```

When using Mw with primitive types, the full name of the type is returned.

```
element> 1 Mw  
"INT"
```

## Type Chart

Throughout the documentation and operator descriptions, the following uppercase letters are used to represent types.

Character	Type
I	Integer
D	Double
F	BigDecimal
N	Number (I or D or F)
B	Boolean
E	Expression (Block)
C	Character
L	List
S	String
U	User Type
M	Module
A	Any

For information on creating your own types, see the [UserTypes](#) section below.

## BLOCKS

---

Blocks contain expressions. They can be used to define functions, map instructions to lists, etc. They are denoted using curly braces { }.

```
{20 50 +}
```

They can be evaluated by using the ~ operator.

```
{20 50 +}~ .# => 70
```

When blocks are evaluated, their contents are dumped to the stack and the stack continues as normal. This is what happens when we call functions as well.

```
100 10+ {1 + 2 *}~  
    110 {1 + 2 *}~  
        110 1 + 2 *  
            111 2 *  
                222
```

## Block Header

A comma ( , ) is used to specify that the block has a header. Anything before the comma is considered the header and everything after is considered the instructions. A block header is used to introduce local variables to the block in the form of arguments or local declarations. Arguments and declarations are separated by a colon ( : ). Arguments must go on the left hand side of the colon and local declarations on the right.

```
{<arg1> <arg2> ... <argN> : <local dec 1> ... <local dec M>, <block body>}
```

If no colon is included in the header, all variable names will be used as arguments.

```
{<arg1> <arg2> ... <argN>, <block body>}
```

If a colon is the first token in a block header, all variable names are considered local declarations.

```
{: <local dec 1> <local dec 2> ... <local dec M>, <block body>}
```

## Arguments

Arguments work like parameters in programming languages with anonymous/lambda functions. Before the block is evaluated, its arguments are popped from the stack and assigned as local variables for the block.

```
element> 4 {a, a2*}~  
8  
  
.# arguments are popped in the order they are written  
element> 8 4 {a b, [a b] R}~  
[8 7 6 5 4]  
  
.# arguments are local variables  
element> 2:n 3{n, n2^}~ n  
2 9.0 2
```

## Argument Type Assertions

Arguments may have type assertions. The type assertion is written as an uppercase letter (see type chart) following the argument name. Whitespace is optional.

```
1 2 {a I bI, a b +}~      .# => 3  
1.0 2.0 {aI bI, a b +}~  .# TYPE ERROR: Type error at ({ARGS}):  
                          Expected (INT)  
                          Recieved (<BIG_DECIMAL:2.0> )
```

## Local Declarations

Local declarations introduce a local scope for that variable. Scope is discussed in greater detail in the [Variable Scope](#) section of this document. Local declarations can not have type declarations.

```
element> "A":a  
"A"  
  
element> a println {a, "B":a; a println}~ a println  
A  
B  
A
```

## FUNCTIONS

---

We now have the basic building blocks for defining functions: variable assignment and blocks. A function is simply a variable that is bound to a block. When the variable is called, the interpreter dumps the contents of the block onto the instruction stack and then continues evaluating. Functions can take advantage of anything that a normal block can including arguments and argument types.

Here are a few examples of function definitions:

`swapcase` takes a character and swaps its case.

```
element> {cC,c!}:swapcase;
element> 'q swapcase
'Q'
```

Below is the definition of the standard library function `roll`, This function will move the last element of a list to the front .

```
element> {_1!I\BK}:roll;
element> [1 2 3 4] roll;
[4 1 2 3]
```

When used with block arguments, functions can be written in very readable ways. The following function `swapitems` takes a list and two indices and swaps the respective elements. It uses block arguments and type assertions.

```
{listL iI jI,
  list i I : tmp;
  (list i : (list j I)) : list;
  list j : {tmp}
}:swapitems;
```

To see more examples check out the standard library located at `/base/std.elem`



## VARIABLE SCOPE

---

A new scope is introduced if a block contains any variable declaration in its header. When a variable assignment occurs, the interpreter will walk outward until a reference to that variable appears. If it does not appear in any of the scopes before the global scope, a new reference will be created there. In order to ensure a variable is using local scope, the variable name must be included in the block header. If a block does not contain a header, a new scope will not be introduced. These concepts are best demonstrated by example.

Let us introduce the variables `a` and `b`:

```
"A":a; "B":b;
```

When blocks have arguments, a scope is introduced for that variable. Here, the number zero is assigned to `b` within the scope of the block. When the block ends, the scope is destroyed and we reference the now global variable `b`.

```
element> 0 {b, b.P}~ b.P  
0B
```

Local variables also create local scopes for that variable. Here, we create a local scope for the variable `b`. `a` is not included in the new scope.

```
element> .# Local variable b declared in header  
{:b,  
  0:a;  
  1:b;  
  "a = $a," .P  
  "b = $b\n" .P  
}~  
"a = $a," .P  
"a = $b\n" .P  
  
a = 0,b = 1  
a = 0,a = B
```

## TICKS AND GROUPS

---

### Ticks

Sometimes we may need to traverse the stack backward before it has been evaluated. The tick ( ``` ) operator will move the item just after it back through the instructions 1 place. This will occur during runtime just BEFORE the item is evaluated.

```
1 `2 3 4      .# => 1 3 2 4

.# The object will be moved back BEFORE being evaluated
1 `+ 1        .# => 1 1 + => 2
```

Ticks can be stacked. The object will move back one place for every tick.

```
` `+ 3 4      .# => 3 4 + => 7
```

Since ticks are evaluated at run time, users can define their own "infix" operators.

```
{`*}:times;
3 times 4      .# => 12
```

### Groups

Objects on the stack can be grouped using parenthesis. Items in parenthesis are dumped and evaluated at run time. They are similar to a block followed by a `~`.

```
{1 2 + 3}~ +    .# => 6
(1 2 + 3) +     .# => 6
```

Grouped items are treated as one item by the interpreter and are therefore especially useful when used with the tick operator.

```
`+ (1 2)       .# => 3
```

If a block is the only thing inside a group, it will be automatically dumped and evaluated.

```
1 2 {+}         .# => 1 2 {+}
1 2 ({+})       .# => 3
1 2 ({n m, n m +}) .# => 3 (This allows groups to have arguments)
```

## MODULES

---

### Namespaces

A module is nothing more than a set of variables. Variables can only be assigned or modified when the module is declared. To declare a module, use a '@' character as the first character in a block header.

```
{@ <module name>,  
  <module body>  
  ...  
}
```

Below is a simple namespace example.

```
.# Define a simple module  
{@ numbers,  
  1:one;  
  2:two;  
  3:three;  
}
```

### ***Access Variables***

Access variables are used to access variables in modules and user types. To create an access variables, use a dot before the variable name.

```
element> numbers .one  
1  
  
.# whitespace is optional  
element> numbers.two  
2
```

## User Types

In Element, user types can be used to define custom types with separate functionality and moderate operator overloading. User types are represented internally as an array of objects paired with a module.

This definition will become more clear if we examine an example. Let us define a basic 2D vector module that can be used as an object.

```
{@ vec,  
  
  .# Fields  
  fields "x y"  
  
  .# Constructor  
  {xN yN,  
    [x y] vec MO  
  }:new;  
  
  .# Member functions  
  
  {self,  
    "<$(self.x),$(self.y)>"  
  }:show;  
  
  {self,  
    self.x 2^ self.y 2^ + Mq  
  }:length;  
  
  .# Operator Overload (+)  
  {a b,  
    [a.x b.x+ a.y b.y+] vect MO  
  }:plus;  
}
```

### Object Creation

User types are simply an array of objects paired with a module. In order to create a user type, we use the MO operator. The first argument is the list of fields, or member variables, and the second argument is the module. To create a vector object, we create a list containing x and y values and pass it and the module name to the MO operator.

```
.# Create a vec object  
[2 3] vec MO
```

This syntax is a bit ugly and uninformative. In order to address this issue, we introduce *constructors*.

## Constructor

If there exists a function `new` in the module definition, it will be used as the constructor for the object. The constructor can be called in the following ways:

```
.# Calling the .new function manually
element> 3 6 vec.new
<3,6>

.# Using the ! operator after the name of the module
element> 1.1 3 vect!
<1.1,3>
```

Notice that when the object is printed to the console, it prints using our definition of `.show`. `Element` will automatically use `.show` to convert objects to strings whenever necessary ( e.g. printing to the console, calling the `P` operator, etc.)

## Fields

When we create an objects using the `MO` operator, the list of fields we pass into it gets saved into the user type instance. In order to access this list we use operators `M>` and `M<`.

### Getting Field Data

Fields can only be accessed by using their location in the list.

It is important to note that we index the lists starting at 1 not 0. This is because the 0<sup>th</sup> index is reserved for retrieving the module itself.

For example, to retrieve the value of `x` in a `vec` object, we use the following code:

```
element> 34 87 vec! :v
<34,87>

element> v 1 M>
34
```

Accessing the module:

```
element> v 0 M>
{Module: vec}
```

Accessing fields that do not exist will throw an error:

```
element> v 3 M>
ERROR: M>: User object ({Module: vec}): field 3 does not exist.
```

## Setting Field Data

To set field data, we use the `M<` operator in the same way we would use the `M>` operator. This time however, we append an additional argument to the front of the call:

```
element> 1 2 vec! :v
<1,2>

element> 6 v 1 M<

element> v
<6,2>
```

## The `fields` Function

Every time we want to access the `x` field of our object we would need to write the following:

```
myvec 1 M>
```

If we wanted to simplify this into something more readable, we could define a function inside of `vec` called `x` that allows us to access the `x` field.

```
{1 M>} :x;
```

This now allows the following behavior to work as one would expect:

```
myvec.x
```

Similarly we may define a function for setting the value of the `x` field:

```
{1 M<} :setx;
```

Defining all of these functions for getting and setting fields can be quite tedious and messy. In order to solve this issue, one may use the `fields` function. `fields` is a **prefix** function defined in the `element` standard library that takes a string of variable names and uses the string evaluation operator `~` to create and run getters and setters for all of the fields. When using this function, it is important to note the order in which the variables are typed; it must be the same as the fields given to the constructor. The call,

```
fields "x y"
```

creates the following functions

```
{1M>} :x;
{1M<} :setx;
{2M>} :y;
{2M<} :sety;
```

## String Conversion

If there exists a function `show` defined for a given user type, Element will call it whenever the type is converted into a string. Element expects a string to be returned from the function but does not check before converting. If `show` does not return a string, unexpected results may occur. In the `vec` example, we defined a `show` function and we can see the result every time the `vec` is printed to the console.

```
element> 1 2 vec!  
<1,2>  
  
element> 1 2 vec! P  
"<1,2>"
```

## Operator Overloading

Several operators have the capability to be overloaded by defining functions with special names. For example, the function `plus` will be called if the user calls `+` on a user object. The following operators may be overloaded:

```
+ - * / & | $ % D E I P Q
```

These operators and their function names can be found by searching "overloadable" in the [QuickSearch](#) feature of element.

In our `vec` example, we defined the following function:

```
{a b,  
  [a.x b.x+ a.y b.y+] vect MO  
}:plus;
```

Now the following statements are equivalent:

```
element> 1 2 vec! 3 4 vec!.plus  
<4,6>  
element> 1 2 vec! 3 4 vec! +  
<4,6>
```

**The number of arguments used in an overloaded function must match the number of arguments the operator normally takes. For example, the `+` operator must take two arguments and the `E` operator must only take 1.**

## DOCUMENTING CODE

---

Element supports dynamic documentation of code. Nearly all of the QuickSearch entries are added from within the standard library. In order to add an item to QuickSearch, use a `?`  directly after a comment operator. Line comments of the form `.#? text` and block comments of the form `.{? text .}` will be added to the search. Comment text may include standard escape characters and special characters like `\n`, `\alpha` and `\{U00A1}`. These will be evaluated before being added to the search. String Interpolation `$` will not be evaluated. Look through the standard library for examples.

## Common Practice

### *Functions*

Function documentation follows the following format:

```
.#? TYPE name\n  line 1\n  line 2
```

The comment will appear in the search as:

```
TYPE name  
  line 1  
  line 2
```

Here are some examples from `math.elem`:

```
{_E\S\}/}:avg;      .#? L avg\n  average of a list of numbers  
{1 3/^}:cbrr;      .#? N cbrr\n  cube root  
{lcm}.F}:llcm;     .#? L<N> lcm\n  least common multiple of a list of  
numbers
```

### *Modules and Types*

Modules should have a multiline comment at the top of the file. The first line should be `module: <module name>` or `type: <type name>`. The following lines should provide a short description of the type or namespace. Below is an example from the `fraction` module defined in `base`.

```
.{? type: frac  
  The frac type provides operations for manipulating fractions.  
  
  Fractions can be created using the fraction literal operator `z` (eg:  
1z2 => 1/2) or by using the constructor manually.  
.}
```



All variables defined in a module should maintain the same format as documenting functions. They shall also always include the module name before the variable name in the format `<module>.<variable name>.`

```
.#? (matrix) matrix.issq\n  returns true if the matrix is a square  
{self, self.rowsE self.rows#E\#= {||}.F}:issq;  
  
.#? (matrix) matrix.transpose\n  transpose a matrix  
{self, [self.rows~, .A] matrix!}:transpose;
```

## THE STANDARD LIBRARY

---

The standard library includes many useful functions, variable definitions, and user types. Some of these files are not very robust but meant as demonstrations of the features of element. Eventually, all of these files will have many more features and will be much more useful.

## Function and Variable Definitions

### ***Standard Functions (std.elem)***

`std.elem` includes many essential function and variable definitions. All control statements (such as `for`, `if`, `while`, etc.) are defined in `std.elem`. This file also defines essential functions such as `fields` (used for creating [UserTypes](#)), `print`, and the scientific notation operator `e`.

This file also has include statements for the rest of the standard library.

### ***Mathematical and Numerical Functions (math.elem)***

This file includes a number of basic mathematical functions relating to trigonometry, statistics, and general math.

### ***Golfing Functions (double\_chars.elem)***

Since 2 character variables are useful when golfing, this file is a list of all the possible 2 character variable names. Most of the file is a placeholder for future definitions.

### ***Units and Conversions (unit.elem)***

This file adds basic unit conversions and defines a unit conversion operator.

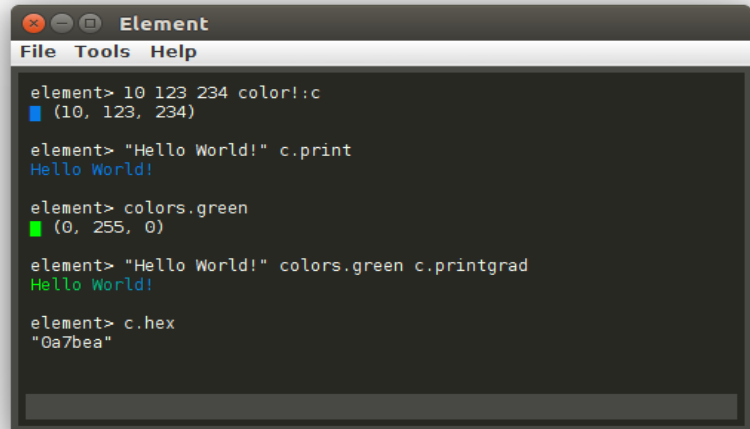
```
element> (30 unit.inch) to (unit.feet)
2.5

element> (45 unit.day) □ (unit.week)
6.4285714286
```

## Types

### ***Color (colors.elem)***

The `color` and `colors` modules are both defined inside of `colors.elem`. The `color` module defines a `color` user type. It has functions for manipulating, interacting, and printing colors. The `colors` module defines a many common colors so that they can be used easily.

A screenshot of the Element REPL window. The window has a title bar with 'Element' and standard OS window controls. Below the title bar is a menu bar with 'File', 'Tools', and 'Help'. The main area shows a series of commands and their outputs. The commands are: 'element> 10 123 234 color!:c' which outputs a blue square followed by '(10, 123, 234)'; 'element> "Hello World!" c.print' which outputs 'Hello World!'; 'element> colors.green' which outputs a green square followed by '(0, 255, 0)'; 'element> "Hello World!" colors.green c.printgrad' which outputs 'Hello World!'; and 'element> c.hex' which outputs '"0a7bea"'.

```
Element
File Tools Help

element> 10 123 234 color!:c
■ (10, 123, 234)

element> "Hello World!" c.print
Hello World!

element> colors.green
■ (0, 255, 0)

element> "Hello World!" colors.green c.printgrad
Hello World!

element> c.hex
"0a7bea"
```

### ***Complex Number (complex.elem)***

Complex numbers can be created using the `complex!` constructor or by simply using the predefined function `im`. Many mathematical operators are overloaded so complex numbers can be used with standard, built in numbers.

```
element> 6 complex!
6i

element> 3im 4+
4+3i

element> 1im 2^
-1+0i

element> 3im4+ 2im8.4+ *
27.6+33.2i

element> 4im8+ 2 /
4.0+2.0i
```

## ***Table (dataframe.elem)***

A module for interacting with tables. This module is not yet complete.

```
element> ["total" "percent"] ["a" "b"] [[10 15][40 60]] dataframe!:df
  total percent
a      10     40
b      15     60

element> "a" df.row
[ 10 40 ]

element> "percent" df.col
[ 40 60 ]
```

## ***Date (date.elem)***

The date module allows for accessing and converting dates.

```
element> date.now :d
06/20/16

element> d.year
2016

element> d.mmddyyyy
"06/20/2016"
```

## ***Dictionary (dict.elem)***

Dict.elem defines a basic dictionary type. It is essentially an array containing key/value pairs. Dictionaries can be accessed using the key which can be of any type. This module overloads the I and D operators for easy indexing.

```
element> [{"a" 1} {"b" 2} {"c" 3}] dict! :d
[dict
 [ a 1 ]
 [ b 2 ]
 [ c 3 ]
]

element> d "a" I
1

element> 10 d "b" D
```

```

element> d
[dict
  [ a 1 ]
  [ b 10 ]
  [ c 3 ]
]

```

## ***Fraction (frac.elem)***

The `frac` module defines a fraction type and several functions for creating, approximating, and performing arithmetic on fractions.

`frac.elem` defines an easy to use constructor used in the previous examples. The `z` operator is an infix operator that takes its left hand side as the fractions numerator and the right hand side as the fraction's denominator.

```

element> 1z7
1/7

element> 2z6
1/3

```

It also defines functions such as `approx` and `reduce`. It also provides overloads for arithmetic operations such as `+`, `-`, `*` and `/`. For such operators, it automatically promotes input to a fraction. The following are all examples of proper use of arithmetic operators.

```

element> 1.25 2z4 +
7/4

element> 4z5 1z9 /
36/5

element> 2z7 1 +
9/7

```

## ***Matrix (matrix.elem)***

The `matrix` module provides a matrix type and several functions for creating and manipulating matrices. Matrices are constructed by typing a list of lists followed by the matrix constructor.

```

element> [[1 2][3 4]] matrix!
| 1 2 |
| 3 4 |

```

Most mathematical operators are overloaded and can be used with both scalar and matrix arguments (where applicable).

```

element> [[1 2][3 4]] matrix! :b;

element> [[10 11][12 13][14 15]] matrix! :a;

```

```

element> [[1 2][3 4]] matrix! :b;

element> a b*
| 43 64 |
| 51 76 |
| 59 88 |

element> b 2 *
| 2 4 |
| 6 8 |

```

To index elements of a matrix, use the overloaded `I` operator with a list containing the row and column numbers. To get an entire row or column use an empty list in place of the number. In the following example, we use the `&` operator which is overloaded to map an expression to each of the matrices' elements.

```

element> [4 4] matrix.ones {10Q}& :mat
| 2 6 0 6 |
| 8 3 0 9 |
| 9 5 9 4 |
| 7 3 4 9 |

element> mat [1 0] I
8

element> mat [1 []] I
[ 8 3 0 9 ]

```

The matrix module includes functions such as `transpose`, `trace`, `augment`, and many more. For more information read the code in `matrix.elem` or search "matrix." in the [QuickSearch](#).

## **Queue (*queue.elem*)**

Queues can be created and used in the following way

```

element> [1 "a" 3.4] queue! :q
queue: <- 1 a 3.4 -<

element> q$
1

element> 46 q+
queue: <- a 3.4 46 -<

```

Here, operators `$` and `+` are overloaded to perform the actions of `queue.next` and `queue.add` respectively. The queue module also provides functions such as `peek`, `length`, and `isempty`.

## **Set (*set.elem*)**

Sets provide a simple list type that may not have any duplicates. Operators `+` and `-` can be used to add or remove elements easily. Sets can be created and used in the following way.

```
element> [2e3 3 1.1 1.1] set!:s
set: [ 2000 3 1.1 ]

element> 67 s+
set: [ 2000 3 1.1 67 ]

element> 3 s+
set: [ 2000 3 1.1 67 ]

element> 3 s-
set: [ 2000 1.1 67 ]

element> 3 s.in
false
```

## **Stack (*stack.elem*)**

Stacks can be created and used in the following way

```
element> [1 2 3] stack! :s
stack: <- 1 2 3 -|

element> s$
1

element> 10s+
stack: <- 10 2 3 -|
```

Here, operators `$` and `+` are overloaded to perform the actions of `stack.next` and `stack.add` respectively. The queue module also provides functions such as `peek`, `length`, and `isempty`.

## **Other**

### **Documentation (*docs.elem*)**

`docs.elem` contains several help text comments whose contents are added to the [QuickSearch](#) when the file is loaded. The comments include things like type charts, quick references, etc.

### **Code Demo (*demo.elem*)**

`demo.elem` contains a few code examples and functions that are used to demonstrate some of element's features.

## THE REPL

---

The GUI REPL can be launched by double clicking the jar file or by running the command:

```
java -jar path/to/element.jar
```

A command-line version of the REPL can be launched by executing the above command with `-i` as an argument

```
java -jar path/to/element.jar -i
```

## REPL Features

Aside from running element code, the REPL has several useful commands. Use a backslash to enter a command. Enter `\h` to view the REPLs commands from within the REPL.

Command	Function
<code>\q</code>	Quit
<code>\help</code>	Print help text
<code>\clr</code>	Clears the console window (GUI version only)
<code>\version</code>	Prints basic version information
<code>\time</code>	See <a href="#">time</a> section below
<code>?</code>	See <a href="#">quick search</a> section below

## Time

Append `\time` to the front of an expression in the REPL to print the amount of time the expression took to execute (initial parse time is not included). Alternatively, one could use the `timeitstart` and `timeitend` functions built into the standard library.

```
\time 99999R #1+ S
705082703
Execution took 0.140919259 seconds
```

## QuickSearch

QuickSearch can be used from within the REPL using the command `\?` or by pressing `Ctrl-Q` (GUI Only). In order to perform a search, type `\?` into the repl followed by a space and then any text you wish to search for. The GUI version of QuickSearch provides an interactive search window with the same functionality as using `\?`.

Using QuickSearch, one can find information on the functionality of operators, functions from the standard library, general information on syntax, and many other things related to Element. Below are some examples.

Most of the entries of QuickSearch are defined in the element standard library (`base/...`). To



learn how to add items to the QuickSearch in your own files, see the Documentation section of this document.

## ***Examples***

Find out what an operator does

```
element> \? $  
$ (L<N>|L<S>|S)  
sort least to greatest  
(operator)
```

Find out which operator to use for a certain function

```
element> \? reverse  
! (S|N|L|B|C)  
  <N> negate  
  <S|L> reverse  
  <B> logical not  
  <C> swap case  
(operator)
```

Search for topics

```
element> \? comments  
docs: comments  
  .# this is a line comment  
  .<open curly>  
    this is a block comment  
  .<close curly>  
  
  .#? this line will be added to interactive help  
  .<open curly>?  
    these lines will be  
    added to interactive help  
  .<close curly>
```

## EXAMPLES

---

Below are some examples of the element language. For a more robust list of examples, check out the standard library and the math library.

### 2D Vector Type Definition

#### *Type Definition*

```
{@ vec,  
  
  members "x y"  
  
  .# Constructor  
  {x y, [x y] vec MO}:new;  
  
  .# Print Override  
  {self, "<$(self.x),$(self.y)>"}:show;  
  
  .# Member Function  
  {self, self.x2^ self.y2^ + Mq}:length;  
  
  .# Operator Overload  
  {a b, [a.x b.x+ a.y b.y] vec MO}:plus  
}
```

#### *Usage*

Call constructor using ``!`` operator and print using ``.show`` definition:

```
element> 1 2 vec!  
<1,2>
```

Perform operations on the type:

```
element> 3 4 vec! :v  
<3,4>  
  
element> v.length  
5.0  
  
element> 10 10 vec! v +  
<13,14>
```

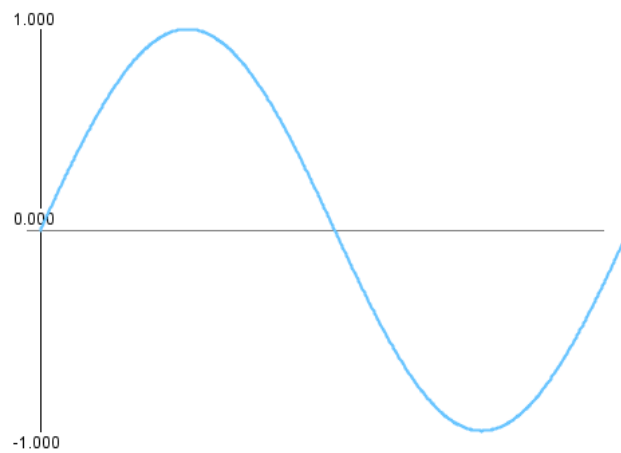
## Plot a sine wave

```
[0Δ2π*,Ms].X
```

### ***Explanation***

- `[a,b]` is list comprehension, it creates a list `a` and maps `b` to it
- The `Δ` (*delta*) operator creates a range between 0 and  $2\pi$
- The `Ms` operator performs the sin operation.
- The `.X` operator plots the list in a GUI

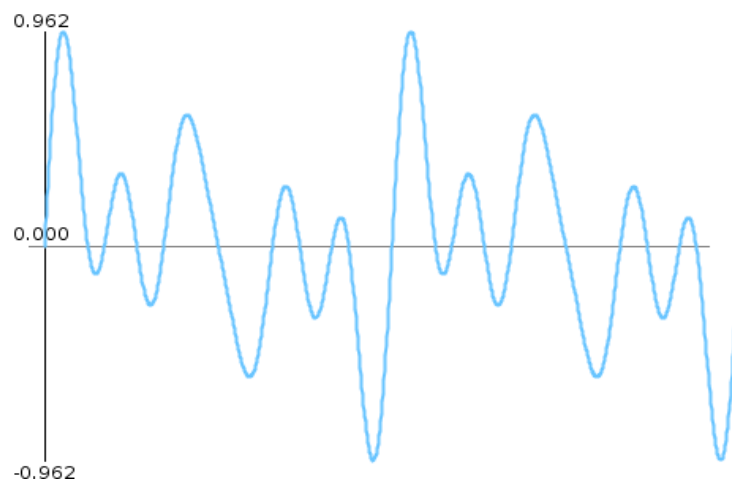
### ***Output***



## Plot a Fourier sine series with random coefficients

```
6:n;  
  
.#Generate coefficients  
[n,.Q]:cs;  
  
.#Series function  
{x, [cs [nR,x*Ms],*]S}:f;  
  
.#Plot series function from 0 to 4pi  
[0dy4pi*,f].X
```

### Output



## Project Euler Golf

### Problem 1 (15 Chars)

```
kVR{15gcd1=!}IS
```

### Problem 6 (12 Chars)

```
hR_S2^2#^S-
```

Explanation:

hR	Generate a list [1,2..100]
	Duplicate the list
-S2^	Evaluate the sum of the list and square it
\	Bring the other list to the top of the stack
2#^	Square each element in the list
S-	Subtract the sum of this list from the previous sum

### Problem 8 (31 Chars)

"list" is the list of numbers

```
list 13R{ _@_@I{*}.F\@#.) }987%;;.Amax
```

Explanation:

13R	Create a list for indexing(13 consecutive indices)
{ _@_@	} Duplicate the lists and bring them to the top
I	} Get 13 consecutive numbers in the sequence
{*}.F	} Take their product
\@	} Move the list and the indices back to the top
#.)}	Increment the indices
987%	Repeat (1000-13) times
;;	Remove the list and indices from the stack
.A max	Take the entire stack as a list and

return the max value

### Problem 34 (25 Chars)

```
[3 5E,,_P#{'0-iM!}S1+i=]S
```

Explanation:

[3 5E	]	List comprehension on the numbers
[3,4..100000]		
''	]	Don't map anything to the list, begin a filter
-P	]	Duplicate the original number, cast to string
#{	}	Map the block to each character in the string
'0-	}	Subtract the character '0'
i	}	Cast the character to an int
M!	}	Compute the factorial
S1+	]	Sum the list and add 1
i	]	Cast the double to an int
=]		Compare it to the duplication made earlier
S		Compute the sum of the LC

### ***Problem 48 (10 Chars)***

```
[k,_^]SdE%
```

Explanation:

[k,	Generate a list from 1 to 1000
_^]	Pow each element
S	Sum the list
dE%	mod 10000000000 to get the last 10 digits