# An *incomplete* Reference Manual

Beta 2

2nd October 2024

# Getting started

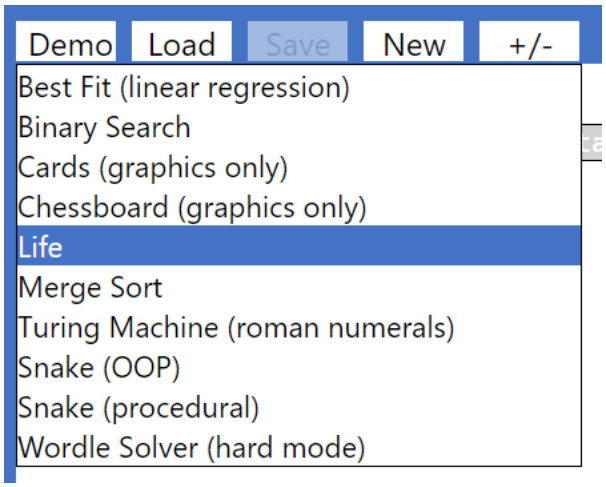
## Technical platform

You can access the Elan Beta at: https://elan-lang.org/beta/

Elan is designed to run within the Chrome browser; correct operation within other browsers is not guaranteed.

## Demo programs

The Beta version includes a **Demo** button that offers a menu of demonstration programs that you can run:




The best way to get started with Elan is to explore these demo programs. You can edit any of them and save your own copy locally.

# Changes and additions for Beta 2

## Language/grammar changes

- Revert 'div' and 'mod' from being functions to being operators Language / grammar
  #750 by richardpawson was closed 28 minutes ago Beta 2
- Support parameterless lambdas Language / grammar
  #742 by richardpawson was closed 3 hours ago Beta 2
- Rename ArrayList to Array; ImmutableList to List Language / grammar
  #723 by richardpawson was closed 3 weeks ago Beta 2
- Rename Iter to Iterable and improve error messages relating to this type Language / grammar
  #722 by richardpawson was closed 3 weeks ago Beta 2
- Make all uses of indexes read-only Language / grammar
  #720 by richardpawson was closed 3 weeks ago Beta 2
- Remove lazy evaluation of let statements Language / grammar
  #719 by richardpawson was closed 2 days ago Beta 2
- Support optional 'out' parameters for procedures Language / grammar
  #718 by richardpawson was closed 2 days ago Beta 2
- Remove constraint of max. 2 indexes on a result Language / grammar
  #712 by richardpawson was closed last month Beta 2
- Simplify enum syntax to a single line Language / grammar
  #703 by richardpawson was closed on Aug 29 Beta 2
- Support dot chaining of functon calls and/or properties Language / grammar
  #670 by richardpawson was closed last month Beta 2
- List and Tuple deconstruction Language / grammar
  #668 by richardpawson was closed 16 hours ago Beta 2
- Explicit syntax to reference a function Language / grammar
  #638 by richardpawson was closed 6 minutes ago Beta 2
- Permit abstract classes to define private concrete methods Language / grammar
  #606 by richardpawson was closed 20 hours ago Beta 2
- typeof operator Language / grammar
  #525 by richardpawson was closed 3 weeks ago Beta 2
- implement 'copy ... with ...' Language / grammar
  #311 by richardpawson was closed last month Beta 2
- Permit '_' (discard/ignore) in deconstructed tuple or list Language / grammar
  #248 by richardpawson was closed 16 hours ago Beta 2
- Support RegEx, including literal forms Language / grammar
  #239 by richardpawson was closed last month Beta 2

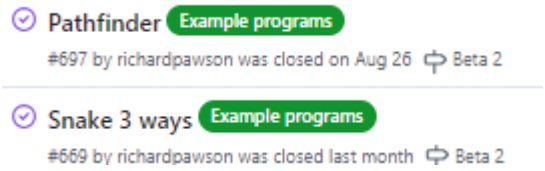
## Standard library changes

- Revise naming & capabilities of block graphics Standard Library
  #733 by richardpawson was closed 2 days ago Beta 2
- 'putAt2D' method to work on quasi '2D' arrays Standard Library
  #726 by richardpawson was closed 3 weeks ago Beta 2
- Provide named standard colours Standard Library
  #704 by richardpawson was closed on Aug 28 Beta 2
- Rename some methods on Dictionary and ImmutableDictionary Standard Library
  #701 by richardpawson was closed on Aug 26 Beta 2
- Graphics.draw() should pause (minimally) automatically Standard Library
  #700 by richardpawson was closed on Aug 26 Beta 2
- Make 'contains' method work for all iterables Standard Library
  #694 by richardpawson was closed on Aug 28 Beta 2
- Revise conversion between unicode and string character Standard Library
  #655 by richardpawson was closed 1 minute ago Beta 2
- simplify asString for lists & dictionaries Standard Library
  #651 by richardpawson was closed last month Beta 2
- Bitwise functions Standard Library
  #637 by richardpawson was closed on Aug 29 Beta 2
- Implement functional Random type Standard Library
  #625 by richardpawson was closed on Aug 27 Beta 2

## Changes to the editor

- Make inheritance clause a single field - so can more easily be edited Editor
  #748 by richardpawson was closed 1 hour ago Beta 2
- Change the way abstract/immutable classes, and abstract/private members are added Editor
  #724 by richardpawson was closed 3 weeks ago Beta 2
- New 'Trim' button to remove all removable 'new code' selectors Editor
  #672 by richardpawson was closed on Aug 26 Beta 2
- Allow a new frame, where no fields have been entered to be deleted by Backspace Editor
  #653 by richardpawson was closed last month Beta 2
- Tint the Code pane light blue when it does not have focus Editor
  #371 by richardpawson was closed last month Beta 2
- Frame numbering ('line numbering') Editor
  #162 by richardpawson was closed on Aug 26 Beta 2

## New Demo (example) programs




# Still to be implemented..

The following lists a few of the *stand-out* items that are not yet implemented. (For a much longer list you are welcome to browse all the open items on our development project planning system – https://github.com/elan-language/IDE/issues . However, please bear in mind that those items are written by and for the development team – rather than for public discussion.)

## Editor

- **Debugger**. Ability to insert breakpoints, pause, single-step, and read the state of variables
- **Navigate** directly from use of an identifier to its definition
- **Renaming** of identifiers (variables, constants, parameters, function & procedure names)
- **Profile** configuration (exists as a proof of concept only at present). In future release you will be able to define multiple profiles and then assign a profile to each user name.
- Ability to switch on **anti-plagiarism** option (exists as a proof of concept only at present)
- Ability to perform all navigation and *actions* by keyboard or mouse (except entering code into fields, which must be done through the keyboard)
- **Auto-complete** (of names) is currently implemented for just a few kinds of field. This will be expanded, for example to offer auto-complete options within 'expression' fields.
- When calling a procedure, or using a function in an expression, **prompting** the user for each of the required arguments, with both parameter name and type required.

## Library methods and types

- Data file reading and writing (data file reading exists as a 'mock up' only in Beta 2)
- `Stack`, `Queue`, `Set`, `Tree`, `Graph` and other standard data structures (both mutable and immutable versions of each will be included as standard)
- Ability to call methods from *external* (JavaScript) libraries & frameworks

## Language

- Currently, the `global.` qualifier - to disambiguate between a global and a local identifier where two have the same name – works for `function` and `procedure`, but does not yet work for `constant`.

# Most significant differences between Elan and other programming languages

- Elan is **case-sensitive**.
- Elan is **space-sensitive**. In general it is better not to add spaces, except where they would be essential to the meaning, for example between a keyword and a variable. Adding unnecessary spaces can result in a parse-error – so if you type a space and the field immediately goes red, then delete that space! Elan will *insert* spaces for readability in a few places, for example around the `+` and `-` operators.
- All **identifiers** – (named values defined by a `constant`, `var`, or `let` statement; parameters; `function` and `procedure` names) – *must start with a lower-case letter*. This may be followed by any combination of lower- and upper-case letters, numeric digits, and/or the underscore _ symbol.
- Elan is a **statically-typed** language. All variables, parameters, properties (on a `class`), and what a `function` returns) *have a defined type which does not change*. Variables *must always be initialised* to a value where they are defined (in the `var` statement), and the type of that initial value defines the type for the variable. Parameters on a method require the type to be specified using the `as` keyword, for example:
  `myFunction(a as String, b as Int)`
- All **type names** – whether standard types that come with the Elan language – or user-defined types (`class` and `enum`) – must start with an upper-case letter, but otherwise follow the same rules as identifiers.
- **constants** are always global in scope.
- There is no such thing as a **global variable** in Elan. All variables exist with the scope of a method. *Within a given scope*, all variable names, parameter names, function and procedure names, must be unique. You may, within the scope of a method for class, define names that coincide with names in a wider scope, in which case the most local definition will be used- effectively 'masking' the global meaning. However, it is possible to access a method defined outside the current context be prefixing the name with `global.` or `library.`
- Elan **functions** are *pure* functions. They cannot generate side-effects, nor have dependency on anything except the items passed in as parameters. A function deliberately *cannot*:
  - Contain any `print` statements
  - Use any
  - **System** methods within its expressions
  - Make a `call` to any procedure (since procedures are not pure)
  - Assign-to or mutate any parameter
  - `return` a result from anywhere other than the last statement
- Elan does not support **overloaded** method names (methods with the same name, but different number or type of parameters).
- Elan does not use the equals sign either for assignment or for comparison (except as part of the `>=` or `<=` comparison operators). Assignment of a value to a variable is done within the `var` statement, or subsequently with the `set` statement. For comparison use the `is` or `is not` keywords. The `is` keyword tests for **equality by value**.

## Most significant differences in OOP

- The name given to a `class` must follow the rules for any Type name i.e. it must *start* with a capital letter.
- Every (concrete) class has a constructor – which is automatically added when you define a class. But it is not essential to define any code within that constructor.
- A property may be assigned (set) a new value within the `constructor`, or within a `procedure` method defined on the class or the sub-class. But a property may never be assigned by code outside the class. If you require this capability you can write your own procedure methods (commonly called 'setter' methods) to do this. A `function` method may read properties, but not write to them.
- If the constructor or a method defines a parameter with the same name as a `property` then using that name will refer to the parameter by default. If you wish to access the property of the same name (for reading, or writing), then prefix the name with `property.` for example:

```
procedure setName(name as String)
   set property.name to name
end procedure
```

- A concrete `class` may inherit from *one or more* `abstract class`es, but may not inherit from another concrete `class`. (This enforces the widely-recognised OOP design principle that 'all classes should be abstract or final (not inheritable)'.)
- An `abstract class` may define `abstract` members (`property`, `function`, `procedure`) – which must always be implemented by any concrete sub-class.
- An `abstract class` may also define private members (`property`, `function`, `procedure`), which are visible to any concrete sub-class, but not visible from code outside the class hierarchy.

- An `immutable class`, may define properties or `function` methods, but not `procedure` methods (because the latter necessarily mutate the instance). Function methods may provide information derived from reading the properties, or may return a new instance based on the current one but with modifications (specified using the `copy … with …` construct). An `abstract immutable class` may define properties or `function` methods that are either `abstract` or `private`.

# The Elan editor – quick reference

## Navigation – using the keyboard

**Note**: For *Apple Mac* users: commands use of the **Ctrl** key in this reference, should be replaced by the **Cmd** key.

| Keystroke | | On a selected Frame | On a selected Field |
|---|---|---|---|
| **Home** | | First *peer-level* frame. | Move text-cursor to start of field. |
| **End** | | Last *peer-level* frame. | Move text-cursor to end of field. |
| **Tab** | | First *field* in frame. | Select next field within frame. Or from last field in a frame, select the frame itself.<br>(If the field has a selected option in the auto-complete popup list then Tab will use that option – the same as **Enter**) |
| ↑ | | Select previous frame (within peer-level only). | Select previous *frame* (in tab order). |
| ↓ | | Select next frame (within peer-level only). | Select next *frame* (in tab order). |
| ← | | Select *parent* frame (if any). | Move text-cursor left within field. |
| → | | Select first *child* frame (if any). | Move text-cursor right within field. |
| **Shift-↑** | | *Add* prev. frame (peer-level) to current selection. | If auto-complete options are offered (drop-down list), move the selection *down* one in the list. (See also **Enter**) |
| **Shift-↓** | | *Add* next frame (peer-level) to current selection. | If auto-complete options are offered (drop-down list), move the selection *up* one in the list. (See also **Enter**) |
| **Esc** | | | Escape from field and select enclosing frame |
| | | | |
| **Ctrl-o** | | Toggle (expand/collapse) outlining on selected frame. | Toggle (expand/collapse) outlining on the frame enclosing this field. |
| **Ctrl-O** (Ctrl-Shift-o) | | Toggle (expand/collapse) outlining on *all* frames. | Toggle (expand/collapse) outlining on *all* frames. |

# Editing – using the keyboard

| Keystroke | | On a selected Frame | On a selected Field |
|---|---|---|---|
| **Backspace** | | On any 'new code' selector: delete the selector. (Note that *all* 'new code' selectors can be removed with the **+/-** button above the code pane).<br>On a new, unmodified, frame, or from any unedited field within that new frame: delete the whole frame and go back to the selector. This capability is to facilitate deleting a frame created unintentionally. As soon as any field has been edited, or any child frame added – the frame can only be deleted using Ctrl-Delete (see below). | Delete character to the left of the cursor. |
| **Delete** | | | Delete the character to the right of the cursor. |
| **Ctrl-Delete**<br>or<br>**Ctrl-d** | | Delete the selected frame, including any frames within it. | |
| **Enter** | | Insert a selector-frame ('new code') *below* selected, at peer level – if permissible. | If auto-complete options are offered (as a drop-down list), enter the selected option into the field.<br><br>Otherwise, move to the next field (in the same frame) – like Tab.<br><br>For last field in frame only: insert 'new code' *after* this field. |
| **Shift-Enter** | | Insert a selector-frame ('new code') *above* selected, at peer level – if permissible | - |
| **Ctrl-↑** | | Move selected frame(s) up, *within peer level*. | - |
| **Ctrl -↓** | | Move selected frame(s) down, *within peer level*. | - |
| **Ctrl-x** | | Cut selected frame(s) into the scratchpad | - |
| **Ctrl-v** | | - | Applies only to a selected 'new code' field. Paste the frame(s) added to the scratchpad in place of the 'new code' field. If any of the frames to be added is not compatible with the content of the 'new code' field then no action will take place. |
| **Ctrl-z** | <- | Undo last operation NOT YET IMPLEMENTED | -> |
| **Ctrl-y** | <- | Redo last undo NOT YET IMPLEMENTED | -> |

# Mouse operation – quick reference

- To select a *frame*, click on the *keyword* at the start the frame. (You can successfully click in several other places within the frame, but the simplest rule to remember is click on the starting keyword).
- To select an additional frame ('multi-select'), hold down the **Shift** key on the keyboard and click on the frame to add to the current selection. Note that all the multi-selected frames must be at peer-level (the same level of indentation and, unless global frames, must be within the same 'parent' frame.
- To select a field, click on the text (or, if empty, the prompt) shown for that field. Having selected the field, you may then click again at a particular place within the text to position the text cursor. (By default, when a field is selected the text cursor will be at the right-hand end of any existing text).
- To collapse a multi-line frame, double-click on the keyword at the start of the frame
- To expand a collapsed frame, double-click on the keyword at the start of the frame (or the '+' symbol in front of it)

NOT YET IMPLEMENTED

- To move selected frame, or frames, up or down *within the same peer level* hold down the **Ctrl** key and drag the mouse, or move the scroll wheel
- Scrolling of options within the 'autocomplete' popup using the mouse wheel.

# Expressions

One of the most important constructs in programming is the 'expression'. An expression is evaluated to return a value. An expression is made up of the following possible elements:

- **Literal value**
- **Named value**
- **Operators** (including brackets)
- **Function calls**

# Literal value

A literal value is where a value is written 'literally' in the code, such as `3.142` – in contrast to a value that is referred to by a name.

The following data types may be written as literal values (follow the links to view the form of each literal value):

**Int**, **Float**, **Boolean**, **String**, Error! Reference source not found., Error! Reference source not found., **Dictionary**, **ImmutableDictionary**, **Tuple**

# Named value

A named value is a value that is associated with a name rather than being defined literally in code.

There are various kinds of named value:

**Constant**, **Variable**, **Parameter**, **Index**, **Enum**

*Identifer*

For all kinds of named values, the name must follow the rules for an 'identifier' – it must start with a lower-case letter, followed by any combination of lower-case and upper-case letters, numeric digits, and the _ (underscore) symbol. It may not contain spaces or other symbols. Once a named value has been defined, it can be referred to by the name.

## Constant

Explanatory video:
https://www.youtube.com/watch?v=KxkCDnYWkZ0&list=PLhZaBW7EbafOPO4YyuovGI1prCViAeVKM&index=10

A `constant` defines a named value that cannot change.

A `constant` is always defined at 'global' level (directly within a file) and are global in scope. A `constant` may not be defined within any method. (However, see the Error! Reference source not found.).

The name of a `constant` follows the rules for any **Identifer**.

The value to which a constant is set must be a **Literal value**, of one of the following types: **Int**, **Float**, **Boolean**, **String**, Error! Reference source not found., **ImmutableDictionary**

Examples:

```
constant phi set to 1.618
constant maxHits set to 10
constant warningMsg set to "Limit reached"
constant fruit set to {"apple", "orange", "banana"}
constant black set to 0x000000
constant red set to 0xff0000 constant scrabbleValues set to {"A":1, "B":3, "C":3,
"D":2, "E":1, "F":4, "G":2, "H":4, "I":1, "J":8, "K":5, "L":1, "M":3, "N":1,
"O":1, "P":3, "Q":10, "R":1, "S":1, "T":1, "U":1, "V":4, "W":4, "X":8, "Y":4,
"Z":10}
constant colours set to {Suit.clubs:black, Suit.diamonds:red, Suit.hearts:red,
Suit.spades:black}
```

(In the last example above, `Suit` is an **Enum**)

## Variable

A variable is a named value where the value may change during the running of the program.

The name of a variable follows the rules for all identifiers

A variable is defined using a **Var statement** and may be re-assigned using a **Set statement**

Elan is a statically-typed language, so that each variable always has a defined type and any value assigned to that variable must be compatible with that type.

## Parameter

- A parameter is a specific kind of variable, defined as part of a method (**Procedure** or **Function**) for the purpose of capturing an argument being passed into that method.
- See also **Parameter passing**.

## IndexedValue

If a variable that is of an indexable type then an index, or a range, may be applied to the variable within an expression. For example:

```
var a set to "Hello World!"
print a[4]
print a[4..7]
```

See also: **Using an Array**, **Using a Dictionary**.

**Important:** unlike in many languages, in Elan, indexes (whether, single index, multiple index, or index range) are only ever used for *reading* value(s). Writing a value to a specific index location is done through a method such as:

- `putAt` on an `Array`
- `withPutAt` on a List
- `putAtKey` on a `Dictionary`
- `withPutAtKey` on an `ImmutableDictionary`

## Enum

Explanatory video:
https://www.youtube.com/watch?v=k0lPAnNCDh0&list=PLhZaBW7EbafOPO4YyuovGI1prCViAeVKM&index=19&pp=gAQBiAQB

An `enum` – short for 'enumeration' – is the simplest form of 'user-defined type' , specifying a set of values, each defined as a name, such that a variable of that type must always hold one of those values.

*Type name*

The name given to an `enum` (see below), which must begin with a capital, is used as the Type name, when passing a value to or from a procedure of function.

*Defining an enum*

Example

```
enum Status incomplete, ready, running, stopped, invalid
```

Further examples of `enum` may be seen in **Cards**.

*Using an enum*

The value is specified by the type name for the specified enum, followed by a dot and the value name, for example:

```
var x set to Status.ready
```

*Notes*

- Enums are *read-only* – once they have been defined it is not possible to add, remove, or update the values.

Example:

```
enum Suit clubs, diamonds, hearts, spades
```

# Operator

## Arithmetic operators

Arithmetic operators can be applied to `Float` or `Int` arguments, but the result is always a `Float`:

- `2^3` gives `8`
- `2/3` gives `0.666…`
- `2*3` gives `6`
- `2 + 3` gives `5`
- `2 - 3` gives `-1`
- `11 mod 3` gives `2`
- `11 div 3` gives `3` (integer division)

Arithmetic operators follow the conventional rules for precedence i.e. 'BIDMAS' (or 'BODMAS' _

**Note:** When combining `div` or `mod` with *any* other operators within an expression, *insert brackets to avoid ambiguity* e.g.:

```
(5 + 6) mod 3
```

The minus sign may also be used as a unary operator, and this takes precedence over binary operators so:

- `2*-3` gives -`6`

Note the Elan editor automatically puts spaces around the + and – *binary* operators, but not around `^`,`/`,`*`. This is just to reinforce, visually, the precedence.

## Logical operators

Logical operators are applied to `Boolean` argument(s) and return a `Boolean` result.

- `and` and `or` are binary operators
- `not` is a unary operator.

The operator precedence is `not` -> and -> `or`.

Example:

```
function xor(a as Boolean, b as Boolean) return Boolean
  return a and not b or b and not a
end function
```

Implements an 'exclusive or'.

## Equality testing

Equality testing uses the `is` and `isnt` keywords with two arguments. The arguments may be of any type.

- `a` is `b` returns `true`, if `a` and `b` are both of the same type and their values are equal. The only exception is that if one argument is of type `Float` and the other is of type `Int`, then `is` will return `true` if their values are the same (i.e. are the same whole number).
- `isnt` returns the opposite of `is`

Note that quality testing in Elan is *always* 'equality by value'. There is no such thing as 'equality by reference' in Elan.

**Notes**

- Where a binary operator is expected, as soon as you type `is` the editor will automatically insert a space after it. To enter `isnt` you need to delete the space (using the **Backspace** key) and then type `nt`

## Numeric comparison

The numeric comparison operators are:

- `>` for 'greater than'
- `<` for 'less than'
- `>=` for 'greater than or equal to'
- `<=` for 'less than or equal to'

Each is applied to two arguments of type `Float`, but any variable of expression that evaluates to an `Int` type may always be used where a `Float` is expected.

**Notes**

- These operators cannot be applied to strings. Use the methods `isBefore` and `isAfter` to compare strings alphabetically.
- Where a binary operator is expected, as soon as you type `<` or `>` the editor will automatically insert a space after it. To enter `>=` or `<=` you need to delete the space (using the **Backspace** key) and then type `=`

### Combining operators

You can combine operators of different kinds e.g. combining numeric comparison with logical operators in a single expression. However the rules of precedence between operators of different kinds are complex. It is strongly recommend that you *always* use brackets to disambiguate such expressions, for example:

```
(a > b) and (b < c)
(a + b) > (c -d)
```

# Function call

An expression may simply be a function call, or it may include one or more function calls within it. Examples:

```
print sinDeg(30)
var x set to  sinDeg(30)^2 + cosDeg(30)^2
var name set to inputString("Your name”)
print name.upperCase()
```

**Notes:**

- The third example (above) is not strictly a function call, but is a **System methods** call. System methods may only be used within the **Main routine** or a **Procedure**, because they have external dependencies or side effects.
- The fourth is an example of a

- **Dot** methods call – `upperCase` being a ‘dot method’ that may be applied to any instance (variable or literal) of the type `String`.

# Procedural programming

## Main routine

Explanatory video:
https://www.youtube.com/watch?v=Tg1SKYcrF4E&list=PLhZaBW7EbafOPO4YyuovGI1prCViAeVKM&index=8

A file must have a `main` method *if it is intended to be run as program*. (You may however develop and test code that does *not* have a `main` method – either as a coding exercise, or for subsequent use within another program).

The `main` method defines the start point when a program is run.

The `main` method does not have to be at the top of the file, but this is a good convention to follow.

There may *not* be more than one `main` method in a file – and the global selector (above) will not show the `main` option when one already exists in the file.

Example:

```
main
  var li set to [3, 6, 1, 0, 99, 4, 67]
  call inPlaceRippleSort(li)
  print li
end main
```

## Input/Output

### Printing to the Console

Explanatory video:
https://www.youtube.com/watch?v=NGYQQeAuKAg&list=PLhZaBW7EbafOPO4YyuovGI1prCViAeVKM&index=20&t=2s

The simplest way to print is with the print statement.

- See also **print & printTab** (standard library procedures).

### Inputting data from the keyboard

Explanatory video:
https://www.youtube.com/watch?v=ziYfalHJ9q4&list=PLhZaBW7EbafOPO4YyuovGI1prCViAeVKM&index=21

See **Input methods**

Also the `readKey` system method on `BlockGraphics`

## Simple BlockGraphics

Explanatory video:
https://www.youtube.com/watch?v=JbLtgVil7fI&list=PLhZaBW7EbafOPO4YyuovGI1prCViAeVKM&index=22

See **On BlockGraphics**

## Reading and writing data files

See
**The limitation of this** simple approach is that – because these methods depend on an unseen random number generator that changes state each time either method is called – the methods may only be used within `main` or a `procedure`. You may generate random numbers in this way and pass the resulting number(s) into a `function` as `Float` or `Int` Type(s), but you may not generate a random number using these methods include a `function`.

*Generating random numbers within a function*

It is possible to create and use random numbers within a function, but it requires a different approach and is a little more complex, using a special *type* named `Random` (note that the R is capitalised). You must always create the *first* `Random` (using the system method `firstRandom`) within `main` or a `procedure` but you can pass this into a function and generate further random numbers within that function – because the `next` number is generated in a deterministic fashion from the previous `Random` number, using a 'pseudo random sequence'. Example of use:

```
main
  var rnd set to firstRandom()
  var dice set to 0
  for i from 1 to 10 step 1
    set (dice, rnd) to rollDice(rnd)
  end for
end main

function rollDice(rnd as Random) return (Int, Random)
  return (rnd.valueInt(1, 6), rnd.next())
end function
```

Note that calling `firstRandom()` uses the system clock as the 'seed' value, and hence will be different each time you run the program. It is also possible to create a pseudo-random sequence that will be the same each time the program is run – and this is primarily intended for testing purposes. To create the fixed sequence replace `firstRandom()` with `firstRandomInFixedSequence()`, but leave the rest of your code unchanged.

Reading & writing data files

# Using variables

Explanatory video:
https://www.youtube.com/watch?v=g6Byq0vhYw8&list=PLhZaBW7EbafOPO4YyuovGI1prCViAeVKM&index=9&t=34s

### Var statement

The `var` statement is used to define, and initialise, a new variable.

The name given to the variable must follow the rules for an **Identifer**.

The value to which the new variable is initialised may be a literal value, or a more complex expression. Either way, the resulting value defines the type for that variable.

### Set statement

The `set` statement is used to assign a new value to an existing variable. The new value must be of the same type (or a type compatible with) that of the variable.

A set statement may not assign a new value to a parameter – see **Parameter passing**.

# Conditions & selection

Elan supports the two forms of 'selection' most widely-used in procedural programming: the **If statement** and the **Switch statement**.

(Elan also supports the **If expression**, which, although often thought of as a **Functional programming** technique, may be used within procedural programming also – within any expression.)

### If statement

Explanatory video:
https://www.youtube.com/watch?v=2l4m3Acl_2g&list=PLhZaBW7EbafOPO4YyuovGI1prCViAeVKM&index=13

### Switch statement

Explanatory video:
https://www.youtube.com/watch?v=NdmqUCpNTYQ&list=PLhZaBW7EbafOPO4YyuovGI1prCViAeVKM&index=17&pp=gAQBiAQB

*Case clause*

*Default clause*

A `default` clause may be added only within a switch statement. See **Switch** for more information. If a default statement is used within a switch, there may only be one, and it must follow all the `case` statements.

# Loops & iteration

### For loop

Explanatory video:
https://www.youtube.com/watch?v=D8HF3386FtI&list=PLhZaBW7EbafOPO4YyuovGI1prCViAeVKM&index=12&pp=gAQBiAQB

### Each loop

Explanatory video:
https://www.youtube.com/watch?v=kTMfiH7wXOs&list=PLhZaBW7EbafOPO4YyuovGI1prCViAeVKM&index=14&pp=gAQBiAQB

### While loop

Explanatory video:
https://www.youtube.com/watch?v=Uwp_7Eh2P88&list=PLhZaBW7EbafOPO4YyuovGI1prCViAeVKM&index=15&pp=gAQBiAQB

### Repeat loop

Explanatory video: https://www.youtube.com/watch?v=b-kD417YopM&list=PLhZaBW7EbafOPO4YyuovGI1prCViAeVKM&index=16&pp=gAQBiAQB

# Procedure

The main routine may delegate work to one or more procedures, and for this reason procedures are often described as sub-routines. A procedure may also delegate work to other procedures.

Like a function, a procedure may define parameters; unlike a function, a procedure does not return a value. However, unlike a function, a procedure can have 'side effects' - indeed it must have side-effects otherwise there would be no point in calling it! For this reason the statements within a procedure can:

- Include `print` statements (or methods).
- include `input` methods or other 'system' methods (such as random number generation).
- `call` other procedures (or itself if 'recursion' is required).
- Modify a parameter, provided that parameter definition is preceded by the keyword `out` (see **Function**
- **A** function is a method that returns a value, of a specified type, that is derived from the argument values which are passed into the function and are captured by the parameters. Example:

```
function binarySearch(list as [String], item as String) return Boolean
  var result set to false
  if list.length() > 0
    then
      var mid set to list.length() div 2
      var value set to list[mid]
      if item is value
        then
          set result to true
        else if isBefore(item, value)
          set result to binarySearch(list[..mid], item)
        else
          set result to binarySearch(list[mid + 1..], item)
      end if
  end if
  return result
end function
```

Functions (in contrast to procedures) may be called (evaluated) within an expression, or as an expression, for example:

```
var result set to binarySearch(fruit, wanted)
```

**Notes**:

- The function must contain one and only one `return` statement, which must the last statement in the function. This is automatically created by the function frame and may not be moved or deleted.
- Simple functions may consist only of the a return statement incorporating an expression. The following example uses an 'if expression' :

  ```
  function north(cellNo as Int) return Int
    return if cellNo > 39 then cellNo - 40 else cellNo + 1160
  end function
  ```
- The return *type* is specified in the function signature (the top-line). The return statement must return a value compatible with this type.
- Each parameter defined in the function signature must specify a name, and a type separated by the `as` keyword. Multiple parameter definitions are separated by commas.
- A function may not:
  - Create any side effect. This includes not using the `print` or `input` methods, creating random numbers (though a random number may be created outside the function and passed in as a `Int` or `Float` value), or making a `call` to any procedure (since procedures may, and typically will, have side effects).
  - Re-assign (`set`) any parameter. Note that, unlike in a procedure, a parameter definition on a function may not use the `out` keyword.
  - Modify the contents of an `Array` or `Dictionary` using an index. (However, a function may 'modify' the contents of a List (or `ImmutableDictionary`)– because the supporting methods for those operations return a new data structure.
  - A function may define no parameters, but in this case it is really acting like a `constant` – always returning the same value. (However, unlike an actual `constant`, a parameter-less `function` may calculate its fixed value by making use of other functions and constants.)

- Parameter passing).

Example:

```
procedure inPlaceRippleSort(out list as [Int])
  var changes set to true
  var lastComp set to list.length() - 2
  repeat
    set changes to false
    for i from 0 to lastComp step 1
      if list[i] > list[i + 1]
        then
          var temp set to list[i]
          set list[i] to list[i + 1]
          set list[i + 1] to temp
          set changes to true
      end if
    end for
    set lastComp to lastComp - 1
  end repeat when not changes
end procedure
```

Procedures are used within a `call` statement, for example:

```
main
  var li set to [3, 6, 1, 0, 99, 4, 67]
  call inPlaceRippleSort(li)
  print li
end main
```

**Notes**:

- Parameters for a procedure are defined exactly the same way as for a function.

## Call statement

A `call` statement is used to call a standalone **Input**/Output

## Printing to the Console

Explanatory video:
https://www.youtube.com/watch?v=NGYQQeAuKAg&list=PLhZaBW7EbafOPO4YyuovGI1prCViAeVKM&index=20&t=2s

The simplest way to print is with the print statement.

- See also **print & printTab** (standard library procedures).

## Inputting data from the keyboard

Explanatory video:
https://www.youtube.com/watch?v=ziYfalHJ9q4&list=PLhZaBW7EbafOPO4YyuovGI1prCViAeVKM&index=21

See **Input methods**

Also the `readKey` system method on `BlockGraphics`

## Simple BlockGraphics

Explanatory video:
https://www.youtube.com/watch?v=JbLtgVil7fI&list=PLhZaBW7EbafOPO4YyuovGI1prCViAeVKM&index=22

See **On BlockGraphics**

## Reading and writing data files

See
**The limitation of this** simple approach is that – because these methods depend on an unseen random number generator that changes state each time either method is called – the methods may only be used within `main` or a `procedure`. You may generate random numbers in this way and pass the resulting number(s) into a `function` as `Float` or `Int` Type(s), but you may not generate a random number using these methods include a `function`.

*Generating random numbers within a function*

It is possible to create and use random numbers within a function, but it requires a different approach and is a little more complex, using a special *type* named `Random` (note that the R is capitalised). You must always create the *first* `Random` (using the system method `firstRandom`) within `main` or a `procedure` but you can pass this into a function and generate further random numbers within that function – because the `next` number is generated in a deterministic fashion from the previous `Random` number, using a 'pseudo random sequence'. Example of use:

```
main
  var rnd set to firstRandom()
  var dice set to 0
  for i from 1 to 10 step 1
    set (dice, rnd) to rollDice(rnd)
  end for
end main

function rollDice(rnd as Random) return (Int, Random)
  return (rnd.valueInt(1, 6), rnd.next())
end function
```

Note that calling `firstRandom()` uses the system clock as the 'seed' value, and hence will be different each time you run the program. It is also possible to create a pseudo-random sequence that will be the same each time the program is run – and this is primarily intended for testing purposes. To create the fixed sequence replace `firstRandom()` with `firstRandomInFixedSequence()`, but leave the rest of your code unchanged.

Reading & writing data files

# Using variables

Explanatory video:
https://www.youtube.com/watch?v=g6Byq0vhYw8&list=PLhZaBW7EbafOPO4YyuovGI1prCViAeVKM&index=9&t=34s

## Var statement

The `var` statement is used to define, and initialise, a new variable.

The name given to the variable must follow the rules for an **Identifer**.

The value to which the new variable is initialised may be a literal value, or a more complex expression. Either way, the resulting value defines the type for that variable.

## Set statement

The `set` statement is used to assign a new value to an existing variable. The new value must be of the same type (or a type compatible with) that of the variable.

A set statement may not assign a new value to a parameter – see **Parameter passing**.

# Conditions & selection

Elan supports the two forms of 'selection' most widely-used in procedural programming: the **If statement** and the **Switch statement**.

(Elan also supports the **If expression**, which, although often thought of as a **Functional programming** technique, may be used within procedural programming also – within any expression.)

## If statement

Explanatory video:
https://www.youtube.com/watch?v=2l4m3Acl_2g&list=PLhZaBW7EbafOPO4YyuovGI1prCViAeVKM&index=13

## Switch statement

Explanatory video:
https://www.youtube.com/watch?v=NdmqUCpNTYQ&list=PLhZaBW7EbafOPO4YyuovGI1prCViAeVKM&index=17&pp=gAQBiAQB

*Case clause*

*Default clause*

A `default` clause may be added only within a switch statement. See **Switch** for more information. If a default statement is used within a switch, there may only be one, and it must follow all the `case` statements.

# Loops & iteration

## For loop

Explanatory video: https://www.youtube.com/watch?v=D8HF3386FtI&list=PLhZaBW7EbafOPO4YyuovGI1prCViAeVKM&index=12&pp=gAQBiAQB

## Each loop

Explanatory video: https://www.youtube.com/watch?v=kTMfiH7wXOs&list=PLhZaBW7EbafOPO4YyuovGI1prCViAeVKM&index=14&pp=gAQBiAQB

## While loop

Explanatory video: https://www.youtube.com/watch?v=Uwp_7Eh2P88&list=PLhZaBW7EbafOPO4YyuovGI1prCViAeVKM&index=15&pp=gAQBiAQB

## Repeat loop

Explanatory video: https://www.youtube.com/watch?v=b-kD417YopM&list=PLhZaBW7EbafOPO4YyuovGI1prCViAeVKM&index=16&pp=gAQBiAQB

Procedure, for example:

```
main
  var li set to [3, 6, 1, 0, 99, 4, 67]
  call inPlaceRippleSort(li)
  print li
end main
```

It may also be used to call a **Procedure method** on an instance of a class, for example:

```
call snake.clockTick(currentDirection, apple)
```

**Notes:**

- A call statement cannot be used to call a function.
- The procedure call cannot form part of an expression (because it does not produce a value)

See also **Function**

**A** function is a method that returns a value, of a specified type, that is derived from the argument values which are passed into the function and are captured by the parameters. Example:

```
function binarySearch(list as [String], item as String) return Boolean
  var result set to false
  if list.length() > 0
    then
      var mid set to list.length() div 2
      var value set to list[mid]
      if item is value
        then
          set result to true
        else if isBefore(item, value)
          set result to binarySearch(list[..mid], item)
        else
          set result to binarySearch(list[mid + 1..], item)
      end if
  end if
  return result
end function
```

Functions (in contrast to procedures) may be called (evaluated) within an expression, or as an expression, for example:

```
var result set to binarySearch(fruit, wanted)
```

**Notes**:

- The function must contain one and only one `return` statement, which must the last statement in the function. This is automatically created by the function frame and may not be moved or deleted.
- Simple functions may consist only of the a return statement incorporating an expression. The following example uses an 'if expression' :

  ```
  function north(cellNo as Int) return Int
    return if cellNo > 39 then cellNo - 40 else cellNo + 1160
  end function
  ```
- The return *type* is specified in the function signature (the top-line). The return statement must return a value compatible with this type.
- Each parameter defined in the function signature must specify a name, and a type separated by the `as` keyword. Multiple parameter definitions are separated by commas.
- A function may not:
  - Create any side effect. This includes not using the `print` or `input` methods, creating random numbers (though a random number may be created outside the function and passed in as a `Int` or `Float` value), or making a `call` to any procedure (since procedures may, and typically will, have side effects).
  - Re-assign (`set`) any parameter. Note that, unlike in a procedure, a parameter definition on a function may not use the `out` keyword.
  - Modify the contents of an `Array` or `Dictionary` using an index. (However, a function may 'modify' the contents of a List (or `ImmutableDictionary`)– because the supporting methods for those operations return a new data structure.
  - A function may define no parameters, but in this case it is really acting like a `constant` – always returning the same value. (However, unlike an actual `constant`, a parameter-less `function` may calculate its fixed value by making use of other functions and constants.)

Parameter passing

# Function

A function is a method that returns a value, of a specified type, that is derived from the argument values which are passed into the function and are captured by the parameters. Example:

```
function binarySearch(list as [String], item as String) return Boolean
  var result set to false
  if list.length() > 0
    then
      var mid set to list.length() div 2
      var value set to list[mid]
      if item is value
        then
          set result to true
        else if isBefore(item, value)
          set result to binarySearch(list[..mid], item)
        else
          set result to binarySearch(list[mid + 1..], item)
      end if
  end if
  return result
end function
```

Functions (in contrast to procedures) may be called (evaluated) within an expression, or as an expression, for example:

```
var result set to binarySearch(fruit, wanted)
```

**Notes**:

- The function must contain one and only one `return` statement, which must the last statement in the function. This is automatically created by the function frame and may not be moved or deleted.
- Simple functions may consist only of the a return statement incorporating an expression. The following example uses an 'if expression' :

  ```
  function north(cellNo as Int) return Int
    return if cellNo > 39 then cellNo - 40 else cellNo + 1160
  end function
  ```
- The return *type* is specified in the function signature (the top-line). The return statement must return a value compatible with this type.
- Each parameter defined in the function signature must specify a name, and a type separated by the `as` keyword. Multiple parameter definitions are separated by commas.
- A function may not:
  - Create any side effect. This includes not using the `print` or `input` methods, creating random numbers (though a random number may be created outside the function and passed in as a `Int` or `Float` value), or making a `call` to any procedure (since procedures may, and typically will, have side effects).

- Re-assign (`set`) any parameter. Note that, unlike in a procedure, a parameter definition on a function may not use the `out` keyword.
- Modify the contents of an `Array` or `Dictionary` using an index. (However, a function may 'modify' the contents of a List (or `ImmutableDictionary`)– because the supporting methods for those operations return a new data structure.
- A function may define no parameters, but in this case it is really acting like a `constant` – always returning the same value. (However, unlike an actual `constant`, a parameter-less `function` may calculate its fixed value by making use of other functions and constants.)

# Parameter passing

The arguments provided to a method (function or procedure) are *always* passed 'by value' and not 'by reference'.

However, *if*...

- the method is a `procedure` AND
- the type of the argument is a 'reference type' AND
- that type is *mutable* such as an `Array`, a `Dictionary`, or a user-defined `class` that is not specified as `immutable`.

then it is possible to *mutate* the parameter that holds that argument within the procedure, such that any reference to the argument outside the procedure will 'see' the changes. However, this can only be done in a procedure (not a function) and if the parameter definition is prefixed by the keyword `out`.

A good example of this is an 'in-place sort' procedure. In the following code the `arr` parameter is mutated in the two highlighted lines:

```
procedure inPlaceRippleSort(out arr as [Int])
  var changes set to true
  var lastComp set to arr.length() - 2
  repeat
    set changes to false
    for i from 0 to lastComp step 1
      if arr [i] > arr [i + 1]
        then
          var temp set to arr [i]
          set arr[i] to arr[i + 1]
          set arr[i + 1] to temp
          set changes to true
      end if
    end for
    set lastComp to lastComp - 1
  end repeat when not changes
end procedure
```

Note however that:

- In a `function` you may not mutate *any* parameter
- In a `procedure` you may not *re-assign* any parameter

# Error/Exception handling

### Try statement

### Throw statement

# Comments

Explanatory video:
https://www.youtube.com/watch?v=Vv2hD3EobKU&list=PLhZaBW7EbafOPO4YyuovGI1prCViAeVKM&index=11

Comments:

- may be added at global level – as well as within other constructs.
- always start with a **#** followed by a space and then free-form text. The text field may be left empty
- are a single line, though if the text is long enough the line may be wrapped within the editor
- are always on their own line. It is it not possible to add a comment after, or within, a line of code.

# Object-oriented programming

## Class

A class is user-defined type – offering far richer capability than an `enum`.

### Definition

Here is an example of class definition – taken from the *Snake OOP* demo program:

```
class Apple
  constructor(board as Board)
    set property.board to board
  end constructor

  property board as Board

  property location as Square

  procedure newRandomPosition(snake as Snake)
    repeat
      var ranX set to randomInt(0, board.width - 1)
      var ranY set to randomInt(0, board.height - 1)
      set location to new Square(ranX, ranY)
    end repeat when not snake.bodyCovers(location)
  end procedure

  function updateGraphics(gr as BlockGraphics) return BlockGraphics
    return gr.withBlock(location.x, location.y, red)
  end function

end class
```

**Notes:**

A class *must* have:

- A name that, like any other type, must begin with a capital letter.
- A `constructor` (added automatically by the class frame), which may be used for setting up the values of properties. The `constructor` may, optionally, define parameters – to force the calling code to provide initial values. However, it is not necessary to write any code within the constructor if you have no need to initialise properties. Code in the constructor may make use of any functions – and follows the same constraints as a `function` (i.e. may not call any procedure, whether defined on the class or outside).

A class *may* define:

- Properties – see **Property**
- Function methods – see **Function method**
- Procedure methods – see **Procedure method**

## Inheritance

A regular (concrete) class may inherit from one or more *abstract* classes – see **Abstract clas**. The concrete class must define for itself a concrete version of every abstract property and abstract method defined in the abstract class(es) that it inherits from.

## Using a class

A class is instantiated using the keyword `new` followed by the class name and brackets, which should enclose the comma-separated arguments required to match the parameters (if any) defined on the constructor for that `class`. For example (also from the *Snake OOP* demo):

```
var board set to new Board(40, 30)
var currentDirection set to Direction.up
var snake set to new Snake(board, currentDirection)
var apple set to new Apple(board)
```

The created instance may then be used within expressions, like any other variable.

## Abstract class

An immutable class is created by selecting the word **`immutable`** in front of the keywork class *when* the class is created using the frame. (It is not possible to make a class immutable retrospectively. However it is possible to create a new immutable class and then cut and paste suitable methods and properties from an existing class.

**Notes:**

- An immutable class may define only properties and function methods.
- Properties may be set only within the constructor. Thereafter an instance of an immutable class may only be 'modified' by a function method that returns a copy of the current instance with specified differences – see below.
- Each property must be of an immutable type e.g. a simple value type, immutable list, or another (or same) immutable class.
- An abstract class may inherit from one or more other abstract classes.

Not yet implemented – the **`copy … with`** expression.

## Immutable class

An immutable class is created by selecting the word **`immutable`** in front of the keywork class *when* the class is created using the frame. (It is not possible to make a class immutable retrospectively. However it is possible to create a new immutable class and then cut and paste suitable methods and properties from an existing class.

**Notes:**

- An immutable class may define only properties and function methods.
- Properties may be set only within the constructor. Thereafter an instance of an immutable class may only be 'modified' by a function method that returns a copy of the current instance with specified differences – see below.
- Each property must be of an immutable type e.g. a simple value type, immutable list, or another (or same) immutable class.

# Property

Examples:

```
property height as Int
property board as Board
property head as Square
property body as [Square]
```

- A property is defined on a **Class** and must specify a name (conforming to **Identifer** rules) and a Type.
- A property may be marked `private` – in which case it is visible only by code within the class.
- If not marked `private`, a property may be read – but may not be written. Properties may only be modified from outside the class by means of a **Procedure method**.
- A property may be given an initial value in the `constructor`.
- If the `property` is not initialised within the constructor then it will automatically be given the `empty` value for that type. You may test whether a property contains this default value by writing e.g.:

  ```
  if head is empty Square
  ```

- If a variable or parameter is defined within a method on the class with the same name as a property then the parameter/variable will take precedence. However, you may disambiguate between a property and parameter/variable with the same name using the `property.` qualifier. This is a commonly used in 'set' procedure methods', and in the constructor, for example:

```
constructor(board as Board)
    set property.board to board
end constructor

procedure setHeight(height as Int)
  set property.height to height
end procedure
```

# Function method

A function method follows the same syntax and rules as a freestanding (global) `function`. The differences are:

- A function method is always referenced (used) by code outside the class using 'dot-syntax' on an instance.
- A function method may directly reference (read only) any `property` defined on the class as though it were a variable/parameter.

## asString() method

- `asString` method. This is just a regular function method with a specific name, no parameters and returning a `String`. If defined for a class, then if an instance of the class is printed, the `asString` function method will automatically be used. Typically `asString`

will return a string made up of one or more of the property values, perhaps with additional text, or the results of function calls.

# Procedure method

A ‘procedure method’ follows the same syntax and rules as a freestanding (global) `procedure`. The differences are:

- A procedure method, like a function method, is always referenced (used) by code outside the class using ‘dot-syntax’ on an instance.
- A procedure method may read, or write to, any `property` defined on the class.

# Functional programming

Elan is designed to support the 'functional programming' paradigm.

Unlike in most 'mixed-paradigm' programming languages, *all* functions in Elan are 'pure functions': Elan does not permit any function to create 'side-effects', and enforces that the returned value is derived solely, and deterministically, from the values passed into the function's parameters. This applies whether or not you are actively seeking to write code according to the functional programming approach.

When writing code according to the functional programming paradigm the aim is to write as much as possible of the program's logic and behaviour within pure functions; to use the **Main routine** and **Procedure** calls solely for implementing input/output; and to keep both main and procedures 'as thin as possible'. Elan's in-built support for character-mapped `BlockGraphics` is a good example of this pattern: almost all the work can be done using the in-built *functions*, such as `withBlock`, which may be used within your own user-defined functions. Only the `draw` method – which is the only one that actually changes the display - is a `procedure`, and this must be called from within `main`, or a `procedure`. (See **On BlockGraphics**.)

*Although it is not a requirement to do so*, adopting the functional programming paradigm also means that, wherever possible, functions should avoid using procedural code constructs: sequence, loop, and branch. Here are some examples of functions that *don't* use any of those procedural code constructs:

```
function w(c as Int) return Int
  return if (c mod 40) > 0 then c - 1 else c + 39
end function

function possibleAnswersAfterAttempt(prior as {String}, attempt as String, mark as
String) return {String}
  return prior.filter(lambda w as String => markAttempt(attempt, w) is
mark).asList()
end function

function nextGeneration(cells as [Boolean]) return [Boolean]
  let cellRange be range(0, cells.length() - 1)
  let next be cellRange.map(lambda n as Int => nextCellValue(cells, n))
  return next.asArray()
end function
```

In the examples above we can see several patterns/techniques that are widely used in functional programming in place of procedural code constructs:

- (Top example) Use of an **If expression**, instead of using an **If statement**.
- (Middle example) Use of **Higher order functions** – in this case, `filter` – together with a **Passing** a function as a reference

If you are passing a reference to a freestanding function as an argument into a HoF (as distinct from defining a `lambda`) then you provide the name of that function, but precede it with the keyword `function`. For example:

…

```
var passes set to allPupils.filter(function passedMathsTest)
…
function passedMathsTest(p as Pupil) as Boolean
  return p.mathsPercent > 35
end function
```

**Notes:**

- When passing in a reference function `passMathsTest`, the name is preceded by `function`, and that no parameters (or brackets) are added to the name as they would have been if you were *evaluating* (calling) the function at that point.
- Lambda, instead of writing a loop
- Use of a **Let statement** (instead of the **Var statement**) to calculate intermediate values.

These are explained below.

# If expression

The 'if expression' is *in certain respects* similar to an **If statement**, but with the following differences:

- It is written entirely within a single expression. This is possible because the `if` expression always returns a value.
- There is always a single `then` and a single `else` clause, and each clause contains just a single expression. The `if` expression returns the result of evaluating one of these two expressions, according to whether the condition evaluates to `true` or `false`.
- These `if` expressions may be 'nested' within each other, using brackets around each nested `if` expression where there could be any ambiguity.
  Some more examples:

```
return if c < 1160 then c + 40 else c – 1160
return if isGreen(attempt, target, n) then setChar(attempt, n, "*") else attempt
return if attempt[n] is "*" then attempt else (if isYellow(attempt, target, n)
then setChar(attempt, n, "+") else setChar(attempt, n, "_"))
```

The last example contains a nested `if` expression.

# Let statement

A `let` statement may be used only within a `function`, where its purpose is to calculate an intermediate result for use within one or more subsequent expressions. This may be for any of the following reasons:

- To avoid duplicating code, where the same sub-expression would otherwise be written more than one
- To break up a complex expression just for clarity or readability
- To overcome the restriction (in Elan) that

- Dot methods calls may not be chained.

Here is an example of `let` statements in use:

```
  let wordCounts be allRemainingWordCounts(possAnswers, possAttempts)
  let best be wordCounts.reduce(wordCounts.head(), lambda bestSoFar as WordCount,
newWord as WordCount => betterOf(bestSoFar, newWord, possAnswers))
  return best.word
```

You are never *required* to use a `let` statement: you may always use a `var` instead. But if you are willing to use `let` where you can, it is considered a good practice in functional programming. `let` may be thought of as somewhere between a Constant and a Variable but also has unique characteristics:

- Like a variable and a constant, a `let` statement defines a new named value.
- Like a constant, but unlike a variable, the named value defined by a `let` may not be subsequently re-assigned.
- Unlike a constant (which may only be defined at global level) a `let` is defined within `main` or any method.
- Unlike a constant, the value specified in a `let` may be defined by an expression i.e. may make use of other variables and constants.

# Higher order functions (HoFs)

A 'higher order function' is one that takes in a reference to another function as a parameter, or (less commonly) that returns a reference to another function as its result.

## Standard HoFs

The Elan standard library contains several HoFs that are widely recognised and used within functional programming. See On any Iterable - Higher order functions (HoFs)

## Passing a function as a reference

If you are passing a reference to a freestanding function as an argument into a HoF (as distinct from defining a `lambda`) then you provide the name of that function, but precede it with the keyword `function`. For example:

```
…
var passes set to allPupils.filter(function passedMathsTest)
…
function passedMathsTest(p as Pupil) as Boolean
  return p.mathsPercent > 35
end function
```

**Notes:**

- When passing in a reference function `passMathsTest`, the name is preceded by `function`, and that no parameters (or brackets) are added to the name as they would have been if you were *evaluating* (calling) the function at that point.

## Lambda

A lambda is lightweight means to define a function 'in line'. You typically define a `lambda`:

- If the functionality it defines is needed only in one location - typically for a particular call to a HoF.
- If you need to capture a local variable in the implementation. (This is called 'closing around a variable')

The syntax for a `lambda` is as follows:

- Start with the keyword `lambda`
- Parameter definitions, comma-separated, follow the same form as parameter definitions in a function or procedure – but with no surrounding brackets.
- The `=>` symbol, which is usually articulated as 'returns' or 'yields' or even 'fat arrow'.
- An expression that makes use of the parameter(s) – and may also make use of other variables in scope.

Example:

```
function liveNeighbours(cells as [Boolean], c as Int) return Int
  let neighbours be neighbourCells(c)
  let live be neighbours.filter(lambda i as Int => cells[i])
  return live.length()
end function
```

**Notes**:

- Although a lambda is commonly defined 'inline' (as shown above) it is possible to assign a lambda to a variable and hence to re-use it within the scope of that variable.
- Although a lambda will *usually* define at least one parameter, it is possible to define a lambda with no parameter, just returning an expression – in which case it acts just like a locally defined variable, but with the advantage (useful in rare circumstances) that the expression is evaluated 'lazily' i.e. only when the lambda is *used*. The following example uses both these techniques within a function:

```
function safeSquareRoot(x as Float) return Float
  let root be lambda => sqrt(x)
  return if x < 0 then 0 else root()
end function
```

## Defining your own Hofs

TODO

The `Iter` type

The `Func` type

# Working with immutable types

You may pass mutable and/or immutable types into a function. However, you may not call any procedure on, not otherwise mutate any of those values passed as parameters.

That said, if you are wanting to learn to write code according to the the functional programming paradigm, it is desirable to work exclusively with immutable types. Elan provides very good support for these, both in the form of standard immutable data structures (initially Error! Reference source not found. and **ImmutableDictionary**) and user-defined **Immutable class**.

# Copy...with

NOT YET IMPLEMENTED

the `copy…with` syntax provides an easy-to-use mechanism for 'updating' one or more properties on an instance of an **Immutable class**. It returns a new instance that is a copy of the original, with all properties the same except where specified otherwise (in the '`to` clauses' following the `with`). This general-purpose mechanism avoids the need to write an individual **Function method** for updating each property individually and copying the others, or a constructor that requires all properties.

When implemented, the syntax will be:

```
let d2 be copy d with height to 3, width to 4
```

where `height` and `width` are both properties of `d`. You can specify as many properties to change as required, each '`to` clause' separated by a comma from the previous.

# Tests

Explanatory video:
https://www.youtube.com/watch?v=nz2JUtFEumc&list=PLhZaBW7EbafOPO4YyuovGI1prCViAeVKM&index=30

Example of a test method:

```
test binarySearch
  var li1 set to ["lemon", "lime", "orange"]
  assert binarySearch(li1, "lemon") is true pass
  assert binarySearch(li1, "lime") is true pass
  assert binarySearch(li1, "orange") is true pass
  assert binarySearch(li1, "pear") is false pass
  var li2 set to ["lemon", "orange"]
  assert binarySearch(li2, "lemon") is true pass
  assert binarySearch(li2, "orange") is true pass
  assert binarySearch(li2, "pear") is false pass
  var li3 set to ["lemon"]
  assert binarySearch(li3, "lemon") is true pass
  assert binarySearch(li3, "lime") is false pass
  var li4 set to empty [String]
  assert binarySearch(li4, "pear") is false pass
end test
```

**Notes:**

- Giving a name to a test is optional – you may have multiple test methods with no name. If you do specify a name, it must conform to the rules of any identifier – starting with a lower-case letter. The name may be the same as an existing `function` or `procedure`. There is no potential for confusion, because a `test` method may not be called from within other methods.
- `test` methods may be written anywhere in the code, at the global (file) level.
- A test method may contain multiple `assert` statements. When tests are run, the test runner (part of the Elan IDE) will attempt to run all `assert` statements and show the pass/fail outcome alongside each one. However, if the test hits a runtime error (as distinct from an assert failure) then execution of the test will stop and remaining `assert`s will be shown `as not run`.
- In addition to `assert` statements a test may contain `call`, `var` and `let` statements.

# Assert statement

An `assert` statement may be added only with a `test`. For examples of use see **Property**

Examples:

```
property height as Int
property board as Board
property head as Square
property body as [Square]
```

- A property is defined on a **Class** and must specify a name (conforming to **Identifer** rules) and a Type.
- A property may be marked `private` – in which case it is visible only by code within the class.

- If not marked `private`, a property may be read – but may not be written. Properties may only be modified from outside the class by means of a **Procedure method**.
- A property may be given an initial value in the `constructor`.
- If the `property` is not initialised within the constructor then it will automatically be given the `empty` value for that type. You may test whether a property contains this default value by writing e.g.:

  ```
  if head is empty Square
  ```

- If a variable or parameter is defined within a method on the class with the same name as a property then the parameter/variable will take precedence. However, you may disambiguate between a property and parameter/variable with the same name using the `property.` qualifier. This is a commonly used in 'set' procedure methods', and in the constructor, for example:

```
constructor(board as Board)
    set property.board to board
end constructor

procedure setHeight(height as Int)
  set property.height to height
end procedure
```

# Function method

A function method follows the same syntax and rules as a freestanding (global) `function`. The differences are:

- A function method is always referenced (used) by code outside the class using 'dot-syntax' on an instance.
- A function method may directly reference (read only) any `property` defined on the class as though it were a variable/parameter.

## asString() method

- `asString` method. This is just a regular function method with a specific name, no parameters and returning a `String`. If defined for a class, then if an instance of the class is printed, the `asString` function method will automatically be used. Typically `asString` will return a string made up of one or more of the property values, perhaps with additional text, or the results of function calls.

# Procedure method

A 'procedure method' follows the same syntax and rules as a freestanding (global) `procedure`. The differences are:

- A procedure method, like a function method, is always referenced (used) by code outside the class using 'dot-syntax' on an instance.
- A procedure method may read, or write to, any `property` defined on the class.

# Functional programming

Elan is designed to support the 'functional programming' paradigm.

Unlike in most 'mixed-paradigm' programming languages, *all* functions in Elan are 'pure functions': Elan does not permit any function to create 'side-effects', and enforces that the returned value is derived solely, and deterministically, from the values passed into the function's parameters. This applies whether or not you are actively seeking to write code according to the functional programming approach.

When writing code according to the functional programming paradigm the aim is to write as much as possible of the program's logic and behaviour within pure functions; to use the **Main routine** and **Procedure** calls solely for implementing input/output; and to keep both main and procedures 'as thin as possible'. Elan's in-built support for character-mapped `BlockGraphics` is a good example of this pattern: almost all the work can be done using the in-built *functions*, such as `withBlock`, which may be used within your own user-defined functions. Only the `draw` method – which is the only one that actually changes the display - is a `procedure`, and this must be called from within `main`, or a `procedure`. (See **On BlockGraphics**.)

*Although it is not a requirement to do so*, adopting the functional programming paradigm also means that, wherever possible, functions should avoid using procedural code constructs: sequence, loop, and branch. Here are some examples of functions that *don't* use any of those procedural code constructs:

```
function w(c as Int) return Int
  return if (c mod 40) > 0 then c - 1 else c + 39
end function

function possibleAnswersAfterAttempt(prior as {String}, attempt as String, mark as
String) return {String}
  return prior.filter(lambda w as String => markAttempt(attempt, w) is
mark).asList()
end function

function nextGeneration(cells as [Boolean]) return [Boolean]
  let cellRange be range(0, cells.length() - 1)
  let next be cellRange.map(lambda n as Int => nextCellValue(cells, n))
  return next.asArray()
end function
```

In the examples above we can see several patterns/techniques that are widely used in functional programming in place of procedural code constructs:

- (Top example) Use of an **If expression**, instead of using an **If statement**.
- (Middle example) Use of **Higher order functions** – in this case, `filter` – together with a **Passing** a function as a reference

If you are passing a reference to a freestanding function as an argument into a HoF (as distinct from defining a `lambda`) then you provide the name of that function, but precede it with the keyword `function`. For example:

…

```
var passes set to allPupils.filter(function passedMathsTest)
…
function passedMathsTest(p as Pupil) as Boolean
  return p.mathsPercent > 35
end function
```

**Notes:**

- When passing in a reference function `passMathsTest`, the name is preceded by `function`, and that no parameters (or brackets) are added to the name as they would have been if you were *evaluating* (calling) the function at that point.
- Lambda, instead of writing a loop
- Use of a **Let statement** (instead of the **Var statement**) to calculate intermediate values.

These are explained below.

# If expression

The 'if expression' is *in certain respects* similar to an **If statement**, but with the following differences:

- It is written entirely within a single expression. This is possible because the `if` expression always returns a value.
- There is always a single `then` and a single `else` clause, and each clause contains just a single expression. The `if` expression returns the result of evaluating one of these two expressions, according to whether the condition evaluates to `true` or `false`.
- These `if` expressions may be 'nested' within each other, using brackets around each nested `if` expression where there could be any ambiguity.
  Some more examples:

```
return if c < 1160 then c + 40 else c – 1160
return if isGreen(attempt, target, n) then setChar(attempt, n, "*") else attempt
return if attempt[n] is "*" then attempt else (if isYellow(attempt, target, n)
then setChar(attempt, n, "+") else setChar(attempt, n, "_"))
```

The last example contains a nested `if` expression.

# Let statement

A `let` statement may be used only within a `function`, where its purpose is to calculate an intermediate result for use within one or more subsequent expressions. This may be for any of the following reasons:

- To avoid duplicating code, where the same sub-expression would otherwise be written more than one
- To break up a complex expression just for clarity or readability
- To overcome the restriction (in Elan) that

- Dot methods calls may not be chained.

Here is an example of `let` statements in use:

```
  let wordCounts be allRemainingWordCounts(possAnswers, possAttempts)
  let best be wordCounts.reduce(wordCounts.head(), lambda bestSoFar as WordCount,
newWord as WordCount => betterOf(bestSoFar, newWord, possAnswers))
  return best.word
```

You are never *required* to use a `let` statement: you may always use a `var` instead. But if you are willing to use `let` where you can, it is considered a good practice in functional programming. `let` may be thought of as somewhere between a Constant and a Variable but also has unique characteristics:

- Like a variable and a constant, a `let` statement defines a new named value.
- Like a constant, but unlike a variable, the named value defined by a `let` may not be subsequently re-assigned.
- Unlike a constant (which may only be defined at global level) a `let` is defined within `main` or any method.
- Unlike a constant, the value specified in a `let` may be defined by an expression i.e. may make use of other variables and constants.

# Higher order functions (HoFs)

A 'higher order function' is one that takes in a reference to another function as a parameter, or (less commonly) that returns a reference to another function as its result.

## Standard HoFs

The Elan standard library contains several HoFs that are widely recognised and used within functional programming. See On any Iterable - Higher order functions (HoFs)

## Passing a function as a reference

If you are passing a reference to a freestanding function as an argument into a HoF (as distinct from defining a `lambda`) then you provide the name of that function, but precede it with the keyword `function`. For example:

```
…
var passes set to allPupils.filter(function passedMathsTest)
…
function passedMathsTest(p as Pupil) as Boolean
  return p.mathsPercent > 35
end function
```

**Notes:**

- When passing in a reference function `passMathsTest`, the name is preceded by `function`, and that no parameters (or brackets) are added to the name as they would have been if you were *evaluating* (calling) the function at that point.

## Lambda

A lambda is lightweight means to define a function 'in line'. You typically define a `lambda`:

- If the functionality it defines is needed only in one location - typically for a particular call to a HoF.
- If you need to capture a local variable in the implementation. (This is called 'closing around a variable')

The syntax for a `lambda` is as follows:

- Start with the keyword `lambda`
- Parameter definitions, comma-separated, follow the same form as parameter definitions in a function or procedure – but with no surrounding brackets.
- The `=>` symbol, which is usually articulated as 'returns' or 'yields' or even 'fat arrow'.
- An expression that makes use of the parameter(s) – and may also make use of other variables in scope.

Example:

```
function liveNeighbours(cells as [Boolean], c as Int) return Int
  let neighbours be neighbourCells(c)
  let live be neighbours.filter(lambda i as Int => cells[i])
  return live.length()
end function
```

**Notes**:

- Although a lambda is commonly defined 'inline' (as shown above) it is possible to assign a lambda to a variable and hence to re-use it within the scope of that variable.
- Although a lambda will *usually* define at least one parameter, it is possible to define a lambda with no parameter, just returning an expression – in which case it acts just like a locally defined variable, but with the advantage (useful in rare circumstances) that the expression is evaluated 'lazily' i.e. only when the lambda is *used*. The following example uses both these techniques within a function:

```
function safeSquareRoot(x as Float) return Float
  let root be lambda => sqrt(x)
  return if x < 0 then 0 else root()
end function
```

## Defining your own Hofs

TODO

The `Iter` type

The `Func` type

# Working with immutable types

You may pass mutable and/or immutable types into a function. However, you may not call any procedure on, not otherwise mutate any of those values passed as parameters.

That said, if you are wanting to learn to write code according to the the functional programming paradigm, it is desirable to work exclusively with immutable types. Elan provides very good support for these, both in the form of standard immutable data structures (initially Error! Reference source not found. and **ImmutableDictionary**) and user-defined **Immutable class**.

# Copy...with

NOT YET IMPLEMENTED

the `copy…with` syntax provides an easy-to-use mechanism for 'updating' one or more properties on an instance of an **Immutable class**. It returns a new instance that is a copy of the original, with all properties the same except where specified otherwise (in the '`to` clauses' following the `with`). This general-purpose mechanism avoids the need to write an individual **Function method** for updating each property individually and copying the others, or a constructor that requires all properties.

When implemented, the syntax will be:

```
let d2 be copy d with height to 3, width to 4
```

where `height` and `width` are both properties of `d`. You can specify as many properties to change as required, each '`to` clause' separated by a comma from the previous.

.

The test frame defines two fields:

```
test
  assert result is expected value
end test
```

The result is typically a function call, but may also be a variable defined earlier in the test. The expected value should be a literal value or a variable, both fields should produce the same Type.

**Notes:**

- If the result type is `Float` and you do not wish to specify all decimal places, then you may use the `round` function, for example:
  `assert round(1/3, 2) is 0.33`
- **You cannot call a procedure directly within a float, but you can test some procedures by calling them and then testing the values of any parameters that the procedure is expected to have modified, for example:**

```
var li5 set to [3, 1, 2]
call inPlaceRippleSort(li5)
assert li5 is [1, 2, 3] pass
```

# Types

## Int

An integer is a whole number i.e. with no ‘fractional’ component.

### Type name

`Int`

### Defining a literal integer

```
var meaningOfLife set to 42
```

### Default value

0

### Constraints

- Maximum value: $2^{53} - 1$ which is just over $9 \times 10^{15}$
- Minimum value: $-(2^{53} - 1)$

If either limit is exceeded the number will automatically be represented as a `Float`, with possible loss of precision.

### Notes

- An `Int` may always be passed as an argument into a method that requires a `Float`.

# Float

Float is short for 'floating-point number' – a number that may have both an integer and fractional part.

## Type name

`Float`

## Defining literal floating-point value

```
var a set to 1.618
```

## Constraints

Since Elan compiles to JavaScript, the constraints on floating point numbers are those of JavaScript:

- Maximum value: just over $1 \times 10^{308}$
- Minimum value: approx. $5 \times 10^{-324}$

For greater detail, refer to the official JavaScript documentation

## Notes

- A variable that has been defined as being of type `Float` may not be passed as an argument into a method that requires an `Int`, nor as an index into an `Array`, *even if the variable contains no fractional part*. However, it may be converted into an `Int` before passing, using the functions `floor()` (the integer value left by removing any fractional part) or `ceiling()` (if the `Float` value *does* have a fractional part, the 'ceiling' will the lowest integer greater than the `Float` value).
- If you wish to define a variable to be of type `Float` but initialise it with a whole number then add `.0` on the end of the whole number, for example: `var a set to 3.0`.

# Boolean

A `Boolean` value is either true or false.

## Type name

`Boolean`

## Defining a literal Boolean

```
var a set to true
```

`true` and `false` must be written lower-case

## Default value

`false`

# String

A `String` represents 'text' – a sequence of zero or more characters.

## Type name

`String`

## Defining a literal string value

```
var a set to "Hello"
```

String are always delineated by double-quote marks

## Default value

`""` – known as 'empty string'.

## Notes

- As on most programming languages, strings are *immutable*. When you apply any operation or function with the *intent* of modifying an existing string, the existing string is never modified. Instead the function or operation will return a *new* string that is based on the original, but with the specified differences.
- Strings may be appended using the plus operator, for example `print "Hello" + " " + "World"`.
- A newline may be inserted within a string as `\n`, for example: `print "Hello\nWorld"`.
- You may insert single-quote marks – `'` – within a string.
- Elan strings are automatically interpolated: you may insert the values of variables, or simple expressions within a string, by enclosing them in curly-braces. For example (assuming that the variables `a` and `b` are already defined as integers) :
  ```
  print "{a} times {b} equals {a*b}."
  ```
- It is not *currently* possible to include `"`, `{`, or `}` *directly* within a literal string. However they can be inserted into a string by creating the character from the Unicode, for example:
  ```
  print "This is a double quote mark: " + stringForUnicode(34)
  ```
  Or even by inserting the unicode within curly-braces:
  ```
  print "{stringForUnicode(123)} and {stringForUnicode(125)}"
  ```

# Regex

The type `Regex` permits a regular expression to be passed as a parameter, or returned by a function.

A regular expression may be defined as a literal, bounded by forward slashes, for example:

```
var email set to /[a-z09_]*/
```

See also: **Regular expressions**

# Date and Time

NOT YET IMPLEMENTED

# Arrays and Lists

## Quick reference

| | Array | List |
|---|---|---|
| Type form | `[String]` | `{String}` |
| Literal | `["plum", "pear"]` | `{"plum", "pear"}` |
| Literal empty | `empty [String]` | `empty {String}` |
| Initial size (filled with default values) | `var a set to createArray(10, 0)`<br>2D: `create2DArray(8, 8, "")`<br>In each case, the last argument is the value to which each element is initialised, and defines the type of elements in the Array | Not applicable |
| Read from position | `a[3]`<br>2D: `board[3][4]` | `a[3]` |
| Read range | `a[5..9]` | `A[5..9)]` |
| Put a value | `call a.putAt(3, "pear")`<br>2D: `call board.putAt2D(3,4,"K")` | `set a to a.withPutAt(3, "pear)` |
| Append/Prepend | `call a.append("pear")`<br>`call a.prepend("pear")`<br>`call a.appendList(anotherList)`<br>`call a.prependList(anotherList)` | Note that + appends a *list* to a list. So if you wish to append/prepend a single *item* then it should be enclosed in square brackets to make it into a list containing one item.<br><br>Append: `set a to a + {"pear"}`<br>Prepend: `set a to {"pear"} + a`<br>Append/prepend a list:<br>`set x to listA + listb` |
| Insert | `call a.insertAt(3, "pear")` | `set a to a.withInsertAt(3, "pear)` |
| Remove by index | `call a.removeAt(3)` | `set a to a.withRemoveAt(3)` |
| Remove by value | `call a.removeFirst("pear")`<br>`call a.removeAll("pear")` | `set a to a.withRemoveFirst ("p")`<br>`set a to a.withRemoveAll ("p")` |
| Deconstruction into head (first element) and tail (all the rest) | Not applicable | `var x:xs set to myList`<br>`set h:t to myList`<br>discarding either the head or tail:<br>`var _:tail set to myList`<br>`var head:_ set t myList` |

## List

A List is a simple data structure that holds multiple elements of the same type.

A list – just like a `String` – is *immutable*. You can still insert, delete, or change elements in a List, but the methods for these operations do not modify the input list: they return a new list based on the input list but with the specified differences.

*Type name*

The type is specified in the following form:

`{String}` for a list of type String
`{Int}` for a list of type Int
`{{Int}}` for a list of lists of type Int

Note that the syntax for the alias type name is similar to that for an `Array` but using curly braces `{ }` instead of square brackets `[ ]`. The same is true for literals...

*Defining a literal*

```
var fruit set to {"apple", "orange", "pear"}
```

*Using a List*

Try these examples:

```
var fruit set to empty {String}
print fruit
set fruit to fruit + "apple"
set fruit to fruit + "pear"
print fruit
set fruit to "orange" + pear
print fruit[0]
print fruit.length()
print fruit[fruit.length() -1]
var head:tail set to fruit
print head
print tail
```

## Array

An 'Array' is a simple data structure that holds multiple elements of the same type.

Unlike a list, an Array is *mutable* – meaning that the elements within the data structure can be altered without creating a new Array from the old.

The type is specified in the following form:

`[String]` for an `Array` of type `String`
`[Int]` for an `Array` of type `Int`

Note that the syntax for the alias type name is similar to that for an `List` but using square brackets `[ ]` instead of curly braces. The same is true for literals...

Where, in this example, `String` represents the type of each element. The element type could be any value type – `Int`, `Boolean`, `Float`, `String` – or the name of a specific `class` such as `Player` or an `enum` such as `Direction`. It may also be another data structure, including another `Array`, (sometimes referred to as a 'nested array') for example:

*Creating an Array*

An Array may be defined in 'literal' form, 'delimited' by square brackets, and with all the required elements are separated by commas. The elements may be literal values (all of the same type):

```
var fruit set to ["apple", "orange", "pair"]
```

including 'nested arrays':

```
var coordinates set to [[3.4, 0.1, 7.8], [1, 0, 1.5], [10, -1.5, 25]]
```

or variables (provided they are all of the same type):

```
var values set to [x, y, z]
```

or a mixture of literal values and variables (all of the same type):

```
var values set to [3.1, y, z]
```

where `y` and `z` are existing variables of type `Float`.

You may also define an array of a specified size, with each element initialised to the same value, for example:

```
var fruit set to createArray(20, "")
```

will create an Array of type `String` with exactly `20` elements, each initialised to an empty `String` and:

```
var scores set to createArray(12, 100.0)
```

will create an Array of type `Float` with exactly `12` elements, each initialised to `100.0`.

*Using an Array*

Elements are read using an index in square brackets – the *first* element being element `[0]`. The last element of an Array of size 10 will therefore be accessed by the index `[9]`.

Attempting to read an element *by index*, where that element does not exist, will result in an 'Index out of range' runtime error.

Unlike in many programming you may *not* modify data by index: elements are *modified* by calling the `putAt` procedure on the array.

Try these examples (the last one will produce an error – make sure you understand why):

```
var a set to new createArray(10, 0)
print a
print a.length()
call a.putAt(0, 3)
call a.putAt(1, 7)
print a
print a[0]
print a[a.length() -1]
print a[a.length()]
```

Unlike in some languages, Elan Arrays may be dynamically extended, using `append` and `prepend` methods.

```
var a set to createArray(3, 0)
var b set to createArray(3, 10)
print a
print b
a.append(3)
b.prepend(7)
print a
print b
a.appendArray(b)
print a
```

## 2-dimensional Array

In Elan, as in many languages, a '2D array' is just an Array of Arrays. However, Elan provides a couple of convenient short-cut methods for working with such data structures:

```
var board set to create2DArray(8, 8, "")
```

will create an Array of Arrays with a total of 64 elements each of type `String`, and initialised to an empty `String`. The type is determined by the type of the third parameter, which might be an `Int`, `Boolean`, or user-defined type. It need not be an empty value. The '2D Array' need not be square.

You can modify individual elements in this data structure using:

```
call board.putAt2D(3,4,"K")
```

and you can read individual elements with a double index, for example:

```
for col from 0 to 7 step 1
  for row from 0 to 7 step 1
    print board[col][row]
  end for
end for
```

# Dictionaries

There are two forms of dictionary in Elan: an ordinary `Dictionary` (which is mutable) and an `ImmutableDictionary`.

## Quick reference

| | Dictionary | Immutable Dictionary |
|---|---|---|
| Type form | `[String:Int]` | `{String:Int}` |
| Literal | `["a":1, "b":4]` | `{"a":1, "b":4}` |
| Literal empty | `empty [String:Int]` | `empty {String:Int}` |
| Read the value for a given key | `d["a"]` | `d.getKey("a")` |
| Get all keys, or all values | `d.keys()` and `d.values()` Both return an immutable list of the appropriate type | |
| Define (or change) a value associated with a key | `set d["c"] to 7` | `set d to d.withKey("c", 7)` |
| Remove both key and value | `call d.removeKey("c")` | `set d to d.withRemoveKey("c")` |

## Dictionary

*Type name*

In the following example, `Int` is the type of the 'key' and `String` is the type of the value associated with a specific key:

```
[Int, String]
```

Important: For both `Dictionary` and `ImmutableDictionary` the value type can be any type, including e.g. a specific type of class, a `List`, another `Dictionary` or another data structure. However, the *key* type must be one of: `Int`, `Float`, `String`, `Boolean`, or a specific type of `enum`.

*Defining a literal*

A literal Dictionary is defined as a comma-separated list of 'key:value pairs' surrounded by square brackets e.g:

```
var scrabbleValues set to ["a":1, "b":3, "c":3, "d":2]
```

*Using a Dictionary*

Try these examples:

```
var dict set to new [String, Int]
print dict
set dict["a"] to 3
print dict["a"]
call dict.removeByKey("a")
print dict
```

*Constraints*

- Key values must be unique
- There is no difference in syntax between *adding* an entry with a new key, and setting a new value for an existing key: if the key does not exist in the dictionary, it will be added.

## ImmutableDictionary

An immutable dictionary may be defined in a `constant`. For examples, see **Snake (OOP)** and **Cards**.

*Type name*

Type name takes the following form:

```
{Int, String}
```

*Defining a literal*

A literal Dictionary is defined as a comma-separated list of 'key:value pairs' surrounded by curly braces e.g:

```
var scrabbleValues set to {"a":1, "b":3, "c":3, "d":2}
```

*Using an ImmutableDictionary*

Try these examples:

```
var immD set to new {String, Int}
print immD
set immD to immD.withKeyValue("a", 3)
print immD.getValueByKey("a")
set immD to immD.withRemoveByKey("a")
print immD
```

# Tuple

A tuple is a way of holding a small number values of *different* types together as a single reference. A common usage scenarios include:

- Holding a pair of x and y coordinates (each a floating point number) as a single unit.
- Allowing a function could pass back a result, together with, say a string message and/or a Boolean flag indicating whether the operation was successful

A tuple is considered a 'lightweight' alternative to defining a specific class *for some purposes*.

## Type name

Written as a comma-separated list of the type of each member, surrounded by round brackets:

```
(Int, Int, Int)
(String, Boolean)
```

## Defining a literal tuple

A tuple is defined, where it is needed, by two or three elements – which variables, or literal values), separated by commas and surrounded by round brackets, for example:

```
var point1 set to (3.769, 4.088)
```

## Using a tuple

- You may pass a tuple into a function, or return one from a function, for example:

  ```
  var d set to distanceBetween(point1, (12.34, 20.0))
  ```
- You may access (read) the individual elements within a tuple using methods `first`, `second`, and `third` for example:

  ```
  var x = point1.first()
  ```

- An existing tuple (`point1` below) may be 'deconstructed' into two new variables:

  ```
  var x, y set to point1
  ```

  or into existing variables of the correct type:

  ```
  var a set to 3
  var b set to 4
  set a, b to point1
  ```

- The 'discard' symbol _ (underscore) may also be used when deconstructing a tuple, if there is no need to capture one (or more) specific elements:

  ```
  var x, _ set to point1
  ```

### Constraints

- Tuples are currently limited to having two or three members, which may be of the same or different types.
- As in most languages, Elan tuples are *immutable*. Once defined they are effectively 'read only': you cannot alter any of the elements in a tuple, nor (unlike an List for example) can you create a new tuple from an existing one with specified differences
- If you invoke the method `third` on a tuple that has only two members you will get a run-time error.
- You cannot deconstruct a tuple into a *mixture* of new and existing variables

# Func

A function may be passed as an argument into another function (or a procedure), or returned as the result of calling another function. This pattern is known as 'higher order function', and is a key idea in the functional programming paradigm. To define a function that takes in another function as a parameter, or returns a function, you need to specify the *type* of the function, just as you would specify the *type* of every parameter and the return type for the function.

### Type name

The *type* of any function starts with the word `Func`, followed by angle brackets defining type of each parameter, and the return type for that function, following this syntax:

```
Func<of String, String, Int => Boolean>
```

The example above defines the type for a function that defines *three* parameters, of type `String`, `String`, and `Int` respectively, and returns a `Boolean` value. For example this type would match that of a function definition that started:

```
Function charactersMatchAt(a as String, b as String, position as Int) return
Boolean
```

# Identifying and comparing types with 'typeof'

The type of a variable (or literal) may be identified by preceding it with the `typeof` operator, which generates a `String` representation of the type. This may also be used to test and compare the type(s) of data items. Try the following:

```
main
  let x be "hello"
  let y be {"apple", "orange", "pair"}
  print typeof x
  print typeof y
  print (typeof x) is (typeof y)
  print (typeof x) is (typeof y[0])
end main
```

# Standard Library Functions

While Elan is still at Beta release, this is a document in progress.  Where explanations are incomplete or missing, you *might* find some assistance by searching for the merh

# Standalone functions

Standalone functions always return a value and are therefore used in contexts that expect a value, such as in the right-hand side of a variable declaration (`var`) or assignment (`set`), either on their own or within a more complex expression. All standalone *library* functions require at least one argument to be passed in brackets – corresponding to the parameters defined for that function.

## unicode

```
stringForUnicode(code as Int) return String
```

Converts a unicode value (expressed in decimal or hexadecimal notation) into a single character string. For example:

```
function hearts() return String
  return unicode(0x2665)
end function
```

## parseAsInt and parseAsFloat

```
parseAsInt(inp as String) return (Boolean, Int)
and
parseAsFloat(inp as String) return (Boolean, Float)
```

`parseAsInt` attempts to parse the input `String` as an `Int`. Returns a 2-tuple, the first value of which is `Boolean`, with `true` indicating whether or not the parse has succeeded, and the second value being the resulting `Int`. `parseAsFloat` does the equivalent for floating point. Usage:

```
print parseAsInt("31") yields (true, 31)
print parseAsFloat("31") yields (true, 31)

print parseAsInt("31.2") yields (false, 0)
print parseAsFloat("31.2") yields (true, 31.2)

print parseAsInt("0") yields (true, 0)
print parseAsInt("0") yields (true, 0)
```

Notes:

- Any string that parses as an `Int` will also parse as a `Float`
- If the parse has failed the second value will default to zero – so you should always check the first value to see if this is a correct parse, or just the default.
- You can 'deconstruct' the tuple into two variables e.g
  `var (outcome, value) = parseAsInt(myString)`

- One usage for these parsing methods is for validating inputs, but note that there is an easier way to do this – see Input methods.

## floor, ceiling, and round

`floor(inp as Float) return Int`
returns the nearest integer value *below* (or equal to) the argument value. Usage:

`print floor(2.5)` yields `2`

`ceiling(inp as Float) return Int`
returns the nearest integer value *above* (or equal to) the input value. Usage:
`print ceiling(2.5)` yields `3`

`round(inp as Float, places as Int) return Float`
Rounds the input number of decimal places specified as the second argument (an `Int`). Usage:

`print round(3.14159, 3)` yields `3.142`

## Maths functions

`pi` -returns the constant `Float` value 3.141592653589793

Each of the following functions takes a single argument of type `Float` and returns a `Float`.
`abs` - returns the absolute value of the input.
`acos` - returns the arccosine of the input, as radians.
`asin` - returns the arcsine of the input value, as radians.
`atan` - returns the arctangent of the input value, as radians.
`acosDeg` - returns the arccosine of the input, as degrees.
`asinDeg` - returns the arcsine of the input, as degrees.
`atanDeg` - returns the arctangent of the input, as degrees.

`cos` - returns the cosine of input interpreted as radians.
`cosDeg` - returns the cosine of input interpreted as degrees.
`exp` - returns $e^x$, where x is the argument, and e is Euler's number (2.718...)
`logE` - returns the natural logarithm of the input.
`log10` - returns the base-10 logarithm of the input.
`log2` - returns the base-2 logarithm of the input.
`sin` - returns the sine of the input interpreted as radians.
`sinDeg` - returns the sine of input interpreted as degrees.
`sqrt` - returns the positive square root of the input.
`tan` - returns the tangent of the input interpreted as radians.
`tanDeg` - returns the tangent of input interpreted as degrees.
`degToRad` – converts input from degrees to radians.
`radToDeg` – converts input from radians to degrees.

Examples of the maths functions being used:

```
test
  assert pi is 3.141592653589793 pass
  assert abs(-3.7) is 3.7 pass
  assert round(acos(0.5), 3) is 1.047 pass
  assert round(asin(0.5), 3) is 0.524 pass
  assert round(atan(1), 2) is 0.79 pass
  assert round(cos(pi/4), 3) is 0.707 pass
  assert round(exp(2), 3) is 7.389 pass
  assert round(logE(7.398), 2) is 2 pass
  assert log10(1000) is 3 pass
  assert log2(65536) is 16 pass
  assert round(sin(pi/6), 2) is 0.5 pass
  assert round(sqrt(2), 3) is 1.414 pass
  assert round(tan(pi/4), 2) is 1 pass
end test
```

## Regular expressions

The method **matchesRegex** is applied to a **String** using dot-syntax and requires a **Regex** parameter, specified as a literal or as variable of **Regex**, for example:

```
test
  var r set to /[A-Z][a-z]*/
  assert "Foo".matchesRegex(r) is true pass
  assert "bar".matchesRegex(r) is false pass
end test
```

## Bitwise functions

```
bitAnd(a as Int, b as Int) return Int
bitOr(a as Int, b as Int) return Int
bitXor(a as Int, b as Int) return Int
bitNot(a as Int) return Int
bitShiftL(a as Int, places as Int) return Int
bitShiftR(a as Int, places as Int) return Int
```

Examples of the bitwise functions being used

```
test bitwise
  var a set to 13
  assert a is 0b1101 pass
  assert a.asBinary() is "1101" pass
  var b set to 30
  assert b is 0b11110 pass
  assert bitAnd(a, b) is 0b1100 pass
  var aob set to bitOr(a, b)
  assert bitOr(a, b) is 0b11111 pass
  var axb set to bitXor(a, b)
  assert bitXor(a, b) is 0b10011 pass
  var nota set to bitNot(a)
  assert bitNot(a) is -14 pass
  var aL set to bitShiftL(a, 2)
  assert bitShiftL(a, 2) is 0b110100 pass
  assert bitShiftR(a, 2) is 0b11 pass
end test
```

The result of `bitNot(a)`, where `a` is `13`, being `-14` might be a surprise. But this is because the bitwise functions assume that the arguments are represented as 32-bit *signed* integers. So 13 is represented as `00000000000000000000000000001101`, applying bit not will give `11111111111111111111111111110010` which is the 32-bit 2s-complement representation of `-14`

## Creating Arrays of specific sizes

The following methods return an Array, of a specified size, and with all elements initialised to a specified value. Although the resulting Array *may* still be expanded subsequently (by using the `add` procedure), the *typical* use for these two methods is for cases that would originally have used a traditional (fixed-size) 'array'.:

```
createArray(size as Int, initialValue as Type) return [Type]
```

where `Type` is one of the following types: `Int`, `Float`, `Boolean`, `String` or any type of `enum`.

There is also a variant of the method that creates a '2-dimensional' rectangular array (actually an `Array` of `Array`s)

```
create2DArray(noOfrows as Int, noOfColumns as Int, initialValue as T) return
[[Type]]
```

See also: Lists - **Quick reference**.

# Standalone procedures

## print & printTab

```
print(arg as String)
printTab(tabPosition as Int, arg as String)
```

These procedures may be called as an alternative to using the `print` *statement*. The differences are that the `print` or `printTab` *procedure*:

- does not automatically add a 'newline' at the end, so you may subsequently print something else on the same line. If you wish to use the `print` procedure and include one or more newlines in specific places, just include \n (the standard form for a newline) within the string.
- Require the data to be printed to be of type `String`. If you want to print a value of another type, you will either need to add `.asString()` to it, or put the value into braces within an 'interpolated' string.

For `print`, the data to be printed is the only argument. For example:

```
for I from 1 to 10 step 1
  call print("{i}")
end for
```

`printTab` helps in the layout of information printed to the console, in particular, the printing of columns of data. `printTab` works like the print procedure, but requires an additional argument specifying the tab position (number of characters from the left of the display). For example:

```
call printTab(0, "No.")
call printTab(10, "Square")
call printTab(20, "Cube\n")
for x from 1 to 10 step 1
  call printTab(0, x.asString())
  call printTab(10, "{x^2}")
  call printTab(20, "{x^3}\n")
end for
```

## pause

```
pause(milliSeconds as Int)
```

Pauses the execution of the program for a specified number of milliseconds (minimum `1`). For example, to pause for 1/10th of a second:

```
call pause(100)
```

There are two uses of pause:

- to control the speed of events – for example in a dynamic game
- to allow the Console and/or BlockGraphics displays to refresh. See Error! Reference source not found. and Error! Reference source not found.. (For this purpose, `pause(1)` is sufficient to enable the display refresh and causes minimum delay in program execution).

### clearConsole

```
procedure clearConsole()
```

Equivalent to pressing the Clear button on the Console, but automatically at specific point(s) in the program execution:

```
call clearConsole()
```

# System methods

'System methods' refers to a set of specific methods provided by the Elan standard library, that depend on the system (outside of the Elan language) in some way. They *appear* to work like functions – in that they return a value – and may be used within an expression in the same way as a regular function, *but may be used only inside* `main` *or a* `procedure`. This is because each system method has a dependency on something more than the arguments (if any) passed into it, and/or generates side effects. Thus, system methods are *not* 'pure' functions.

## Input methods

Input methods provide some validation of the input type and, optionally, values. Each of these methods, also defines a `prompt` string, which will be printed immediately before, and on the same line, as the input cursor – and will be repeated if a given user-input is not valid.

```
inputString(prompt as String) return String
inputStringWithLimits(prompt as String, minLength as Int, maxLength as Int) return
String
inputStringFromOptions(prompt as String, options as {String}) return String
inputInt(prompt as String) return Int
inputIntBetween(prompt as String, min as Int, max as Int) return Int
inputFloat(prompt as String) return Float
inputFloatBetween(prompt as String, min as Float, max as Float) return Float
```

Examples of use:

```
var name set to inputString("Your name? ", 2, 50)

var name set to inputStringWithLimits("Your name? ", 2, 50)

var action set to inputStringFromOptions("Action ?", {"add", "remove",
"exit")?
```

Note that the options must be specified as an *immutable* list i.e. within curly braces if specified as a literal list, as above.

```
var moveSquares set to inputInt("Move squares)

var age set to inputInt("Your age in years? ", 5, 21)

var payment set to inputFloatBetween("Payment value: ", 0.0, 99.99)
```

## Clock

```
clock() return Int
```

The clock methods returns an integer representing the current time in milliseconds since 'the epoch' (midnight, January 1, 1970 UTC). This is useful for measuring elapsed time. For example:

```
var startTime set to clock()
# Your code here
print "Elapsed time in milliseconds {clock - startTime}"
```

## Random numbers

There are two ways to create random numbers in Elan. The first is the simpler to understand and uses either of these two system method:

```
random() return Float
```

returns a random `Float` in between 0 and 1, and:

```
randomInt(min as Int, max as Int) return Int
```

takes two integer arguments – a lower-bound followed by an upper-bound - and returns a random integer value between those two (inclusive) values.

Examples of use:

```
var probability set to random()
var dice set to randomInt(1, 6)
```

The limitation of this simple approach is that – because these methods depend on an unseen random number generator that changes state each time either method is called – the methods may only be used within `main` or a `procedure`. You may generate random numbers in this way and pass the resulting number(s) into a `function` as `Float` or `Int` Type(s), but you may not generate a random number using these methods include a `function`.

*Generating random numbers within a function*

It is possible to create and use random numbers within a function, but it requires a different approach and is a little more complex, using a special *type* named `Random` (note that the R is capitalised). You must always create the *first* `Random` (using the system method `firstRandom`) within `main` or a `procedure` but you can pass this into a function and generate further random numbers within that function – because the `next` number is generated in a deterministic fashion from the previous `Random` number, using a 'pseudo random sequence'. Example of use:

```
main
  var rnd set to firstRandom()
  var dice set to 0
  for i from 1 to 10 step 1
    set (dice, rnd) to rollDice(rnd)
  end for
end main

function rollDice(rnd as Random) return (Int, Random)
  return (rnd.valueInt(1, 6), rnd.next())
end function
```

Note that calling `firstRandom()` uses the system clock as the 'seed' value, and hence will be different each time you run the program. It is also possible to create a pseudo-random sequence that will be the same each time the program is run – and this is primarily intended for

testing purposes. To create the fixed sequence replace `firstRandom()` with `firstRandomInFixedSequence()`, but leave the rest of your code unchanged.

## Reading & writing data files

Data file handling is not yet implemented. However, a 'mock-up' of the API for reading data files exists, illustrated here:

```
main
  var file set to openRead(myFile)
  while not file.endOfFile()
    var line set to file.readLine()
  end while
  call file.close()
  print file
end main

constant myFile set to "Now is the time\n for all good men\n to come to the aid\n of the party."
```

When the API is properly implemented, the argument passed into the `openRead` system method (`myFile` in the example above) will specify the filename.

*For the current mockup*, however, the argument must be a reference to a string holding the content of the file – defined as a `constant` within the code, as shown in the example above.

# Dot methods

'dot-methods' are invoked on a variable or property of the appropriate type, using 'dot syntax'.

## On a String

`upper(input as String) as String`
Returns a new string based on the input with all alpha-characters in upper-case.

`lower(input as String) as String`
Returns a new string based on the input with all alpha-characters in upper-case.

`contains(partString as String) return Boolean`
Takes a single parameter of type `String`, and returns a Boolean value indicating whether or not that argument string is contained within the string on which contained was called. Usage:

```
var a set to "Hello World!"
print a.contains("ello")
```
prints `true`

`trim() return String`
returns a new string based on the string on which the method is called, but with any leading or trailing spaces removed.

`indexOf(partString as String) return Int`

The following methods are used for comparing strings alphabetically – for example in a sort routine.

```
isBefore(otherString as String) return Boolean
isAfter(otherString as String) return Boolean
isBeforeOrSameAs(otherString as String) return Boolean
isAfterOrSameAs(otherString as String) return Boolean
```

`asUnicode() return Int`
Returns the Unicode (integer) value for a character. If the string is more than one character long, the Unicode value returned is that for the *first* character in the string only.

## On an Array

In contrast to a `List`, an Elan `Array` is a *mutable* data structure. It has the behaviour of a traditional array, but may also be dynamically extended in size.

See also: **Quick reference**

Functions:

`myArray.contains(item)` returns `true` or `false`

`myArray.asList()` returns a `List` containing the same elements as the `Array` on which the method was called. This is often used to permit an `Array` to be passed into a function that has been designed to accept a `List`.

The following are all procedures, so invoked in a `call` statement, for example:

```
call fruit.append("banana")
call fruit.appendList(anotherList)
call fruit.insertAt(4, "cherry”)
call fruit.prepend("melon”)
call fruit.prependList(anotherList)
call fruit.putAt(2, "grape")
call fruit.removeAll("apple")
call fruit.removeAt(3)
call fruit.removeFirst("apple")
```

### 2D Array

If you have an `Array` of an `Array` of a given type (typically, though not necessarily, created by the `create2DArray` method then you may set a value using `putAt2D`, for example.:

```
call board.putAt2D(3,4,"King")
```

## On List

**Important**: in Elan, a List is *immutable*. Methods never *modify* the `List` on which they are called: instead they return a *new* `List` based on the original but with the specified differences – the same as happens for an ordinary `String`.

See also: Lists - **Quick reference**

These are all functions

`myList.contains(item)` returns `true` or `false`
`myList.asArray()` returns a new `Array` with the same contents as `myList`

The following functions all return a new `List`, copied from the list on which the function was called, but with the differences specified by the function:

```
myList.withInsertAt(4, "cherry”)
myList.withPutAt(2, "grape")
myList.withRemoveAt(3)
myList.withRemoveFirst("apple")
myList.withRemoveAll("apple")
```

## On a Dictionary

See also: Dictionaries - **Quick reference**

```
putAtKey
removeAtKey
keys
values
```

## On a ImmutableDictionary

See also: Dictionaries - **Quick reference**

```
hasKey
withPutAtKey
withRemoveAtKey
```

## On a Tuple

```
first
second
third
```

These methods return a specified element within the tuple (tuples being limited to a maximum of three elements). Usage

```
var t set to (3, "apple", true)
print t.first()
print t.second()
print t.third()
```

## On BlockGraphics

**Important**: BlockGraphics are designed to work best at 100% browser magnification. At larger magnifications, all the blocks will not necessarily be shown; at smaller magnifications there may be gaps between the blocks.

Examples of using the dot-methods on `BlockGraphics` may be found in the following example programs:

- Snake (OOP)
- Life

Both functions and procedures are called on an instance of `BlockGraphics` using dot notation.

*Functions*

Note that because the `BlockGraphics` type is immutable, all functions that appear to modify the contents (i.e. those starting `put...`) return a new instance of type `BlockGraphics` with the difference(s) specified. This instance will need to be assigned either to the original variable, or to a new one.

```
withBlock(x as Int, y as Int, colour as Int) return BlockGraphics
withText(x as Int, y as Int, text as String, foreground as Int, background as Int)
return BlockGraphics
withUnicode(x as Int, y as Int, unicode as Int, foreground as Int, background as
Int) return BlockGraphics
withBackground(backgroundColour as Int) return BlockGraphics
getChar(x as Int, y as Int) return String
getForeground(x as Int, y as Int) return Int
getBackground x as Int, y as Int) return Intbackground
getDetails(x as Int, y as Int) return (String, Int, Int)
```

Note that colours are represented as integer values, corresponding to the standard RGB colour scheme as used in all web graphics. The following basic colours are predefined as global constants: `black`, `grey`, `white`, `red`, `green`, `blue`, `yellow`, `brown`.

Note that when defining your own colours it can be helpful to use the hexadecimal notation, for example:

```
constant lightBlue set to 0x80abff
```

*Procedures*

These procedure are all called on an instance of BlockGraphics using dot notation:

```
draw()
clearGraphics()
clearKeyBuffer()
```

*System methods*

As with other system methods, these may only be called from within `main` or a `procedure`.

```
getKeystroke() return String
getKeystrokeWithModifier() return (String, String)
```

Example use where gr is an instance of type `BlockGraphics`:

```
var k set to gr.getKeystroke()
```

- If the user has pressed a key that key will be returned as a `String`.
- If no key has been pressed, the method will return an empty string: `""`.
- If the key is a printable character, it will be returned as a single-character string, for example: `"a"`, `"X"`,`"3"`,`":"`
- Non-printable keys are returned as words, for example: `"Space"`, `"Backspace"`,`"Enter"`,`"ArrowDown"`
- `getKeystrokeWithModifier` allows you to check whether the keystroke was 'modified' by, for example, the **Shift**, **Ctrl**, or **Alt** keys. The method returns a 2-tuple consisting of the key and the modifier (which may be an empty string). Learn how this works with this example:

```
clearKeyBuffer()
```

All keystrokes go into 'queue' called the keyboard 'buffer'. If you are reading keys (see Error! Reference source not found. ) and you wish to prevent the user from adding keystrokes faster than they can be consumed, then:

```
call clearKeyBuffer()
```

## On an instance of a class

`typeAndProperties` returns a string that summarises the instance on which the function is called, including the name of the class and the value of all properties.

## On any Iterable - Higher order functions (HoFs)

These dot methods are called on any 'iterable' type, which includes concrete Lists, Strings, or iterables returned by other functions. As 'higher order functions' they take either a lambda or a reference to a function as one of their arguments.

**Important:** Several of these methods return an abstract type named `Iterable`. The result may easily be turned into a form that can be printed, or passed into other functions, by appending `.asList()` or `.asArray()` at the end of the expression.

*These are not yet fully documented* but, for readers familiar with HoFs from another language, some examples are shown below.

`Filter`
Usage (from example program

Turing machine):

```
let matches be rules.filter(lambda r as Rule => (r.currentState is
currentState) and (r.currentSymbol is tape[headPosition]))
```

`map`
Usage (from example program Life):

```
let next be cellRange.map(lambda n as Int => nextCellValue(cells, n))
```

`reduce`
Usage (from example program

An implementation of a Turing Machine as a single class, with a separate (immutable) class to capture the transition rules.

Wordle solver):

```
let d2 be possibleAnswers.reduce(d, lambda dd as {String:Int}, possAnswer
as String => incrementCount(dd, possAnswer, attempt))
```

`max` and `min`
Both functions may be applied to an `Iter<Float>` e.g. a `[Float]` or `{Float}` and return the maximum/minimum value found therein.

```
var a set to {33, 4, 0,99, 82, 55}
print "Max: {a.max()} Min: {a.min()}"
```

`maxBy` and `minBy`
Alternative implementations of `max` and `min` that take. Usage:

```
var a set to {33, 4, 0,99, 82, 55}
print a.maxBy(lambda x as Int => x mod 10)
```

`any`
Returns **`true`** or **`false`** indicating whether any of the members of the iterable individually pass the test defined by the lambda (which itself returns a **`Boolean`**). Usage:

```
var a set to {33, 4, 0,99, 82, 55}
print a.any(lambda x as Int => x > 50)
```

`sortBy`

*Additional sort methods will be introduced in a later Beta*.

For now, `sortBy` takes a lambda that takes two arguments (of the same type as that of the iterable being sorted) and compares them, returning an integer, with one of the values -1, 0, 1, to

indicated whether the first argument should be placed before, after, or just adjacent to (does not matter whether before or after) the second argument in the sorted result. Example:

```
var source set to {2, 3, 5, 7, 11, 13, 17, 19, 23, 27, 31, 37}
print source.sortBy(lambda x as Int, y as Int => if x > y then 1 else -1)
```

The following are not HoFs, but are included here because they are most likely to be used *with* one of the HoFs listed above.

`range(first as Int, last as Int) as Iter<of Int>`
returns an literable that will produce all the integer values between the two argument values.

## On many different types

`asString`
`asIter`
`length`

`head` returns the first item in an Array or a List

# Example programs

The Demo button on the Elan Beta provides a range of example programs that you can explore, and run, including the following list. In the full release – Elan v1.0 – the Demo button will, deliberately, not appear by default, but it can be specified (in the editable profile) to be shown if desired.

## Snake (OOP)

An implementation of the popular 'Snake' game – using OOP with classes for `Snake`, `Apple`, `Square`, and `Board`. Uses the BlockGraphics pane.

## Binary Search

An implementation of the 'binary search' algorithm in a single function, with small `main` routine to demonstrate it in use, and automated tests.

## Merge sort

A recursive implementation of the 'merge sort' algorithm in two functions, complete will a small main routine to demonstrate it in use, and automated tests.

## Life

An implementation of Conway's famous 'Game of Life` simulation using the BlockGraphics pane, and with automated tests.

## Turing machine

An implementation of a Turing Machine as a single class, with a separate (immutable) class to capture the transition rules.

## Wordle solver

This program is capable of solving a Wordle, and in 'hard mode' (where each guess must be a possible answer). It solves 99% of all possible Wordle puzzles in 6 or fewer attempts, with an average of just 3.67 attempts.

## Cards

This is not a working card game – just a sketch to show the beginnings of a `Card` class, with associated `enum`s and `constant`s, that may easily be rendered on the BlockGraphics display.

# Index to all keywords


The 'if expression' is *in certain respects* similar to an **If statement**, but with the following differences:

- It is written entirely within a single expression. This is possible because the `if` expression always returns a value.
- There is always a single `then` and a single `else` clause, and each clause contains just a single expression. The `if` expression returns the result of evaluating one of these two expressions, according to whether the condition evaluates to `true` or `false`.
- These `if` expressions may be 'nested' within each other, using brackets around each nested `if` expression where there could be any ambiguity.
  Some more examples:

```
return if c < 1160 then c + 40 else c – 1160
return if isGreen(attempt, target, n) then setChar(attempt, n, "*") else attempt
return if attempt[n] is "*" then attempt else (if isYellow(attempt, target, n)
then setChar(attempt, n, "+") else setChar(attempt, n, "_"))
```

The last example contains a nested `if` expression.

`function` - see Function and Passing a function as a reference
`global` - TODO
`if` - see Elan supports the two forms of 'selection' most widely-used in procedural programming: the If statement and the Switch statement.
(Elan also supports the If expression, which, although often thought of as a Functional programming technique, may be used within procedural programming also – within any expression.)

If statement & Error! Reference source not found.
`immutable` - see Immutable class
`import` – (Not yet implemented)
`in` - see Each loop
`inherits` - see Inheritance
`is` - see Equality testing
`isnt` - see Equality testing
`lambda` - see Passing a function as a reference
If you are passing a reference to a freestanding function as an argument into a HoF (as distinct from defining a `lambda`) then you provide the name of that function, but precede it with the keyword `function`. For example:

```
…
var passes set to allPupils.filter(function passedMathsTest)
…
function passedMathsTest(p as Pupil) as Boolean
  return p.mathsPercent > 35
end function
```

**Notes:**

- When passing in a reference function `passMathsTest`, the name is preceded by `function`, and that no parameters (or brackets) are added to the name as they would have been if you were *evaluating* (calling) the function at that point.

Lambda
`let` - see If expression
The 'if expression' is *in certain respects* similar to an If statement, but with the following differences:

- It is written entirely within a single expression. This is possible because the `if` expression always returns a value.
- There is always a single `then` and a single `else` clause, and each clause contains just a single expression. The `if` expression returns the result of evaluating one of these two expressions, according to whether the condition evaluates to `true` or `false`.
- These `if` expressions may be 'nested' within each other, using brackets around each nested `if` expression where there could be any ambiguity.
  Some more examples:

```
return if c < 1160 then c + 40 else c - 1160
return if isGreen(attempt, target, n) then setChar(attempt, n, "*") else attempt
return if attempt[n] is "*" then attempt else (if isYellow(attempt, target, n)
then setChar(attempt, n, "+") else setChar(attempt, n, "_"))
```

The last example contains a nested `if` expression.

Let statement

`library` - TODO
`main` - see **Main routine**
`mod` - see **Arithmetic operators**
`new` - see **Using a class**
`not` - see **Logical operators**
`of` - see **Types**
`or` - see **Logical operators**
`out` – see **Parameter passing**
`print` - see **Printing to the Console**
`private` - see **Object-oriented programming**
`procedure` - see **Procedure**
`property` - see **Object-oriented programming**
`repeat` - see **Repeat loop**
`return` - see **Function**
`set` - see **Using variables**
`step` - see **For loop**
`switch` - see **Switch statement**
`test` - see Error! Reference source not found.
`this` - see **Object-oriented programming**
`throw` - see **Error/Exception handling**
`to` - see **For loop**
`true` - see **Boolean**
`try` - see **Error/Exception handling**
`typeof` - see **Identifying and comparing types with 'typeof'**
`var` - see **Using variables**
`while` - see **While loop**
`with` - see **Cop**