# About Elan

Elan is a programming language…

* Designed specifically for education –not just learning how to program, but for learning how to program *well*.
* Elan is committed to continuing to serve the needs of education to the best possible degree, and not to be tempted to try to attract professional developers – who have different needs. But that does not imply that Elan is in any way a primitive language
* Is a ‘multi-paradigm’ language, meaning that it supports the ‘procedural’ (PP), ‘object-oriented’ (OOP), and ‘functional’ (FP) programming paradigms. Its support for FP – seen by many computer scientists as set to become the next ‘dominant paradigm’ in programming – is considerably better than most multi-paradigm programming languages used in education today.
* We argue, therefore, that learning to program in Elan, sets you up very well to transition to a wide range of advanced programming languages should you decide to

## Requirements

Elan runs within a browser, and we recommend using Chrome.

We also strongly recommend that once you have navigated to the Elan page, that you switch the browser to full-screen mode e.g by pressing **F11** – in order to maximise both the screen area available to view the code and the input/output windows when running a program.

## Elan code files

Elan code files…

* have a .elan ‘file extension’ e.g. myProgram.elan
* must be created, saved, and edited, from the Elan IDE. If you try to create or edit an Elan code file outside the IDE (for example in Notepad, or another IDE), the file you create will almost certainly not be able to be loaded into the Elan editor, and hence not be able to be run.
* may be *loaded* into the Elan IDE from anywhere on your own machine, or from a file server that you have access to.
* are always *saved* to the Downloads directory, from which they may be copied or moved to another location. (This is a restriction imposed by browsers - for security reasons – not by Elan).
* are plain text files. They may be viewed in e.g. Notepad, checked into a code repository such as GitHub, and the difference between versions of a file during development viewed using a standard ‘diff’ tool

## Accessibility

* Because Elan runs in a browser, you can use the standard browser actions for magnifying the text size – for example by holding down the **Ctrl** button and using the mouse scroll-wheel.
* Elan makes heavy use of ‘traffic light’ colours (red, amber, green) to distinguish the status of various aspects of the code. If you have difficulty distinguishing these colours, teacher may easily create a profile for you that uses alternative colours.
* Most actions can be initiated either via the mouse, or through keyboard short-cuts. It is possible to do everything without a mouse.
* (A longer term aim of the Elan designers is to create a version of Elan that is designed specifically for blind users.)

## Versioning

Unlike in most languages, the Elan language and the IDE (integrated development environment) form a single product, with a single version. There is no possibility of the two becoming ‘out of synch’.

The version number that you are using, is always shown within the ‘header comment’ found at the top of every Elan code file. This comment is generated by the system and may not be edited or deleted. Indeed the header comment must exist in the file in order to load a file.

Elan adopts the discipline of ‘semantic versioning’ (see <https://semver.org/> ). The version has three parts, for example:  
v1.2.35

While Elan will be updated quite frequently, the expectation is that the *first* (‘major’) number will remain at 1 for many years. This is an important advantage. Let’s say that you are running v1.3.14 :

* Any programs that you wrote for earlier versions will still run, without any changes or problems
* You will also be able to load and run any code file that has been written by someone running a version of Elan that has a higher-level *third* number - for example v1.3.15
* If you try to load a code file written by someone who was using a version of Elan with a higher *second* level number – for example v1.4.0 – indicating that the version has *additional* capabilities, then you will be advised to upgrade to that or, preferably, the *latest* available version, *because it will always be safe to do so* without any risk that your existing programs may no longer run.
* A higher *first* level number indicates that it may be necessary to make some (typically small) changes to some of your existing programs in order to run them on the new version. **For this reason, the developers of Elan are committed *not* to introduce a first-level change without extensive consultation with, and support from, teachers that use Elan.** In a **v2.0.0** were to be introduced in future, you will then *automatically* be advised of how to check the compatibility of any existing programs, or the small changes that might be needed, *before* you decide whether to upgrade.

# Viewing and navigating around code

The best way to *start* using Elan is explore an example program.

Use the **Load** button to find and load a .elan file.

[screenshot]

## Viewing code in ‘outline’ view

Use the **+/-** button (above the **Code** window) to collapse the code to outline view. Each ‘multiline’ code region will be reduced to a single line starting with a coloured **+** symbol, for example:

[screenshot]

Clicking the **+/-** button again, will expand the code to normal view.

Note: Pressing the keys **Ctrl-Shift-O** (the ‘O’ stands for ‘outline’) has the same effect as clicking the **+/-** button.

You may also expand or collapse a region by double-clicking it with the mouse, or by selecting it (see below) and then using the keys **Ctrl-o** (i.e. without the **Shift** key).

## Frames

You will have seen already that whether the code is in normal (expanded) view or collapsed, *some* part of the code is always highlighted with a background colour (see Accessibility). If the background colour is pale blue, then the code being highlighted is a ‘frame’.

### Single-line frames

Some frames are just a single line of code, for example:

[screenshot]

### Multi-line frames

And some are ‘multi-line frames’. *Most* multi-line frames start with a ‘keyword’ such as main, function, while (keywords are rendered in dark blue) and finish with the keyword end , immediately followed by the starting keyword. The following screenshots show such multiline frames selected:

[screenshots]

As well as highlighting the first and last lines, a multi-line frame highlights a vertical bar between them to show indicate the extent of other frames that are contained within the highlighted frame.

When not selected, this vertical bar is not highlighted, but serves to ‘indent’ the contained frames.

The few exceptions to the general rule stated above include the then and else clauses found within an if statement, and the case and default clauses within a switch statement. These clauses do not have individual end statements – they are terminated by the start of next clause, or by the end of the *enclosing* frame (i.e. by end if or end switch). So the highlighting can be described as ‘r-shaped’ rather than the ‘c-shape’ for most frames:

[screenshots]

Also the final line of a repeat ‘loop’ contains some additional code on the same line as the end repeat:

[screenshot]

### Wrapped lines

Depending on the size of your screen and the magnification of the text, if any line is longer than the available width it will *automatically* ‘wrap’ onto multiple lines – rather than resort to a horizontal ‘scroll-bar’. For example:

[screenshot showing more than one example]

It is important to understand that a *wrapped* line is still treated as a single line of code – it is not broken into multiple lines. Expand the browser window, or reduce the magnification, and the line breaks on a wrapped line will adjust automatically. Select on one any of those wrapped frames, and the full extent of the code will be highlighted, as in the last example above.

### Selecting frames

You can select on a *frame* by mouse-clicking on the starting keyword of that frame.

Alternatively you can navigate to any frame using the following keys.

**Cursor-up**, **Cursor-down** to select the next frame above or below the current one that *is at the same level of indentation.*

**Cursor-left, Cursor-right** – are of use only in the context of multiline frames. **Cursor-right** takes to *into* the selected multi-line frame to the first frame that it contains – sometimes described as the first ‘child’ frame. Cursor-left selects the enclosing frame for the current selection – sometimes described as the ‘parent’. If the current selection has no indentation then it has no parent. We describe these frames as being ‘global’ frames, or global ‘code constructs’. The only possible global constructs are those beginning: main, constant, function, procedure, class, enum, test, and # (a comment).

**Home, End** – select the first, or the last frame at the same level of indentation (sometimes stated as ‘at peer level’) the currently-selected frame. If the currently-selected frame is a global construct then **Home** / **End** will take you to the first / last frame in the file.

## Fields

A whole frame may be added into code. moved, cut, pasted, or deleted [TODO: add links] but you may not edit or remove any of the keywords that *define* the frame. In other words the *integrity* of a frame is enforced. This is one of the key features of Elan that is designed to prevent the possibility of the ‘syntax errors’ that are a feature of almost every other text-based language. (If you have previously used a ‘block-based’ programming language such as Scratch, or Snap! Then you will be familiar with the idea of enforced code integrity.)

Most frames (the exception is main) define, in addition to the fixed keywords or punctuation, one or more ‘fields’. Fields are designed to contain code that is entered and/or edited by the programmer. The text contained in a field will normally be rendered as black text and/or dark red (for strings) and dark blue if the field contains a keyword. However, when selected, the field will be shown with a green background (see Accessibility) assuming it is in a valid state – as all fields should be in any example program you are exploring.

[screenshot]

(Shortly we will look at editing code, and entering new code).

To select a field, you can either:

* Mouse-click within the text of the field
* From the frame that the field is a part of, press **Tab**to select its first field, then **Tab** again for the next field, and so on. If you press **Tab** from the last field in the frame then you will get back to the selected *frame*.

# Editing code

## Editing at frame level

Having selected a frame, you can:

* *Delete* the whole frame by pressing **Ctrl-Delete** or (for keyboards that do not have a **Delete** key) **Ctrl-d**
* *Move* the whole frame up or down by pressing **Ctrl-↑**  or **Ctrl-↓**. You can move the frame up or down *only at the same level of indentation.* With each keypress the selected frame will skip over the previous/next frame at the same indentation, whether it is a single-line or multi-line frame. If you reach the top or bottom of an enclosing frame, you will not be able to move further – you will need to use cut/paste (explained below) instead.  
    
  **Note** A few frames are held in a fixed position - as the first child of their parent frame ( constructor and then) or the last child (see return and default). You will not be able to move these fixed frames, nor will you be able to move other frames past them.
* *Cut* the whole frame to the (unseen) ‘scratchpad’ by pressing **Ctrl-x**
* *Insert* new code below a selected frame by pressing **Enter**, or above the selected frame by pressing **Shift-Enter**. These actions will insert a new code ‘selector’ below/above the frame, and move the focus to that selector. (If there was already a new code selector in that location, then these actions will simply move the focus to that selector).
* *Paste* a frame from the scratchpad by pressing **Ctrl-v** *when the focus is on a new code selector*. You may only paste code onto a new code selector. Additionally, the paste action will work only if there is a frame in the scratchpad, and *only if the type of frame in the scratchpad is compatible with the location you are attempting to paste into*. In other words you cannot paste in code that you could not create in the same location.

# Got to here

Long lines will ‘wrap’ automatically onto the next line, or even over multiple lines.

[screenshot]

# Types

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Literal value example** | **Type name** | **Notes** |
| **Integer** | 42 | Int | Size? |
| **Floating Point** | 1.618  3.0 – must always include a decimal point | Float | An Int may be passed as an argument to a Float parameter |
| **Boolean** | true or false | Boolean |  |
| **String** | "Hello" | String | Newline may be embedded as \n |
| **Date** | Not yet implemented | | |
| **List** | ["apple", "orange", "pear"]  [1.0, 2.5, 3.0, 0.0, 2.0]  [[3,7,1], [5,5,7], [0,1,0]] | List<of String>  List<of Float>  List<of List<of Int>> | Members of a list must all be of the same type. To create an *empty* list:  new List<of String>() |
| **Dictionary** | ['a':121, 'b':23, 'c':35] | Dictionary<of Char, Int> |  |

## 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: 253 – 1 which is just over 9 x 1015
* Minimum value: -(253 – 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 x 10308
* Minimum value: approx. 5 x 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 ArrayList, *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 double-quote marks *within* a string. This is likely to be made possible in a future release.

## Date and Time

Dae and Time are not currently yet implement as standard types in Elan. They are likely to be introduced in a future release.

## Array-list

An ‘array-list’ is the simplest way to represent a collection of data items *of the same type*. It is called an ‘array-list’ because it offers the functionality both of a traditional array, including:

* access elements by index
* create an empty structure of a defined size

and of a traditional list (sometimes referred to as a ‘linked list’), including:

* be able to append to the list, extending it dynamically – starting from an empty list if desired
* be able to find elements within the list
* be able to insert an item between existing members of the list

### Type name

The type name for the array-list must specify the type of the elements it contained, in angle brackets:

ArrayList<of String>

ArrayList<of Boolean>

ArrayList<of ArrayList<of Int>>

### Defining a literal array-list

A literal array-list is ‘delimited’ by square brackets, and the 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 lists’:

var coordinates set to [[3.4, 0.1, 7.8],, 15.3] [1, 0, 1.5], [10, -1.5, 25]]

Or of variables of the same type, or a combination of literal values and variables.

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:

### Default value

The default value of any type of array-list is empty. This may be created by:

## var players set to new List<of String>()

### Constraints

* All the members of the list must be of the type specified (either explicitly in the name of the Type, or implicitly in the literal values with which the list was initialised).
* An array-list may be passed as an argument into a procedure, but may not be passed as an argument into a function because the latter may accept only *immutable* types, hence …

## Immutable-list

An ImmutableList has similar capabilities to an ArrayList but – just like a string – is *immutable*. You can still insert, delete, or change elements in an Immutable--Lst, 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 name for the immutable-list must specify the type of the elements, for example:

ImmutableList<of String>

ImmutableList <of Boolean>

ImmutableList <of ImmutableList<of Int>>

### Defining a literal immutable-list

Like an array-list but delimited by curly-braces rather than by square brackets, for example:

var fruit set to {"apple", "orange", "pair”}

### Constraints

* Like an ArrayList the members of an ImmutableList must all be of the same type.
* Unlike an ArrayList, an ImmutableList may be passed as an argument into a function (and also to a procedure).

## Enum

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

end enum

* The name must begin with a capital letter, which may be followed by any combination of lower-or-upper-case letters, the underscore ­\_ symbol, or decimal digits (though it is not common practice to use decimal digits in enum names).
* The values that an instance of that type may take, are separated by commas. Each value must take the same form as a variable name i.e. start with a lower-case letter followed by any combination of lower-or-upper-case letters, the underscore ­\_ symbol, or decimal digits.

### 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.
* *Internally*, enum values are held as integers, with the first named value represented as 0. It is possible to use an enum value *as though it were an integer* – for example as an index into an array-list, or to compare two values of the same Enum type using the standard comparison operators (is, is not, >, etc).

## 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 separate two new variables:

var (x, y) set to point1

or into existing variables of the correct type:

var a set to 0

var be set to 0

set (a, b) to point1

### Constraints

* Tuples are currently limited to having two or three members, which may be of the same or different types. (There is no point in defining a tuple with only one members, and so this is disallowed.)
* 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 immutable-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

## Class

A class is user-defined type – offering far richer capability than an Enum. (Refer to the section on object-oriented programming for more details.)

### Type name

The type name for a class is just the name given to that class when it was defined, which, like any other type must always begin with a capital letter.

## 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 functions, and is a key idea in the functional programming paradigm. In order 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. For example:

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

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 the a function definition that started:

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

## Default values

Every type in Elan – whether pre-defined (such as Int or String) or user-defined (see class and enum) – has a default value

# Variables

Variables are defined using the var

Variable names must commence with a lowercase letter, followed by any combination of lower-case or upper-case letters, numeric digits, and the underscore.

Variable names are type-sensitive.

Variables may only be defined within the main, a function, or a procedure – and are thereby scoped to that construct. (There is no such thing as a ‘global variable’ in Elan - although there are global constants).

## Variable definition

### Var

## Re-assigning a variable

### set

## Scope of variables

# Expressions

## Operators

### Arithmetic operators

Elan recognises five ‘binary’ arithmetic operators (‘binary’ here means simply that the operators require *two* ‘operands’ – it is nothing to do with the binary notation of numbers). Each may be used with literal values or variables, a y of which may be of type Int or Float.

Addition: var x set to a + b  
Subtraction: var x set to a - b  
Multiplication: var x set to a\*b  
Division: var x set to a/b (the type of result will always be a Float, irrespective of whether the operands are of type Int or Float)

div and mod - which are used to undertake integer-division (yielding an Int result) and the ‘modulus’ or remainder from an integer-division – are invoked as *functions* not operators. See xxxx.

Elan also recognises one *unary* operator (working on just one operand) – the minus symbol – to negate a number for example:

var x set to -b

Arithmetic operators are evaluated according to rules of ‘precedence’ as captured in the popular mnemonics BIDMAS or BODMAS widely taught in Mathematics, so, for example:

3 + 4\*7 – 2 will evaluate to 29

(3 + 4)\*(7 – 2) will evaluate to 35

3^2 + 4^2 will evaluate to 25

Note that the Elan editor always adds spaces around the + and – operators, but no

### Logical operators

is, not, and, or, xor, >, <, >=, <=

### Function call

### New Instance

#### With clause

# Comments

A comment starts with # (hash symbol) :



This is typically the last option offered by the new code selector:



A comment must be defined on its own line – a comment cannot be placed at the end of a line of code, nor within it.

For multi-line comments, each line must start with the # symbol.

There is a system-generated comment at the top of every Elan code file, showing the Elan version number and (in some cases) additional information:



This comment cannot be edited or removed, and an Elan file cannot be loaded into the editor without this line (which also contains other, hidden, technical information) being present.

## Global constructs

What this term means and what they are

Illustrated by what’s visible in the global selector

## Main

Any program that you wish to *run* (as distinct from being a library intended for use within another program) must have a main.

The main can be defined anywhere within the file, but the convention is to define it at the top of the file.

There may only be one main within a file, so if a main already exists, the main option no longer shows up as an option in the new code selector.

A main may contain any of the statements that may be added into a procedure.

## Function

### Defining a function

#### Return statement

### Rules & patterns

Must return a value of the type specified in the signature

May have only a single return statement, which must be the last statement in the function – it is auto-created by the Function frame and may not be moved from the last position, nor deleted.

When the function is created the return statement takes the form: return result.

The result keyword references a variable, scoped to the function wherein it is used, defined (behind the scenes) to the type that the function must return and initialised to the default value for that type.

The result may be assigned to a new value within the function. [example]

It is not necessary to make any use of the result keyword, it is just a convenience. You may write, for example:

[function defining and writing own result]

And many functions may simply return the result of evaluating an expression [example]

Functions may not cause any side effects – such as creating output, or modifying a reference passed into the function as a parameter – and they may not depend upon any external information such as a on an input, on today’s date, or a random number generated inside the function – everything the function needs must be passed in as a parameter.

For this reason, the input, print, and external statements are not offered as options by the new code selector within a function body.

For the same reason

Example of invalid code:

* Attempting to set a parameter
* Attempting to pass in an array
* Attempting to call a procedure method on a class

### Using a function

May only be called as part of an expression e.g.

In a var, set, return

In the condition for a selection or iteration statement

As an argument or index defined in-line

## Procedure

### Example

### Rules

Can include input, output, or external dependencies

If the type passed in is a *reference type* and *mutable* (e.g. an Array or an of a user-defined class) then any change made to that parameter inside the procedure can be observed outside the procedure

Example – in-line sort

### Calling a procedure

#### call statement

## Constant

### Examples

Numeric values

Literal lists or dictionaries

### Rules

Must define a literal value or data structure – cannot make use of functions or other constants

Always declared at global level.

### Using a constant

Example

Option to use the global qualifier to disambiguate from a variable.

## Class

See also Class Members

### Class Members

See also Class

Notes:

* Properties may be read or written to *within the class, but may only be read by code outside the class. If you want to be able to change a property from outside you need to write a procedure method to do this. A common* convention *in OOP is that a method that simply modifies (sets) the property name Foo, is named setFoo().*
* private properties and methods are not currently supported. They will be in v1.0
* You can set properties, or use function methods within the constructor, but you may not call any procedure, whether defined within the class or externally. This is because procedures may have side effects and it is a bad idea to cause side effects in a constructor.

### Inheritance

#### Notes

Elan implements ‘invariance’. For example:

* a Foo can be passed into a parameter expecting AbstractFoo,
* a List<of Foo> can be passed into a parameter expecting Iter<of Foo>, but…
* a List<of Foo> *cannot* be passed into a parameter expecting List<AbstractFoo>

## Enum

## Test

Like most modern languages, Elan supports the writing and execution of automated ‘unit tests’. However, tests in Elan differ from those in most languages in several important ways:

* Elan tests are ‘first class’ – they are *part of the language*, not an add-on a library of functions and types.
* Elan tests do not need to be written within a separate class, or file, or project – they can safely be mixed in with your application code. Some Elan programmers prefer to write the tests for a function immediately adjacent to the function definition, others prefer to define all the tests in one place – at the end of the file, say.
* If tests have been written they execute automatically whenever the Elan code successfully compiles.
* When run the result from each assert statement will be shown alongside it
* If a test contains multiple assert statements the test runner does not stop at the first failure – it will attempt to execute every assert statement. (An assert will only show the status not run if the code has thrown an exception – not because a prior assert has failed.)

#### Defining a test

A close-up of a black and red text

Description automatically generated

* Each assert statement specifies that a *generated value* is the *expected result*. In the first example above, the generated value is setChar("ABCDE", 0, "\_") and the expected result is "\_BCDE".
* A test may also include var and/or let statements to hold the generated value and/or the expected result. But a test may not include any other type of statement.

#### Running tests

Tests execute automatically whenever the Elan code successfully compiles. A typical result is shown here:

A group of red and white text

Description automatically generated

If all asserts in all tests have passed, then the test status shown the **Status** panel will indicate pass (green – unless configured to be a different colour):

A blue background with green rectangles and white text

Description automatically generated

but if *any* assert in any test has failed then the overall test status will show as fail (red by default):

A screenshot of a computer

Description automatically generated

#### Notes

* Tests may be written anywhere within the program.
* Giving a name to the test is *optional*. If you are writing multiple tests within a program it is a good idea to give a test a name that describes its scope and/or intention – so that you can navigate to a particular test easily in outline view. Test names are not, however, required to be unique, since they perform no role at run-time. A test may, for example, have the same name as the function it is testing. There is no danger that the two will be confused, because a function name may be referenced in other code (within an expression), but a test may never be referenced from other code.

### Constraints

* Currently, unit tests can test functions, but not *procedures* (since the latter may contain input/output). It is hoped that a future version of Elan will offer support for the automated testing of procedures and even the main routine, for programs that use only console input/output.

# Statements

## Selection (conditional statements)

### switch

#### case

#### default

### if

#### else

#### else if

## Iteration (loops)

### for

### each

### repeat

### while

## Handling errors

### try

#### catch

### throw

## Input / output

### print

### input

### ‘Impure’ library functions

# Debugging

Elan v 1.0 will have a powerful debugging capability, but it does not exist in the Beta version.

Meantime, we recommend that you use the powerful Testing capability to help with debugging functions and procedures. And - we hate this idea, but just as a stop-gap solution – for the main routine and procedures, you can add-in temporary print statements.

# Input/Output and other system capabilities

## Random numbers

Note that both of the methods shown below have a dependency upon the system. Therefore they may be used

### random

returns a Float in the range 0 <= x < 1

Example use:

### randomInt

returns an integer in the range spaced. Note that both limits are *inclusive.*

*Example use:*

## Timing

clock

pause

## Console pane

### **print**

### **input**

### **getKeypress**

**Note: The console display is automatically updated with whatever has been printed, when the program ends, or whenever the system is waiting for an input. If you want it to update while the program is running – for example if using a loop to print out a long list of values – then insert the following line of code:**

**pause(1)**

**Whenever *any* pause method is encountered, program execution will be paused and the console display will then be updated. pause(1) just specifies the minimum delay – 1 millisecond.**

## Graphics pane

## File reading/writing

Not currently implemented. Elan v1.0 will have support for reading and writing, both text and binary files.