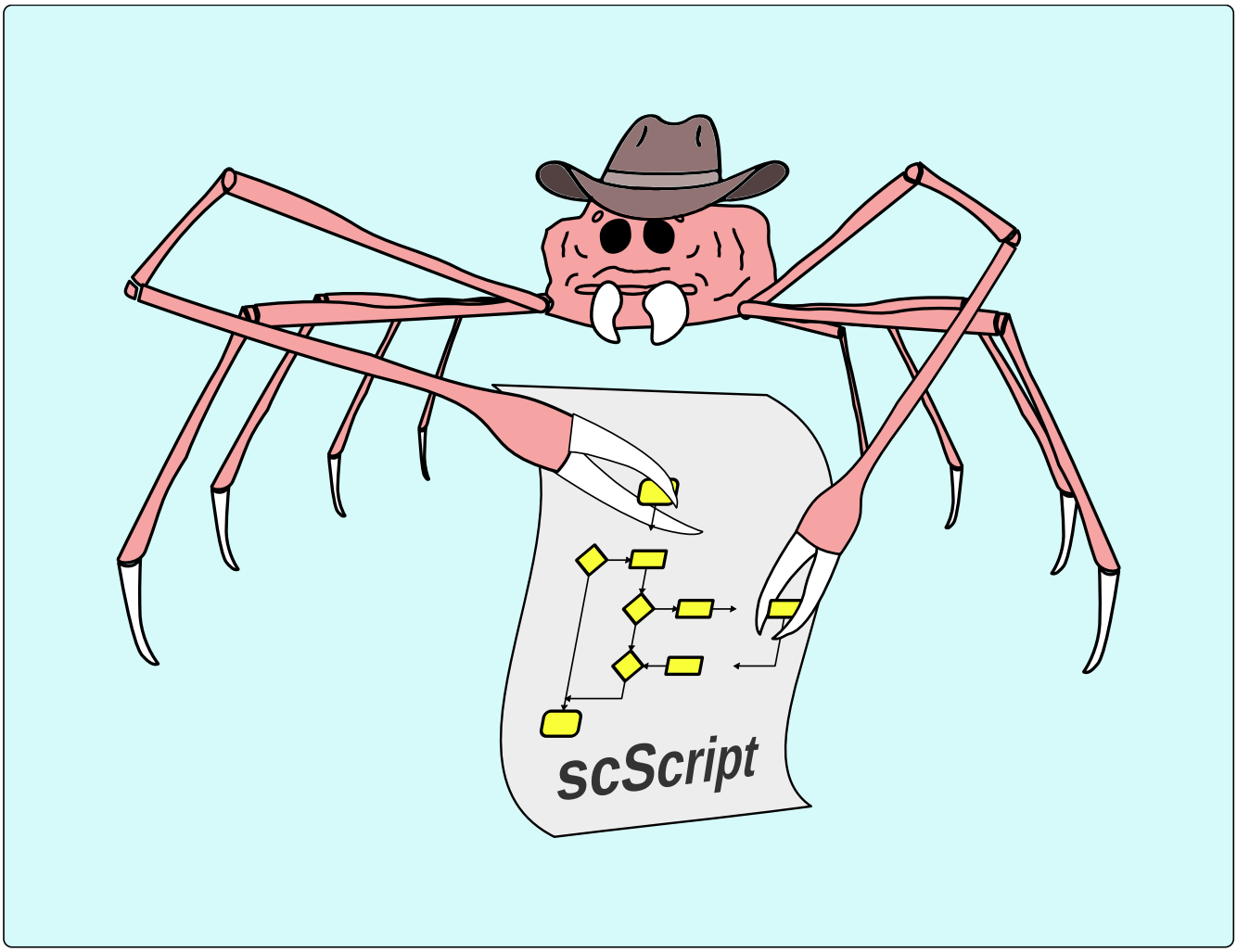


scScript™ Guide

(Includes scEmacs)

Overview

scScript is a powerful but simple to use C-like scripting language with *very little* of the weirdness and strange syntax common to scripting systems. scScript compiles to efficient bytecode that runs in a fast virtual machine. It is integrated with scEmacs, a small display editor for I/O, program control, source editing, pattern matching, and data display. Everything is written in C, runs on 64-bit Linux, is easy to customize or extend, and can be embedded in larger applications.

The scScript language:

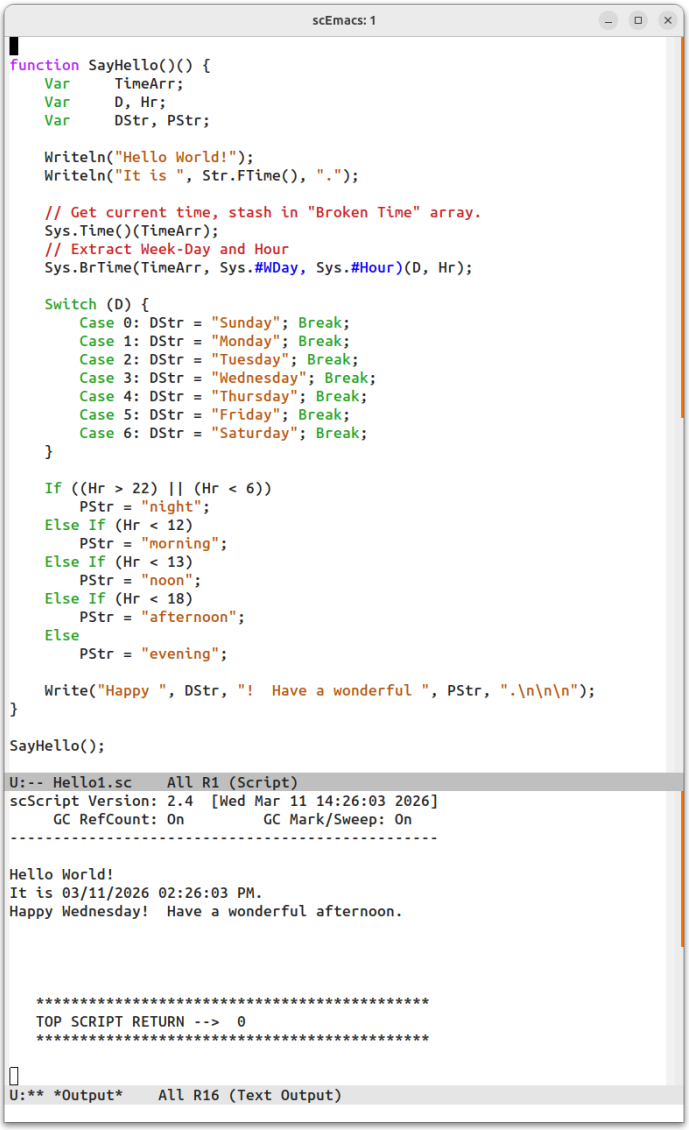
- Has familiar C-like syntax and operators.
- Provides a simple *pointer-less* scripting environment with:
  - 64-bit integer and double floats
  - Immutable UTF-8 text strings
  - Arrays that grow automatically
  - Dictionaries (flexible assoc lists)
  - External structures (3 flavors)
  - Nested function declarations and first-class closures
  - Variadic Pass-by-value and Pass-by-reference function args
  - Subordinate scripts
- Has *stackful* asymmetrical (full) coroutines and non-local exits.
- Compiles to optimized bytecode running in a small VM.
- Can display line-by-line source AST and resulting bytecodes.
- Optimizes *tail call* recursion (eliminates stack frame).
- Uses Mark/Sweep GC as well as Ref Counting.
- Provides a powerful (text) pattern definition and matching system.
- Readily interfaces to C routines for package libraries and features.
- Employs slab-based low-level memory management.
- Only requires `libx11` and `libxft` for scEmacs, plus `libffi` for *printf* interface.

scEmacs is an integrated source editor with:

- Multiple buffers in multiple panes on multiple windows,
- Color source code display,
- UTF-8 text, *Undo* stack, *Kill Ring*, and *Mark* list,
- Incremental search, query replace, and auto command completion,
- Mouse-based selection, scrolling, and even Unix style XSel copy/paste,
- Pop-up window/menu facility for command enumeration and selection,
- Specialized commands to drive scScript.

Hello World

`Hello1.sc` is a sample script, slightly more than the obligatory “Hello World!”. The image shows it run from scEmacs using `M-x script-do`. The upper pane displays sources and the bottom is `*Output*` when the script is run.



Functions in scScript are *defined* with *two* param lists, an *in* (Pass-by-value) list and an *out* (Pass-by-reference) one. But a function can be *called* with just the *in* list. `SayHello` defines a simple function with no *in* or *out* parameters, but five local vars.

scScript, like most scripting languages, does not specify *types* for its variables. Any variable can hold anything and the content itself keeps track of *what* it is. So variables, whether local or global, are simply declared as `Var`.

`Sys` is one of the main pre-defined packages in scScript. As of this writing, the list of packages includes:

- `Sc`: Language features
- `Sys`: System calls
- `Show`: Display routines
- `Str`: String processing
- `Ed`: scEmacs calls
- `Pat`: Text matching
- `Log`: Recording operation logs
- `Math`: Math routines

Each package defines routines, such as `Write` and constants such as `#Day` or `#Hour`. These are accessed via the *dot* operator, which scScript also uses to specify keys for dicts, just as C specifies fields in a record.

`sc.Write` is the main output function, it takes a list of *in* args and writes them to the `*Output*` buffer (in scEmacs) one after the other. But routines in the `sc` package (and *only* that package) can be accessed *directly* without the need for package and dot prefix, so `sc.Write` is usually just coded as `Write`. `Writeln` is similar (remember Pascal?), but adds a `newline` character at the end. (There is also a `Sys.Print` that works like `printf` in C and takes all the same `%` directives!)

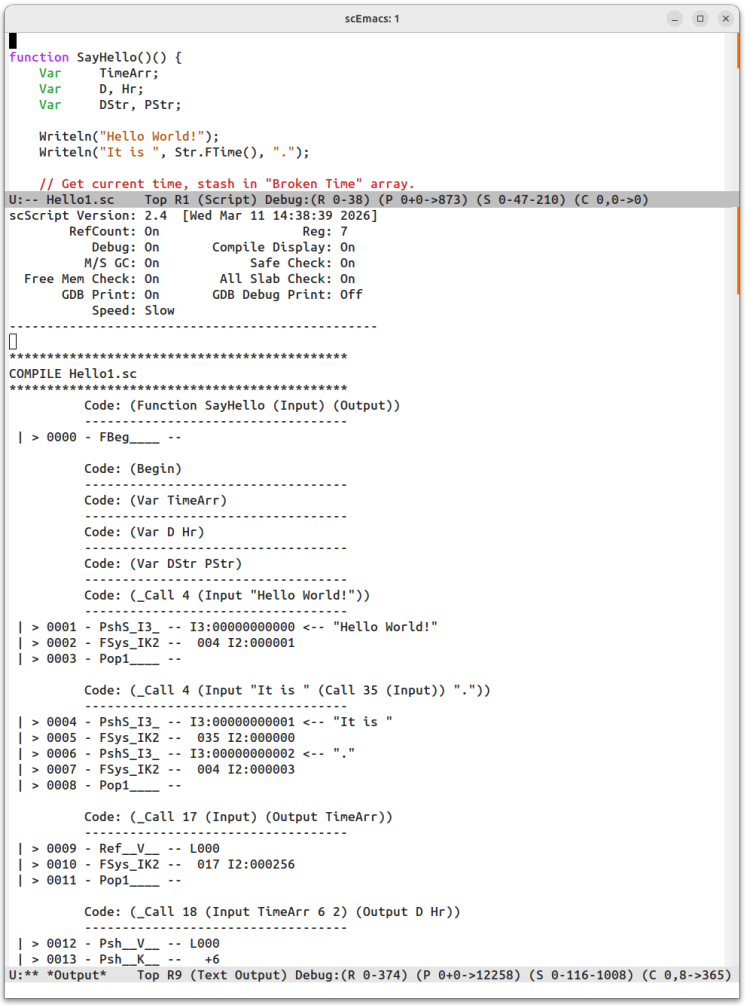
`FTime` in the `Str` package takes the current (local) system time and makes a date-and-time string out of it (uses Linux `strftime`). This then becomes an argument to `Writeln` which draws the second line.

`Sys.Time` is a routine that normally just returns the current time in secs since the Unix epoch began, exactly as `time` does in Linux. But if given an *out* arg (`TimeArr` in this case), `Sys.Time` will place a

broken time record in it. This value (the broken time) is handed over to `Sys.BrTime` along with two pre-defined flags. `Sys.BrTime` expects to have an arg in its *out* list for each flag in its *in* list. It will extract the value indicated by the flag and place it in the corresponding *out* var. So `D` will get the `#WDay` (week day) value and `Hr` will get the `#Hour` value from `TimeArr`.

The *switch* statement works just like C, and stashes the selected `DStr` (Day Str). An i*f* ladder stashes the proper `PStr` (Period Str). Both vars make it to the final `Write` and print out the last line of greeting.

In this example, `Str.FTime` and `Sys.Time` both call the OS for their very own system time, but there can be a slight discrepancy! So `Str.FTime` could get the time just before midnight on Sunday while `Sys.Time`, called a few microns later, may claim it is Monday. This can be remedied by calling `Sys.Time` first, stashing the value it returns, and passing it to `Str.FTime` as an *in* arg.



While the sample code above capitalized keywords (in green and purple), scScript is entirely case *insensitive* for language keywords and *pre-defined* system names. However, just like C, all *user-defined* function, variable, and constant names *are* case sensitive. This allows scScript keywords to adopt whatever case convention is preferred. (Finally an entire program in CamelCase, a dream!!)

The screenshot on the right comes from a `DEBUG` version of scScript, compiled using the `DEBUG` option in its `makefile`. The image shows an expanded and scrolled `*Output*` pane. Both the *mode* line and `*Output*` header clearly indicate the `Debug` state and provide additional developer information.

When in this mode, scScript can output *parsed* script sources (the AST in Lisp-like format) and their corresponding bytecodes on a line-by-line basis, showing exactly how it read and compiled the sources.

Here, the first *executable* line is `Writeln("Hello World!")`. This is parsed as a `Call` to the 4th system function with one `Input` string, and generates three byteops. The first will *Push* the input string on the runtime stack, the second will *Sys* call the function (`Writeln`), and finally the third instruction will *Pop* the (unused) result returned by `Writeln` off the stack.

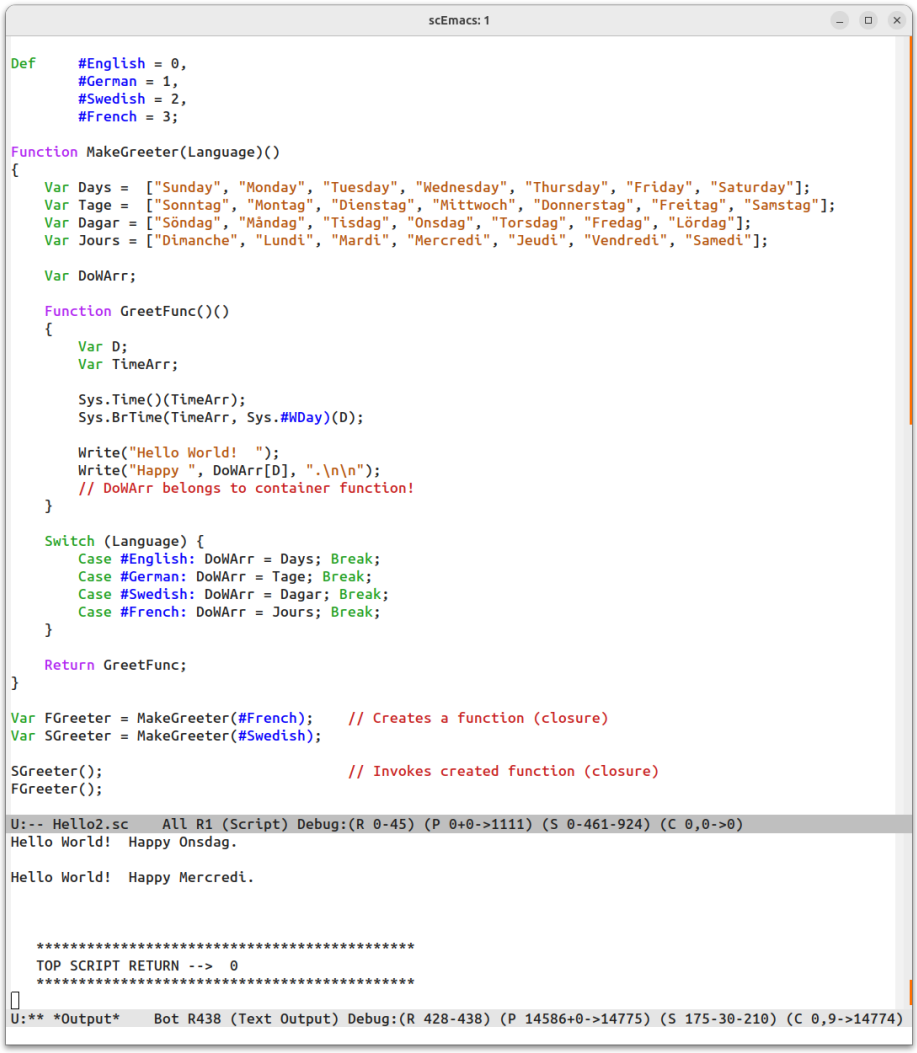
> The leading *underscore* in `(_Call 4 (Input...))` indicates its (returned) result will be ignored, hence the `Pop1` op on line `0003`, as all called functions normally place a return value on the run stack.

Hello2.sc is a functionally more complex example, written to illustrate *nested* functions and *closures* in scScript. The main body, bottom four lines of source code, makes two calls to `MakeGreeter`, a local function that itself creates (and returns) a `Greeter` function. The resulting functions are stashed as part of global variable (`FGreeter` and `SGreeter`) declarations, and subsequently called with no args.

The script starts with some constant declarations. `Def` is used just like `Var`, but is only a compiler directive to define a *name* (preferably with a `#` prefix) for a number or string literal. The scope of `Def` is determined by *where* it is declared, global in this case and accessible by all subsequent lines.

`MakeGreeter` is a local function that first defines arrays of strings, in 4 languages, then chooses one for its *`DoWArr`* (Day-of-week-array) using a *Switch*. So far, the operation is trivial.

But `MakeGreeter` also declares a *nested* `GreetFunc` that uses `DoWArr` from its (superior) container function. When called, `MakeGreeter` returns its nested `GreetFunc` after choosing a language. It returns the function *itself*, as defined, but does *not* call it. This is just like C returning a function ptr, but there is more to it!



Storing or returning such a local function actually creates a *closure*, the combination of a function and all *inherited* local variables it *needs* to operate. Each called instance of `MakeGreeter` sets its own value for `DoWArr`, determined by the `Language` *`in`* param (`#French` or `#Swedish` in this example). But unlike persistent global variables, `DoWArr` will be gone when `MakeGreeter` returns. So the `GreetFunc` inside `MakeGreeter` is forced to *close over* this inherited variable; otherwise, it will be hopelessly lost after `MakeGreeter` returns and terminates. The *closed-over* value of `DoWArr` lives on and persists in the closure for the particular `GreetFunc` instance! (The closure has a pointer to the *storage location* of each variable it closes over.)

Each call to `MakeGreeter` creates a new run for it (scScript uses a *RunRecord* instead of a stack frame) and a whole new set of local variables. Similarly, the `GreetFunc` that it returns will close over a correspondingly different local `DoWArr`. Since the two closures, for `#Swedish` and `#French` are stored in two globals, it is an easy matter to compare them with a system call:

```
Show.Vars(Show.#Max)(SGreeter, FGreeter);
```

This will generate a detailed listing of the two: (placed side-by-side below for comparison)

```
Var (Ref:1) [Entry 000048 (16 Bytes) on Mem:1 (VEnt) Slab:1]
    --> Clos (Ref:1) [Entry 008328 (40 Bytes) on Mem:5 (Chnk) Slab:1]
        --> Closed Vars:1
            001 Var (Ref:1) [Entry 000528 (16 Bytes) on Mem:1 (VEnt) Slab:1]

        --> Code (Ref:3) [Entry 008648 (112 Bytes) on Mem:5 (Chnk) Slab:1]
            **********************************
            **********************************
            Code:0008           Len:  56 Bytes
            Reg:0    P:00+00    L:002    C:001
            **********************************
            * 000000 - FBeg____ -- XXX
            * 000002 - Ref__V__ -- 002
            * 000004 - FSys_II2 -- 014     256
            * 000008 - PopN_K__ --   1
            * 000010 - Psh__V__ -- 002
            * 000012 - Psh__K__ --   6
            * 000014 - Ref__V__ -- 001
            * 000016 - FSys_II2 -- 015     258
            * 000020 - PopN_K__ --   1 XXX XXX
            * 000024 - PshS_I3_ --          28 <-- "Hello World!  "
            * 000028 - FSys_II2 -- 002       1
            * 000032 - PopN_K__ --   1 XXX XXX
            * 000036 - PshS_I3_ --          29 <-- "Happy "
            * 000040 - IGet_VVV -- 000 003 001
            * 000044 - PshS_I3_ --          30 <-- ".\n\n"
            * 000048 - FSys_II2 -- 002       3
            * 000052 - PopN_K__ --   1
            * 000054 - FEnd____ -- XXX
            **********************************
            **********************************
```

```
Var (Ref:1) [Entry 000672 (16 Bytes) on Mem:1 (VEnt) Slab:1]
    --> Clos (Ref:1) [Entry 008296 (32 Bytes) on Mem:5 (Chnk) Slab:1]
        --> Closed Vars:1
            001 Var (Ref:1) [Entry 000496 (16 Bytes) on Mem:1 (VEnt) Slab:1]

            --> Code (Ref:3) [Entry 008648 (112 Bytes) on Mem:5 (Chnk) Slab:1]
            **********************************
            **********************************
            Code:0008           Len:  56 Bytes
            Reg:0    P:00+00    L:002    C:001
            **********************************
            * 000000 - FBeg____ -- XXX
            * 000002 - Ref__V__ -- 002
            * 000004 - FSys_II2 -- 014     256
            * 000008 - PopN_K__ --   1
            * 000010 - Psh__V__ -- 002
            * 000012 - Psh__K__ --   6
            * 000014 - Ref__V__ -- 001
            * 000016 - FSys_II2 -- 015     258
            * 000020 - PopN_K__ --   1 XXX XXX
            * 000024 - PshS_I3_ --          28 <-- "Hello World!  "
            * 000028 - FSys_II2 -- 002       1
            * 000032 - PopN_K__ --   1 XXX XXX
            * 000036 - PshS_I3_ --          29 <-- "Happy "
            * 000040 - IGet_VVV -- 000 003 001
            * 000044 - PshS_I3_ --          30 <-- ".\n\n"
            * 000048 - FSys_II2 -- 002       3
            * 000052 - PopN_K__ --   1
            * 000054 - FEnd____ -- XXX
            **********************************
            **********************************
```

Each global var has a 16 byte (VEnt) *storage location* in memory that contains a different closure. The first closure (Clos) is 8 bytes bigger, but this is common as memory is constantly re-used and sometimes a slightly larger block is recycled without enough excess to warrant carving off another. While each closures has its own single *Closed Var*, the code block referenced by both is the very same. In fact, the `(Ref:3)` notation for the `Code:2` indicates a third pointer, and that comes from `MakeGreeter` itself which also maintains a pointer to its own code block.

> Note: different images in this document have been generated from different development versions of scScript, there will be some inconsequential discrepancies. For examples, internal system functions may have different numbers, bytecode variations or their args may be different, and display formats may evolve with advancing versions. (The latest version also shows *flags* for vars that have them!)

Operators

Except for pointer manipulation, types, and records, scScript has all the operators and control structures expected by C programmers, plus a few script-specific ones (highlighted in green).

Arithmetic and logic			
A + B	Add	A ^ B	Bit xor
A - B	Subtract	~ A	Bit invert
A * B	Multiply	A && B	Logical and
A / B	Divide	A || B	Logical or
A % B	Modulo (remainder)	! A	Logical Negate
A & B	Bit and	A << B	Shift left
A | B	Bit or	A >> B	Shift right

Assignment			
A = B	Simple assignment	A &= B	A = A & B
A += B	A = A + B	A |= B	A = A | B
A -= B	A = A - B	A ^= B	A = A ^ B
A *= B	A = A * B	A <<= B	A = A << B
A /= B	A = A / B	A >>= B	A = A >> B
A %= B	A = A % B	A =& B	A *becomes* B (Alias!!)

Comparison			
A == B	Equal	A > B	Greater than
A != B	Not equal	A >= B	Greater than or equal
A < B	Less than	A === B	Identical (same storage)
A <= B	Less than or equal	A !== B	Not identical

Accessors, Constructors, and Definition	
Pkg.Func(...)(...)	Call function in pre-defined Pkg. (Can also call func stashed in Dict!)
MyDict.Key	Access dict entry where "Key" must be a string. (E.g. to use "Friday" as a key, use *MyDict.Friday*)
MyDict[Field]	Access dict entry where Field is interpreted. *MyDict[14]*, *MyDict[MyVar]*, or *MyDict["Friday"]* (the last is same as *MyDict.Friday*)
MyArr[Index]	Access array entry. Index is interpreted and must have integer value.
Array(N)	Constructs new empty array, uses default minimum if N is 0.
[A, B, C, ...]	Array constructor where A, B, and C values are interpreted. Arr[0] will get the value of A, Arr[1] will get B...
Dict(N)	Constructs new empty dict, sized for N entries. Uses default minimum size if N is 0.
[K1:A, K2:B, ...]	Dict constructor where keys and values are interpreted.

@A	Unpacks A into *in* or *out* list of a function call. (Advanced operation!) All functions accept a variable number of args! Sys.GetArgs extracts full *in*/*out* arg lists.
Var	Declares variables, which can also be initialized in place.
Def	Defines new names (Starting with '#') for Int, Flt, or String values.
Function	Declare/Define functions. (Has 2 syntactic forms! *****)

Control Flow	
If and If/Else	C style conditionals and nested conditionals are supported.
While	C style While loop, supports *Break* and *Continue*.
Do While	C style Do-While loop, supports *Break* and *Continue*.
For	C style For loop, supports *Break* and *Continue*.
MapI MapK	MapI will iterate over all cells of an array or dict. MapK will iterate over all elts (non-empty cells) of an array or dict. Both support *Break* and *Continue*.
Goto	Goto is supported with standard limitations in scope and target.
Switch	C style Switch selector, supports *Default* case and *Break*.
Return Return Val Return Val OutOf RC	Works with/without value, but can also Return *OutOf* RC (RunContext). Given RC of a caller, can return (*non-local*) directly from *its* caller (N levels up)!
Submit	Any function can *Submit()* to become a persistent coroutine. *Submit* returns an *RC* to caller.
Yield Yield Val Yield Val OutOf RC	Coroutines can *Yield* to return a value to its caller. Also works with *OutOf* for *non-local* Yield.
Resume	Caller can *Resume(RC)* a coroutine after it does *Yield* or *Submit()*.
Release	Caller can *Release(RC)* to terminate a coroutine.
Debug	Call in Script to jump into debugger, uses "raise(SIGINT)" so can continue!

*****

```
Function MyFunc(InArgs...)(OutArgs...) {...}
```

is the preferred form of function declaration. In the case of sub-functions (nested declaration) the scope is *hoisted* to the very top container function. Therefore, the container and any other sub-functions can simply call it.

```
Var OtherFunc = Function(InArgs...)(OutArgs...) {...};
```

is the second form. Here it first declares a variable, then stores a nameless (lambda) function in it; but the var may be declared separately, and assigned later. If declared within a function, this form creates a normal local variable with a *standard* scope. It will *not* be hoisted to the top of its container function and may not be visible to some sub-functions. (`OtherFunc` will be a global if declared at the top level!)

```
Function MyFunc;
```

no *in*/*out* list) can be used to *pre-declare* `MyFunc`, so code calling it (perhaps for mutual recursion) can be written before the function itself is properly defined.

Variables

Variables can be declared in the main body (globals), in a function (locals), or in any {} sub-block (also local). A function definition will usually include named *in* and *out* parameters that will act as pre-initialized local variables when the function is called. The scope of a declaration extends to the end of its block, or the end of the program in the case of globals. Variable names must start with a letter, potentially followed by more letters and digits. scScript is case sensitive (like C) but only for *user-defined* names.

A variable declaration has the form:

```
Var MyVar;
```

or perhaps

```
Var MyVar1, MyVar2, MyVar3;
```

These are all automatically initialized to `Int 0`.

Just like C, these may also be initialized/assigned *within* the declaration:

```
Var MyVar1 = 12, MyVar2 = "This", MyVar3 = MyGlobVar - ThatFunc(MyArr);
```

All variables are declared the same way, and are free to assume any value type, including integer, floating point, string, closure, array, dict, etc. Empty arrays and dicts are created with their respective constructors:

```
MyArr = Array();   or   MyArr = Array(12);
```

Also

```
MyDict = Dict();   or   MyDict = Dict(12);
```

This assignment can be done when the variable is created, or later. In scScript, a missing arg in a call is generally interpreted as `Int 0`, so `Array()` and `Array(0)` are the same and will create an array with the (default) minimum size. (Some pkg functions may assume other *default* values for missing args.)

Dicts are simple *assoc* lists, storing *key-value pairs* that are later retrieved with the key. Like other variables, both the key and its value can have any form, although the key is most often a short string. Once a key-value pair is stored in the dict, it *cannot* be removed. But the value side of the pair can be easily changed.

> The Dict (or Arr) itself can be removed by assigning another value (often `Int 0`) to the variable that holds it. The Dict (or Arr) will be *garbage collected* when there are no more variables *pointing* to it.

Arrays and dicts can also be constructed with specific initial content:

```
MyArr = ["this", "is", "interesting"];
ArrX = [Var1+2, OtherArr[3], MyFunc(Var3+4)];
MyDict = ["W1":"this", "W2":"is", "W3":"interesting"];
```

The contents of the `[]` constructors are *evaluated*, so named variables are accessed and any indicated operations (such as indexing into `OtherArr` or calling `MyFunc`) are performed. Similarly, a string value must be quoted, lest it be interpreted as a named variable.

Both arrays and dicts can be *indexed* using `[]` brackets:

```
MyArr[12] = MyArr[X + 2] + (MyDict[“Count”] * 2);
```

In the case of arrays, the index *must* be an integer, but it can be any key type for dicts. Importantly, the contents of the index is evaluated, so the string “`Count`” in `MyDict["Count"]` must be quoted. In contrast, `MyDict[Count]` will use the *value* stored in the `Count` variable as index.

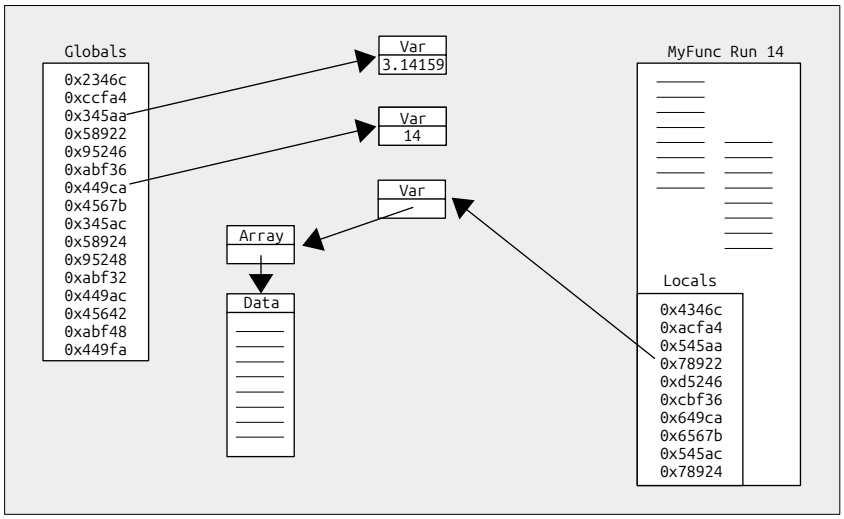
Dicts can also be *indexed* using the dot format:

```
MyArr[12] = MyArr[X + 2] + (MyDict.Count * 2);
```

The dot format *assumes* a string key name so `MyDict.Count` is exactly the same as `MyDict[“Count”]`. The dot provides its own implicit quotes around the key that follows (`Count`) and will *not* evaluate it.

Variables are named for the benefit of human programmers. The compiler just translates them to indices into a storage list. Once compiled, bytecodes deal only with the index and know nothing of variable names. In fact it takes substantial extra work to include global variable names in the running script for debugging displays!

In the diagram below, `MyGlobal` (holding *Pi*) is the 3rd entry on the list of globals and `ArrX` is the 4th local variable in the 14th run of `MyFunc`. These lists simply point to the *storage* space for the variable. In the case of numbers (integer or float), the values are stored directly in the variables. But for anything more complicated (such as an array or dict), the variable will in turn point to the storage for that data structure, which may itself have multiple parts.



scScript uses Garbage Collection to recover memory from data structures that can no longer be accessed. For example in:

```
MyVar = Array(7);              // Allocate array with 7 elts
…
MyVar = 3.14159;               // Replace the array with a Flt!
```

The old value of `MyVar` (the array of 7 elts) is replaced, rendering the array itself inaccessible! But scScript uses a `RefCount` to track the number of references to the array. When this `RefCount` reaches

zero, the storage taken by the array is immediately recovered. (scScript also has a Mark/Sweep garbage collector for handling circular references that cannot be recovered with `RefCounts`.)

scScript allocates local variables within the top (body) block of a function, even if defined in nested sub-blocks. The current (ByteCode) instruction set refers to these locals using a 1-byte index. But each function currently reserves 7 special local variables as *registers*. In addition, index `0x00` and `0xFF` designate the runtime stack (differentiates data vs reference address on stack). As a result, a function is limited to only 247 locals. While these will be allocated (hoisted) in the top block of a function, they are scoped and accessible only within the sub-block that defines them. scScript is currently not smart enough to share/re-use variable allocations in disjoint nested sub-blocks, so all count towards the limit.

The system also allows for up to 65K global variables, defined in the *main* body of the script. While these globals are indexed with a 2-byte designator, they cannot be directly addressed in the instruction set. Instead, they are first *aliased* with the aforementioned local registers. These are repurposed and reused as necessary, so all globals can be manipulated within a given function. Importantly, the main scScript body is itself regarded as an implicit function. Therefore, in addition to registers, sub-blocks within the main body define local variables and there can only be a *total* of 247 such variables.

scScript treats functions as simple variables (whether local or global), but there is one difference. The scope of a defined sub-function is the *top* level of the function that defines it. So even if `MySubFunc` is defined 5 levels deep within `BigFunc`, it will be accessible throughout `BigFunc`. (This only applies if the variable is explicitly declared with a `Function` keyword; not if it is initially declared as a `Var` and then assigned a nameless lambda function!)

DEFs

scScript supports (compile time) naming of pre-defined constants. For example, a program can use the following to explicitly name a few:

```
Def   #Monday = 1, #Tuesday = 2;      // Integers
Def   #Name = "George";               // Strings
Def   #Big = 99E20;                   // Floats
Def   #VeryBig = #Big * 2;            // Def based on previous Def!!
```

While the leading # character is not mandatory it is *strongly* encouraged. scEmacs will only recognize and colorize these Def names if they start with #.

Additionally, key names used to access Dict with the dot format are normally *not* evaluated. So `MyDict.SomeDay` is really read as `MyDict["SomeDay"]` where the key is taken verbatim as a small string. But the # starting the key in `MyDict.#Monday` will trigger an evaluation, and scScript (using the Def) will read it as `MyDict[1]`. Likewise, `MyDict.#Name` will be processed as `MyDict["George"]`.

But beware that the lack of # in

```
Def HisName = "John";
```

will result in

```
MyDict.HisName
```

being simply read as

```
MyDict["HisName"].
```

> But `MyDict[HisName]` will be read as `MyDict["John"]` because `[...]` do evaluate their contents!

scScript is limited to Def naming of integers, floats, and strings, as these are the only literal types. Importantly, Defs are only for the compiler. The runtime script is blissfully unaware of names, whether Defs, functions, or variables, it only sees values and memory locations. So Defs can be used to help program legibility with no impact whatsoever on runtime performance.

The scope of a Def extends to the end of the lexical `{...}` block in which it is defined. Any nested sub-blocks can simply inherit and use them. But such a nested sub-block can also re-define an *inherited* Def and lower-level blocks within it will then see and inherit the re-defined value.

Most packages (`Sys`, `Ed`, `Str`, `Pat`, etc.) will provide their own predefined set of Def constants (often for flag names). These are accessed with the usual *dot* package notation, such as `Str.#Zulu`, just as pkg functions (e.g. `Sys.Print`) are used. Different packages may have Defs with the same name, but their values will usually be different.

Functions

Functions in scScript are passed around as nameless lambda closures stored in variables. They are a first class data type that can be freely stored, returned, and assigned, just as integers, strings, arrays, and dicts. The function *itself* does not have a name, that belongs to the variable that holds its closure.

Functions can be declared and defined globally as well as within other functions. So a function can return a sub-function that is later stored in a dict and called by a subsequent one that retrieves it.

A simple (somewhat pointless) function definition with no formal parameters can be:

```
Var MyFunc = Function()() Show.Globs();
```

or more commonly written as:

```
Function MyFunc()() Show.Globs();
```

The preferred latter form, which *implicitly* declares `MyFunc` as a variable, is slightly more concise and perhaps more stylistically elegant (scEmacs also knows to hilite the `Function` keyword making it very visible in `script-mode`). But a new function can be defined and assigned to a var long after the variable has been declared.

```
Var ArrX, ArrY, ArrFunc;
...
ArrFunc = Function()() {blah blah blah… }
```

However, it is *not* possible to use

```
Function ArrFunc()() {blah blah blah… }
```

for this purpose, as it would attempt to *re*-declare the previously declared `ArrFunc` variable!

One can *pre-declare* a function (must be without parameters) to satisfy coding dependencies, such as for mutual recursion:

```
Function MyFunc;                            // Pre-decl with no arg lists!
...
Function OtherFunc(A)() {MyFunc(A); … }     // Uses the pre-decl of MyFunc!
...
Function MyFunc(X)() {OtherFunc(X); … }     // Finally define MyFunc here...
```

Every function returns a value. This is normally specified with:

```
Return N;
```

where N can be a literal value (`Int`, `Flt`, `Str`), a var, a function call, an arr/dict, or even a function closure. A simple `Return;` or just reaching the end of the function *without* an explicit `Return`, will also return `Int 0`. (Note that "`Function F()();`" will create an empty function that returns `0` immediately!)

A function definition requires both *in* (pass-by-value) and o*ut* (pass-by-reference) parameter lists, even if both are empty. A function call, however, requires only the *in* list, *out* args are optional.

From within the function, all *in* and *out* parameters look like local named variables. When the function is called, *in* parameters are initialized with the *values* provided by the caller. For example, if:

```
Function TheFunc(InA, InB, InC)(OutA, OutB, OutC) { ...body... }
```

is eventually called with:

```
MyVar = 14;
TheFunc(MyVar, "This", 3.14)(OtherVar, MyArr[2], MyDict.Size);
```

the body of the function will see a local variable `InA` with the value `14`, `InB` with the string "`This`" as value, and `InC` storing `3.14`. `NewFunc` can re-assign, increment, or otherwise manipulate these values within its body. But the outside world, including the caller of `TheFunc` will not see these changes nor know of `InA`, `InB`, or `InC`. The *in* list simply passes *values* into the function (hence *pass-by-value*) which can use and manipulate them at will. But just like C, the outside world *will* be able to see the new contents if entire arrays or dicts are passed into a function that changes what is stored *inside* them!

The above call also has an *out* list where `OutA` *becomes* `OtherVar`, `OutB` *becomes* `MyArr[2]`, and `OutC` *becomes* `MyDict.Size`. `TheFunc` can also re-assign, increment, or manipulate these variables, treating `OutA`, `OutB`, and `OutC` as any other local variables. But the result of such manipulations will be directly manifested outside `TheFunc` where these storage locations exist and are quite visible. Any changes to `OutA` are exactly reflected in `OtherVar`, which can be a global or perhaps a local variable in the caller of `TheFunc`. Similarly, setting `OutB` to `9` will have the exact same effect as setting `MyArr[2]` to `9`, because `OutB` is just a *reference* to `MyArr[2]`, therefore points to the same storage location. So *pass-by-reference* will set the formal *out* parameter to a pre-existing storage location.

scScript functions can be called with *fewer* args than what is specified in their definition. So `TheFunc` (defined above) can be called with:

```
TheFunc(3)(OtherVar);
```

This will pass `3` to `InA`, but also a *default* `Int 0` to `InB` and `InC`. On the *out* side, `OutA` *becomes* `OtherVar`, but `OutB` and `OutC` will be given their own *temporary new storage*, each initialized with `Int 0` as value. So `TheFunc` will run as if called by:

```
TheFunc(3, 0, 0)(OtherVar, NewVar1, NewVar2);
```

where `NewVar1` and `NewVar2` are implicitly declared as

```
Var NewVar1 = 0, NewVar2 = 0;
```

But of course, these two are not named and cannot be seen outside of `TheFunc`.

scScript functions can also be called with *more* args than what is specified in their definition. These extra args obviously cannot be mapped to named formal parameters. But they can be extracted in array form with a simple `Sys.GetArgs` call.

Suppose `TheFunc` is called with:

```
TheFunc(MyVar, "This", 3.14, "That", 2.718)
       (OtherVar, MyArr[2], MyD.Height, MyD.Weight);
```

All *in*/*out* args can be extracted with: (from within `TheFunc` itself)

```
Var InArr, InCount, OutArr, OutCount;             // Declare locals first
Sys.GetArgs()(InArr, InCount, OutArr, OutCount);  // Get proper values for them
```

Note: `Sys.GetArgs` uses its *out* parameter list, not *in* like most functions!

`Sys.GetArgs` will pack *in* list vars (from `TheFunc`) into `InArr`, so `InArr[2]` (arrays are `0`-based, like C) will get `InC` (value `3.14`), and `InCount` will get the arg count of `5`. `OutArr` will pack the vars passed to

the *out* list into an array, just like the `InArr`, but the elts *will be* the vars! Rather than create/allocate new elts for the arrays, they will use the *same storage* as their respective args. So `OutArr[3]` will *be* `MyD.Weight`, and setting `OutArr[3]` to `150` will set `MyD.Weight` to `150`! As expected, `OutCount` will be `4`.

`Sys.GetArgs` is itself a function call. So its *output* variables are placed in its *out* list. One can also call `GetArgs` itself with fewer args, perhaps as

```
Sys.GetArgs()(InArr, Null, OutArr)
```

if not interested in getting the arg counts (no need for a trailing `Null`). Either way, `Sys.GetArgs` will discard any previous values in its *out* args as it loads in parameters from its function. (Note: `Null` or `Void` are valid *out* list args and naturally used as placeholders!)

But `Sys.GetArgs` works in a slightly unexpected way if `TheFunc` is called with fewer args than formal parameters. In that case, the `InArr` or `OutArr` will also contain fewer entries than formal parameters! So the default entries in the *in* list and the temporary storage for the *out* list parameters will *not* show up in the `GetArgs` results. While logically correct, this may come as a surprise.

As mentioned above, every function returns a single value (which defaults to `Int 0`). But the *out* list allows functions to conceptually export *more* than one value. This is very much in keeping with the spirit of C and an original way of providing that functionality without the unwelcome addition of pointers (or *types*) in scripting. The only downside is the minor *weirdness* of an extra `()` in every function definition!

Note: Functions can be defined with empty *in* or *out* lists. Functions may also have `Null` (or `Void`) in their argument lists. These are generally used as *placeholders* for expected, but ignored, args. While legal, it makes little sense to have extra `Null` args *after* the last named one.

`F(Null, Null, Null)(Null, Null)`	compiles as	`F()()`
`F(Null, Null, A, Null)(Null, X, Null)`	compiles as	`F(Null, Null, A)(Null, X)`

The 2nd example is useful if the script expects to call F with F(V1, V2, V3)(Out1, Out2) but the function itself really only cares about V3 and Out2.

Limits: ByteCodes, low level ops that run scScript, can only address local variables with a single byte. So the total number is limited to `256`. Unfortunately, `0x00` and `0xFF` are reserved, as are `7` local *registers* used to alias global variables. So the total number left is only `247`. Importantly, *in* and *out* parameters are included in this count (as are locals *closed over* from superior functions). So a function with `10` *in* and `7` *out* parameters, can have only `230` local variables. Yes, it is truly difficult to write a function with only that many locals, but to your best.

Strings

scScript supports two string formats, *Short* (quoted) strings and *Long* (bracketed) ones. The distinction is one of format and content, *not* length. Short strings must be delimited by matching single *or* double quotes. But they can trivially include quote characters dissimilar to the delimiting ones. For example:

```
MyVar = "John said 'he will be late'.";
```

or

```
MyVar = 'John said "he will be late".';
```

These short strings may also include inline *escape* sequences:

`\'` or `\"`	Insert non-delimiting quotes
`\\`	Insert backslash character itself without escaping
`\n`	Insert newline
`\r`	Insert carriage return
`\t`	Insert tab
`\z`	Skip inline whitespace (space, tab, newline)
`\#`	No-op separator if a digit has to follow a numeric char spec
`\ddd` or `\[ddd]`	Insert char with decimal spec
`\xfff` or `\[xfff]`	Insert char with hex spec (typically 2 digits, leading 'x' or 'X')
`\okkk` or `\[okkk]`	Insert char with octal spec (typically 3 digits, leading 'o' or 'O')

Where numerical character specs can go all the way to `1114111` (`0x0010FFFF`) for Unicode (UTF-8).

So the above example could also be written as:

```
MyVar = "John said \"he will be late\".";
```

Long strings, in contrast, do *not* support any inline escape characters. They begin with `|>` with 0 to 8 intermediate `-` (`dash`) characters, such as `|--->`. The string must end with the *exact* mirror image delimiter sequence. So the above example can be:

```
MyVar = |>John said "he will be late".<|;
MyVar = |->John said "he will be late".<-|;
MyVar = |-->John said "he will be late".<--|;
```

etc.

Importantly, long strings will ignore an *initial* and *final* newline character, if included. This is so source files can include long strings verbatim and without disturbing the formatting:

```
MyVar = |---->
This text includes |>, |->, and |-->!
But it is represented verbatim,
with just two intervening return characters!
<----|;
```

Numbers

scScript relies heavily on C (and its runtime library, `libc`) for its numerical computations and math support. Numbers can be represented in integer (`Int`) or floating point (`Flt`) format. The former is really just 64bit *signed* integers (`Int64` in sources) while the latter is standard C double float (`double` in C). In addition to C arithmetic operators, scScript also provides a `Math` package with support for logarithms, exponentiation, square roots, and basic trigonometry.

Integers can be written in signed decimal, hexadecimal (hex), and octal format:

```
X = -42;          X = -0x2a;          X = -052;
```

All three assign `-42` (decimal) to `X`. Note the leading `0x` for hex and leading `0` for octal. Importantly, scScript *does not* sign extend, so `0xFF` is just `255`, *not* `-1`!

Floating point numbers can only be written in signed decimal or hex:

```
X = -42.1875;       X = -0x2a.3;
```

Both will assign `-42.1875` to `X`. In the case of hex, `2a` is decimal `42` and `.3` brings `3/16` as each digit is a power of `16` on the denominator. Similarly, `.35` in hex would mean `3/16 + 5/256 == 0.20703125`. But, `-052.3` will be *promoted* to decimal (not octal) and read as simply `-52.3`, despite the leading `0`.

Floating point numbers, both decimal and hex, can also be written in exponential format:

```
X = 8.64E4;         X = -0x2a.3P11;
```

Both will assign `86400.0` to `X`. In the decimal case the number is `8.64 * 10`^{4} while the hex number is really `42.1875 * 2`^{11}. Whereas the `E` (or `e`) for *Exponent* is readily obvious, the hex format must do with `P` (or `p`) for *Power* as `E` is a valid hex digit! Also, the base of an exponential number can be in decimal or hex, but the exponent is *only* written in decimal. So the `11` in `-0x2a.3P11` really means 2^{11} and not 2^{0x11} which would be `64` times bigger.

When two large integers are added (or multiplied), it is entirely possible to overflow the 64 bits and create numerically erroneous results as bits roll over with no obvious warning. But this is not the case with floating point numbers. These reserve special values to flag overflow/underflow conditions as *`Inf`* and *`-Inf`*. Subsequent numerical operations will preserve these values and their specific semantics. For example: *`Inf`*`/2` and (*`Inf`* `- 1000.0`) still yield *`Inf`*. A *`NaN`* (Not-a-Number) value is also reserved and created for specific situations, as in (`0.0/0.0`). *`NaN`* has the surprising attribute of *not* being equal to itself, as it is *outside* the closed realm of numbers! (But, *`NaN`* is *identical* (`===`) to itself!)

`Print` and `Write` functions in scScript will use `libc` for displaying numbers. These will therefore display `inf` and `-inf` where appropriate. Strangely though, `libc` currently displays *`NaN`* with a negative sign, apparently because the floating point value reserved for *`NaN`* is negative. These *special* values are typically generated as part of degenerate floating computations, but on the off chance that one actually needs such values, the `Math` package defines `#Nan`, `#Pos_Inf` and `#Neg_Inf` constants.

> Note: *`NaN`* and *`Inf`* are theoretical concepts.
> "`-nan`", "`inf`", and "`-inf`" are what scScript prints, thanks to `LibC`.
> `#Nan`, `#Pos_Inf`, and `#Neg_Inf` are Defs in the Math package that can be used in scripts.

scScript will optimize simple arithmetic at compile time. So the statement:

```
Math.Sin((45 * Math.#Pi) / 180)        becomes      X = Math.Sin(0.785398)
```

But as of this writing, the compiler does not pre-compute the Sin function itself!

Running scScript

The current version of scScript is run and controlled exclusively from scEmacs. The most common strategy is to open an `.sc` file in scEmacs, usually with `C-x C-f` for `Find File`. Once that `.sc` text file has been loaded into the active pane, the command `M-x script-do` will compile and run the *whole* script. The results will show up in the `*Output*` buffer on scEmacs. (If `*Output*` is not already visible, the main frame of scEmacs will split to show that buffer in a lower pane.)

The very useful command `M-x script-wipe-output` serves to

a) Display the `*Output*` buffer if not already visible *and*
b) Clear out the `*Output*` buffer (*only* the buffer) of any previous contents.

Multiple (same or different) scripts can be run in sequence, with the resulting output getting automatically appended to the `*Output*` buffer, which can be *wiped* as desired. The `*Output*` pane will automatically scroll to show the new (appended) lines, but *only* if its cursor remains at the end of the buffer. Naturally, the `*Output*` buffer can also be written (`C-x C-w`) to a file and saved for later.

scScript can be compiled with a `DEBUG` flag and `#define` macros that allow for a great deal of control on the compiler output. For example, it can be set to show how the compiler interprets every single line of the script and show bytecodes generated for each. But adding and displaying `*Output*` is the slowest thing scScript does, so this will drastically slow down the compile process. Otherwise, compiling without the debug displays will be *instantaneous* for most scripts.

Each time scScript executes, it leaves the global *memory state* around. This includes all global variables and their stored values. The `M-x script-try` command can then be used to compile and run selected *chunks* (fractions) of a script using these values. For example, after running a script with `script-do`, one can select: (hi-lite within scEmacs)

```
Show.Globs()
```

and `M-x script-try` it to see a display of all global variables and their values. In that sense, `M-x script-try` will compile and run script chunks incrementally, in the memory state left behind. The script chunk will also have access to scScript *strings*, *constants*, and *defs* left by the previously run script. In turn, it can declare/define new globals, change values, introduce new strings, or define new constants and defs! So script-try can be used to *incrementally* debug/develop an scScript without having to run the whole thing repeatedly from the top.

> A `try` chunk can change the value stored in a global, but it cannot change any constants or strings. All it can do is define *additional* constants and strings that it, or subsequent `try` chunks, will use. Similarly, while it can change a `Def` value, the previously compiled code already has the old values incorporated. Any changes will only effect the compilation of subsequent chunks. (`Def` are only for the compiler. Once compiled bytecode use/reference `Def` values directly.)

The command `M-x script-reset` will expunge the memory state left behind by a previous script run, creating a fresh start. But this is seldom used, as every `script-do` (and many other script commands) will first do an implicit reset. (`script-wipe-output` does *not* `script-reset` and vice versa.)

scScript also provides the `M-x script-compile` command to compile the entire script (obviously in a fresh memory state) and write out a `.bsc` object file. It will not *run* the script, but the object file it generates can subsequently be loaded and run using `M-x script-run`. The object file has everything tightly packaged and pre-compiled, so loads very quickly. Currently, `script-try` is quite limited after running a `.bsc` file, as the object file does *not* store global variable names! Without names, the subsequent script chunk cannot find and access global variables and their values.

Importantly, `script-run` will prompt for a filename, so can be called from any scEmacs pane. It can actually *run* a `.bsc` object file *or* an `.sc` script. In the latter case, script-run will *compile*, *write* a .bsc file, then *run* the script.

scScript also allows a running script to *load* a sub-script into a variable and execute it, albeit in a subordinate capacity. `Script.Load` will accept a file/pathname string. It will compile and load the `.sc` file or just load the `.bsc` object code into memory, ready for execution. A subsequent `Script.Exec` call will then execute the loaded sub-script. In all cases, the sub-script runs in its own environment and cannot directly alter memory in the parent script. (This was initially implemented for *validation* & *verification* suites, but may get expanded in the future.)

Common Errors:

The most common user error in scScript is writing `MyDict.K` instead of `MyDict[K]`, where `K` is a variable. Next is confusing *in* and *out* args, especially for predefined functions, as in:

```
Sys.GetArgs(InA, InC, OutA, OutC);
```

instead of

```
Sys.GetArgs()(InA, InC, OutA, OutC);
```

When using coroutines, it is common to `Yield` without a loop, thus inadvertently terminating a persistent function.

Pattern matching is quite complex and has its very own section on common errors.

Performance

scScript, as it stands, has good runtime speed. It is of course possible to further optimize to excel at given benchmarks; but the system currently should be entirely fast enough for general use. On all but the very longest scripts, compile feels instantaneous and scScript rapidly blows through execution.

When compiled with the `DEBUG` flag, scScript will add *very many* extra checks and redundancies to quickly catch any memory management errors that might creep in during software development. Such errors, if not caught immediately, will slowly corrupt memory and lead to entirely unpredictable results that are all but impossible to trace. The `DEBUG` options will necessarily add a great deal of overhead and

processing, so will drastically slow down the system (as if running `valgrind`). The source code contains many #define options and parameters to fine tune this behavior.

With `DEBUG` and `Compile Display`, scScript will show each line of source script as it is compiled and write out exactly what ByteCodes are assembled. While not so useful for script writers, this is of immense value to system development and debugging as well as a great instructional tool. Naturally, this display will slow everything as writing to `*Output*` and updating scEmacs is time consuming.

Importantly, the header message displayed at the top of fresh `*Output*` buffers will list the current compile options, and what performance speed to expect:

```
scScript Version: 2.4  [Tue Apr  7 14:44:45 2026]
        RefCount: On                    Reg: 7
           Debug: On       Compile Display: On
          M/S GC: On            Safe Check: On
  Free Mem Check: On         All Slab Check: On
       GDB Print: Off      GDB Debug Print: Off
           Speed: *SLOW*
```

scScript source code also has flags to control *garbage collection*, both periodic *Mark-Sweep* GC as well as on-going *RefCounting*. In the case of the former, the initial memory and threshold step size for re-invoking GC are also defined. It is entirely possible to set GC parameters and debug tests to slow the system below acceptable levels, as in forcing frequent collection along with tests that scan entire memory slabs each time a single record is released!

Packages

scScript includes a number of pre-defined packages. These are implemented as efficient C routines and provide useful utility functions (and often defined constants) for invocation from within scripts. They are called as `Pkg.FuncName()` or (`Pkg.#DefName` for constants), but functions in the main (sc) package (and constants defined therein) can be used *directly*, without the `Pkg` name and `dot` prefix.

In addition, the main package also includes *Inline Functions Ids* (`IFIDs`) which are pre-packaged bytecode sequences that are inserted into the compiled code instead of a C function call. These are generally language features such as the `Map` iterators or `Array/Dict` constructors. They are invoked with a seemingly normal func call from the script, as in `"Array(12);"` but the generated bytecode is more efficient and skips the function set-up and call. Like any function in scScript, these can be called with fewer or more args. `MapI` and `MapK` also differ from normal functions in having a *body* (code statements) to execute for each iteration.

Main (sc) Package

IFIDs: (Inline Functions)	
MapI MapI(S)(V, I) {Body... }	Iterates over every cell (even *unpopulated* ones) in S (array or dict). V gets the cell value and I gets its numeric (0-based) index. Note: "MapI(S)(V, I);" does NOTHING as it has an empty body! V only copies the value, use "S[I] = 12;" to change elts in S.
MapK MapK(S)(V, K) {Body... }	Iterates over every elt (only *populated* cells) in S (array or dict). V gets the elt value and K its key if dict or its (0-based) index if array. Note: "MapK(S)(V, K);" does NOTHING as it has an empty body!
Array A = Array(N);	Returns an empty array with N (unpopulated) cells. Uses default minimum if N is 0 or missing.
Dict D = Dict(N);	Returns an empty dict sized optimally for N pairs. Uses default size if N is 0 or missing.
Int X = Int(N);	Returns the value of N converted to an integer. • Float numbs get their *floor*, • Non-numbers (Closure, etc.) get their memory loc, • Str get their index #.
Flt X = Flt(N);	Returns the value of N converted to a float (double). Non-numbers get their Int(N), converted to flt.
GetRC R = GetRC();	Returns RC (RunContext) of enclosing routine. RC is used for non-local return/yield as in "Return X OutOf R;" Note: Tail recursion optimization is NOT performed if function gets RC.
Submit x = Submit();	Coroutine (P) submits its RC to its original caller (M) and goes to sleep. M can RESUME P later, P can SUBMIT only once. Submit always returns 1 to its own routine.
Resume N = Resume(S_RC);	M calls sleeping P to execute. RESUME will return what P will YIELD/RETURN. P_RC must be RC previously returned from P when it did original SUBMIT.
Release x = Release(S_RC);	M Releases (zaps) sleeping P. RELEASE always returns 1. P_RC must be RC previously returned from P when it did original SUBMIT.
Yield and *Return*	*Yield* and *Return*, are part of the language, not IFIDs!
Functions:	
TypeOf X = TypeOf(V);	Returns the type constant for the value type: #Type_Unknown... #Type_Script.
SizeOf X = SizeOf(V);	Returns the *conceptual* (current) *cell* count of V. Capacity for Arr and Dict, *byte* count for Strs and Clos, 1 for all others.

CountOf X = CountOf(V); Y = CountOf(ArrA, I); Z = CountOf(DictA, K);	Returns the conceptual (current) *elt* count of V. Counts *populated* cells for Arr, pairs for Dict, enclosed functions for Clos, and *char* count (UTF-8 chars are 1-4 bytes) for Strs. 2-arg version returns 1 if the ArrA[I] or DictA[K] is *populated*, 0 if not.
Write X = Write(U, V, W...);	Writes out concatenation of all *in* args to end of *Output* (scEmacs) buffer. Returns total number of bytes written. Write accepts any valid value!
Writeln X = Writeln(U, V, W...);	Same as Write, but adds an extra \n at the end. (Yes, Pascal was a good language :-)
Defs:	
Logic	#True, #False
Types	#Type_Unknown, #Type_Int, #Type_Flt, #Type_Str, #Type_Arr, #Type_Dict, #Type_Ext, #Type_GExt, #Type_RC, #Type_Pat, #Type_FExt, #Type_Func, #Type_Script. Closures are exposed as #Type_Func to script writers. #Type_Ext uses memory block from client. #Type_GExt uses script array memory, can *Grow*. #Type_FExt uses script chunk memory, is *Fixed* and cannot grow.

Sys Package

Functions: (Utility and Misc)	
GetArgs X = Sys.GetArgs()(IArr, IC, OArr, OC); (Call with OUT args!!!)	Returns *TotalCount* (OutCount<<8 + InCount) of args for caller. Stores array of all *in* args (for caller) in IArr. Stores count of all *in* args in IC. Stores array of all *out* args in OArr. (Elts *are out* args!) Stores count of all *out* args in OC.
Print X = Sys.Print("%ld", V);	Does C-style *printf* to *Output* and returns total bytes written. scScript Ints are LONG, use %ld, %lx, etc. Both %f and %lf will work. (Sys.Print uses libffi to call into libc... will crash with bad args!)
Error Sys.Error("Script is unhappy!");	Exits the script with specified error str or default msg if none. Does NOT return a value to the script! User can change the script and recompile/rerun.
Exit Sys.Exit("Script is done!");	Exits the script normally with the specified str or default msg. Does not return a value to the script! Script exits normally, can be run again.
Abort Sys.Abort("Script had problem!");	Exits the script with the specified error str or default msg. Does not return a value to the script! Goes directly into low-level debugger, *cannot* continue!
Debug X = Sys.Debug(M);	Prints M to stdout and drops into low-level debugger (raises SIGINT). User can "Continue" from debugger to resume operation. Returns int 1 to script.
Pause X = Sys.Pause(M);	Prints M to stdout, Pauses the running script. User can resume execution with *M-x script-resume* command. Returns int 1 to script (when resumed).
gc X = Sys.GC();	Does a full Mark/Sweep Garbage Collection. Will return byte-count of freed memory blocks. GC will also dispose of empty *slabs*, but does not report that size.
Reverse X = Sys.Reverse(Val); X = Sys.Reverse(Val, S); X = Sys.Reverse(Val, S, E);	Reverses Str/Arr/Dict Val. Returns reversed Val or 0. S is optional 0-based Start index and E is optional End (0 means all). Arr and Dict are reversed in place, but creates and returns new Str! S is INCLUSIVE, but E is EXCLUSIVE (one past last). NOTE +++
Copy X = Sys.Copy(Val); X = Sys.Copy(Val, S); X = Sys.Copy(Val, S, E);	Copies and returns a new Str/Arr/Dict Val, returns 0 for other types. S is optional 0-based Start index and E is optional End (0 means all). Returned value is the copy, Val itself is unchanged. S is INCLUSIVE, but E is EXCLUSIVE (one past last). NOTE ***

SeedRand X = Sys.SeedRand(V);	Initializes *seed* for random number generator and returns it. Use consistent V for most reproducible pseudo-random sequence. If no V arg or 0, will use *time(NULL)* OS call and return that value.
Rand X = Sys.Rand(); X = Sys.Rand(V); X = Sys.Rand(N, M);	Returns positive random number generated from *seed*. Args, if given, must be positive, all int or all flt. If called with no *in* arg or with V == 1, returns positive 32-bit int. If called with V == 2, returns positive 64-bit int. If called with flt V arg, returns flt from 0.0 up to 1.0. If called with int N to M, returns int from N to (not including) M. If called with flt N to M, returns flt from N to (not including) M.
Time T = Sys.Time(Flag)(BRT);	Returns Int number of seconds since start of Unix epoch. BRT, if given, will get a new #Type_FExt struct for *broken down time*. Optional Flag is Sys.#Zulu (global) or Sys.#Local (default) for BRT.
BRTime Sys.BRtime(BRT, F1, F2...) (V1, V2);	Returns Int 1. Places flagged (F1, F2, etc.) values in corresponding variables (V1, V2, etc.). Each *in* flag must have a matching *out* var. Flags range from Sys.#Sec to Sys.#IsDST. Unlike C, #Year is actual year (not 1900-based) and #Mon is 1 for Jan!
Ticks Tk = Sys.Ticks();	Returns clock (processor) ticks (Int) since re-boot. (This system value rolls over quite often as up to 1E7 tks/sec!)
Secs X = Sys.Secs(Tk);	Returns elapsed time in secs (Flt) using T (time *Ticks*) since event. (Tk is usually TKEnd - TKStart)
Defs:	
Time flags	#Zulu, #Local
BRTime flags	#Sec, #Min, #Hour, #MDay (day of month), #Mon, #Year, #WDay (day of week), #YDay (day of year), #IsDST (Daylight Savings)

+++ `Sys.Reverse` will return `Int 0` for all Val types other than Str/Arr/Dict. It will return the original Str/Arr/Dict (or `Int 0`) if `S` and `E` specify a non-existent range (`E` is trimmed to size if too large).

Note: `Sys.Reverse` will alter Arrays and Dicts *in place*, but strings are immutable, so only the return string is reversed!

Given `A = Array(); A[0] = "Zero"; A[1] = "One";` will create an array with `2` strings at index `0` and `1`, plus `6` empty slots at index `2` to `7`. Therefore `Sys.Reverse` on this array (without `S` and `E` args) will yield an array with `6` empty initial slots, `"One"` in `A[6]`, and `"Zero"` in `A[7]`. But `Sys.Reverse(A, 0, 1)` can be used to reverse the two initial (populated) slots. (`B = ["Zero", "One"];` does not have this problem, as `B` will only have `2` slots.)

`Sys.Reverse` will reverse *pair* order in dicts. This will only manifest in `MapK` (not even `MapI`) iteration order as normal `Dict.Key` access (and hash array indexing) is independent of pair ordering. But `Sys.Reverse(MyArr)` will actually change `MyArr[I]` access, because `Reverse` will re-arrange array contents. Worse, as noted above, `Sys.Reverse(MyStr)` appears to do nothing! Strings are immutable and cannot be changed in place. So `NewStr = Sys.Reverse(MyStr);` is required to get a reversed string.

S and E refer to the nth (`0-based`) array elt and pair dict. In the case of strings, they indicate the nth *character*, *not* the nth byte, as a string can have multi-byte UTF-8 characters.

*** `Sys.Copy` will only work on Str/Arr/Dict and will return `Int 0` for all other types. It will also return `Int 0` if `S` and `E` specify a non-existent range. A copy of the full string *is* the full string itself, the system will *not allow* copies of the same exact string.

Show Package

Functions: (Data display for development and debugging)	
Vars X = Show.Vars(Flag)(V1, V2);	Writes details of *out* vars (V1, V2, etc) to *Output*, then returns Int 1. Flag determines detail level, defaults to #Low. If no vars, will display all locals in current function run. Local var names are currently NOT stored, so labels var categories. NOTE: +++
Globs X = Show.Globs(Flag);	Writes details of all globals to *Output*, then returns int 1. Flag determines detail level, defaults to #Low.
Run X = Show.Run(Flag);	Writes details of current Run record to *Output*, then returns int 1. Flag determines detail level, defaults to #Low.
Script X = Show.Script(F1); X = Show.Script(F1, F2);	Writes details of current Script to *Output*, then returns int 1. F1 determines detail level of the script, defaults to #low. If F2 is given *and* called from a sub-script, then superiors are shown.
VM X = Show.VM();	Writes details of VM, all memory slabs, loaded packages, defined C functions, Run Stack, and even the current Script(s) to *Output*, then returns int 1.
Mem X = Show.Mem(T); X = Show.Mem(T, O);	Writes details of T (Type) memory slabs to *Output*, then returns int 1. If given O (Ord) will show only that slab for the specified Type, otherwise defaults to Ord 0 which means all. (T and O match numbers in Show.VM.)
VMStrs X = Show.VMStrs();	Writes details of the main string hash table (stored in STab) and every stored string to *Output*, then returns int 1.
Strs X = Show.Strs();	Writes every string indexed by the compiler to *Output*, then returns int 1. Note: VMStrs will show *many* other strings NOT used by this script!
Pkg X = Show.Pkg(Id); X = Show.Pkg(Name);	Writes details of the specified Pkg to *Output* then returns 1. Pkg can be specified by Id (int) or Name (str). If Id is negative or missing, all pkgs will be detailed.
Defs:	
Display level flags	#Min, #Low, #High, #Max

Note: Various displays will show (Ref:1) if SC_GC_REFCOUNT compile flag is 0, as the RefCount field is still there, just no longer counted or used for purging memory.

+++ The main body of the script is also a function, albeit one that is *implicitly* called. It will not have any *in* or *out* formal parameters but usually has multiple *Registers* and *may* also have *Local* variables. The latter are defined in {} sub-blocks of the main script and are *hoisted* to the top.

Str Package

Functions: (Str are immutable, all created strs are hashed and reused!)	
Print S = Str.Print("%ld\n", Val)(L);	Does C-style *printf* to a new Str and returns it. First *out* arg, if given, will get the StrLen (bytes). scScript Ints are LONG, use %ld, %lx, etc. Both %f and %lf will work. (Str.Print uses libffi to call into libc... may crash if bad args!)
Write S = Str.Write(U, V, W...)(L);	Writes out concatenation of all *in* args to a new Str and returns it. First *out* arg, if given, will get the StrLen (bytes). Write accepts any valid value, turns it into a descriptive string!
Writeln S = Str.Writeln(U, V, W...)(L);	Same as Str.Write, but adds an extra \n at the end.
ReadN E = Str.ReadN(S, B, F)(N);	Reads number N (Int or Flt) from str S at byte pos B (0-based). Returns E (EndPos of N) if successful, or 0. Can read +/-, Decimal, Octal, and Hex. Can read Flt in regular or exponential form (Decimal/Hex only). F defaults to #Sep, use #NoSep to read without separators. Note: ***
Form S = Str.Form(U, V, W...)(L); S = Str.Form(U, N, V, W...)(L);	Form and return a Str from multiple copies of arg strings. *In* args can be numbers and strings, nothing else. Each N means concatenate N copies of each subsequent string. New N resets for next strings. First *out* arg, if given, will get StrLen (bytes).
FTime S = Str.FTime(FStr, T, F)(L);	Creates and returns the formatted time string (like *strftime* in C). FStr if given, specs the format, defaults to "%x %X" Date+Time. T, if given, specifies the time (ticks), defaults to now! F, if given, specifies #Zulu or #Local (default) time. First *out* arg, if given, will get the StrLen (bytes).
Del S = Str.Del(S1, B, E, F)(L)	Creates and returns a copy of S1 str with the specified range deleted. S1 as well as B (Begin) and E (End) args are required. Value of -1 for E, means go to the end. F, if given, combines flags: defaults to #Pos + #Byte. First *out* arg, if given, will get the StrLen (bytes). Note: +++
Sub S = Str.Sub(S1, B, E, F)(L);	Creates and returns a subset of the specified S1 string. S1 as well as B (Begin) and E (End) in args are required. Value of -1 for E, means go to the end. F, if given, combines flags: defaults to #Pos + #Byte. First *out* arg, if given, will get the StrLen (bytes). Note: +++

Adjust
S = Str.Adjust(S1, F)(L);
	Creates and returns a copy of S1 str with indicated adjustments.
	F can ask for:
	  #Low, #Up, or #Cap to change the case (word by word),
	  #Root to replace accented (Latin) chars with their root version,
	  #Lig to expand ligatures to their base letters.
	First *out* arg, if given, will get the StrLen (bytes).

BLen X = Str.BLen(S);	Returns the byte len of S str, or -1 if not a str.
CLen X = Str.CLen(S);	Returns the char len of S str, or -1 if not a str.
BPos X = Str.BPos(S, CPos)(BLen, BStart);	Returns byte pos (BStart) of nth char (CPos is 0 based). BLen will get byte len of char and BStart gets its start position. Returns -1, BLen of 0, and BStart of -1 for bad CPos. (BStart *out* arg is superfluous as the same value is returned.)
CPos X = Str.CPos(S, BPos)(BLen, BStart);	Returns char pos of nth byte (BPos is 0 based). BLen will get byte len of char and BStart gets its start position. Returns -1, BLen of 0, and BStart of -1 for bad BPos. (BStart *out* arg is necessary because BPos may be mid-UTF-8 char!)
Find X = Str.Find(DataS, S, BFrom, F) (BStart, BEnd);	Returns 1 if it finds S (Str) within DataS (Str), 0 otherwise. Searches DataS starting at BFrom (0 by default). Accepts #Insen/#Sen + #Smart/#Exact + #UniDir/#Dir for F. Sets BStart/BEnd to start/end of matching sub-string in DataS. BEnd (like E above) will be AFTER last matching byte, so EXCLUSIVE. (Note: Because of #Smart, Match in DataS may be shorter/longer than S!)

Defs:	
Unit Flags	#Byte or #Char
Count Flags	#Pos or #Len
Local Flags	#Zulu or #Local
Number Flags	#Sep (default) to accept thousands separator, #NoSep if not. (Uses decimal and separator char defined in sc Params!)
Adjust Flags	#Low, #Up, or #Cap to capitalize. (Default is no case change) #Root to replace accented chars with their roots. #Lig to expand known ligatures to their base chars. (Flags on different lines can be | or + together.) #Norm (#Low | #Root | #Lig) is shortcut flag for *normalizing*. #Norm turns "Löwe Straße" to "lowe strasse" for easier search/lookup.
Find Flags	#InSen (default) or #Sen for case comparison. #Smart (default) or #Exact for handling Unicode accents. #UniDir (default) or #BiDir for matching accents to root. Note: ### (Flags on different lines can be | or + together.)

*** Numbers in data very often use a comma as thousands separator, such as 1,256,853. The default #Sep flag will read and ignore the comma *Sep* chars. Different locales have slightly different rules, so the system requires a minimum of 2 digit separation. It also uses the Dec and Sep chars defined in sc params. But data can sometimes use commas as delimiters between individual numbers, such as 12,4582,13449. In that case #NoSep will view the comma as a break to read three separate numbers, 12, 4582, and 13449. This info also applies to Ed.ScanN, which reads numbers from scEmacs buffers instead of strings.

+++ Specifying a range, for Sub and Del, can be done using #Pos, as in *from position X to Y*, or by #Len, as in *start at X and go L steps*. In both cases, the *steps* can be #Byte or #Char. These will yield very different results for multi-byte UTF-8 chars. In the case of #Byte units, if the count falls in the middle of a UTF-8 char, Del will adjust to *minimize* the segment while Sub will adjust to *maximize*.

Str.Del from #Pos 8 to 10 (assuming #Char) will remove the chars at position 8 and 9. This will be equivalent to starting at 8 with a #Len of 2.

Str.Find will *match* accented chars against their unadorned (root) version in its default #Smart mode. But will require an exact match (adjusting for case) if #Exact is specified. If #Smart is set, under the default #UniDir flag, accented chars in S will *only* match accented chars in DataS, but root chars in S will match their accented version in DataS. With the #BiDir flag, accented chars in S can also match their root versions in DataS. (#UniDir and #BiDir flags are ignored if not #Smart.)

Flag	S (template)				DataS	
#Exact	E	←	✓	→	E	
	E		×		É	
	É		×		E	
	É	←	✓	→	É	
#Smart + #UniDir	E	←	✓	→	E	
	E		✓	→	É	
	É		×		E	(Same as scEmacs and Pat.Match)
	É	←	✓	→	É	
#Smart + #BiDir	E	←	✓	→	E	
	E		✓	→	É	
	É	←	✓		E	
	É	←	✓	→	É	

Adjust above matches for #InSen (default) or #Sen flags...

Ed Package

Functions in the *Ed* package generally deal with scEmacs and its *buffers*. Most will indicate a particular buffer with a `BN` (Buffer Number) obtained from `Ed.ReadFile` or `Ed.GetBuf`. But they can also use `0` (default) to mean the *latest* `Cur` buffer (from `Ed.SetCurBuf`) or `*Output*` if no `Cur` was set. (Write/Writeln/Print will print to `stdout` (gdb, terminal, etc.) if `BN` is -1!)

Many functions accept a `P` (Position), which will be the scEmacs *CursorPos* for visible buffers, where new text is written to. This number can be based on `#Byte` (default) or `#Char` units, depending on the optional `F` (Flag). These will only diverge if the buffer contains UTF-8 (multi-byte) characters. In either case, `P` defaults to `0` while `-1` indicates the buffer *EndPos*.

Some functions will indicate a *range*, using `S` (Start, defaults to `0`) and `E` (End, defaults to `-1`) values. `S` is just a position, so is indicated in `#Byte` (default) or `#Char` units. But `E` can be a position, same units as `S`, *or* a *len* count of `E` (`#Byte` or `#Char`) steps from the originating `S`. An `E` of `-1` (default value) indicates the buffer *EndPos*, whether as position or len. The optional `F` (Flag) will indicate `#Byte` (default) or `#Char` units along with `#Pos` (default) or `#Len` counts.

Note: When using the *Output* buffer, *EndPos* will be the end-of-buffer *when* the function is called. This *EndPos* will advance as the script runs and writes additional information to the buffer!

Functions: (scEmacs interactions)	
ReadFile BN = Ed.ReadFile(FStr, F)	Opens named file (FStr) in scEmacs and returns its buffer number. Will show (attach to a Pane) if #Show flag in F. FStr is required (may include filepath), F is optional. Buffer number is *(Int64)BufferP* from scEmacs. F can be #Show or #NoShow (default).
SaveFile X = Ed.SaveFile(BN, FStr)	Saves/writes specified scEmacs buffer (BN). If FStr (may include filepath) is given, writes BN buffer to it. Returns 1 if successful, 0 otherwise.
GetBuf BN = Ed.GetBuf(NStr, F);	Opens/Creates named buffer (NStr) in scEmacs and returns its number. Will show (attach to a Pane) if #Show flag in F. NStr is required, F is optional. Creates a new *TEMP-n* buffer if NStr is 0, NULL, or "". NStr can also include a path if necessary. F can be #Show or #NoShow (default).
SetCurBuf X = Ed.SetCurBuf(BN, P, F);	Sets *Cur* buffer to BN, sets its cursor position to P, then returns 1. Routines use this *Cur* buffer if their BN parameter is 0. P, if given, defaults to #Byte, but can be #Char if flagged by F. P of -1 means EndPos of buffer (see note above). F can be #Byte (default) or #Char.
ShowBuf X = Ed.ShowBuf(BN, P, F);	Shows BN buffer, sets its cursor position to P, then returns 1. P, if given, defaults to #Byte, but can be #Char if indicated by F. P of -1 means EndPos of buffer (see note above). F can be #Byte (default) or #Char.
ShowBufSel X = Ed.ShowBufSel(BN, S, E, F);	Shows BN buffer, sets cursor *SelRange*, then returns 1. S (Start) specifies beginning of range, defaults to #Byte pos. E (End) specifies end of range, also defaults to #Byte and #Pos. Both can be in #Char units, and E can be a #Len if indicated by F. E of -1 (default value) means EndPos of buffer (see note above). F can be #Byte (default) or #Char + #Pos (default) or #Len. Note: Only buffers *shown* in a Pane can have a SelRange!
Print X = Ed.Print(BN, "%ld", V);	Does C-style *printf* to BN buffer and returns printed byte count. scScript Ints are LONG, use %ld, %lx, etc. Both %f and %lf will work. (Ed.Print uses libffi to call into libc... will crash with bad args!) Prints to stdout (debugging) if BN is -1.

Write X = Ed.Write(BN, U, V, W...);	Writes out concatenation of subsequent IN args to BN buffer. Return total number of bytes written. Write accepts any valid value!
Writeln X = Ed.Writeln(BN, U, V, W...);	Same as Ed.Write, but adds an extra \n at the end. Write and Writeln print to *stdout* (debugging) if BN is -1.
Xfer L = Ed.Xfer(DBN, SBN, S, E, F)	Reads SBN buffer from S to E and writes it to DBN buffer at its Cursor. Returns number of bytes transferred. S (Start) specifies beginning of range, defaults to #Byte pos. E (End) specifies end of range, also defaults to #Byte and #Pos. Both can be in #Char units, and E can be a #Len if indicated by F. E of -1 (default value) means *EndPos* of SBN buffer. F can be #Byte (default) or #Char + #Pos (default) or #Len.
ScanN E = Ed.ScanN(BN, P, F)(N);	Reads number N (Int or Flt) from buffer BN at byte pos P. Returns E (EndPos of N) if successful, or 0. Can read +/-, Decimal, Octal, and Hex. (Flt in Dec/Hex only). F defaults to #Sep, use #NoSep to read without separators.
Del X = Ed.Del(BN, S, E, F);	Deletes specified range from BN buffer and returns deleted bytecount. S (Start) specifies beginning of range, defaults to #Byte pos. E (End) specifies end of range, also defaults to #Byte and #Pos. Both can be in #Char units, and E can be a #Len if indicated by F. E of -1 (default value) means EndPos of buffer (see note above). Ed.Del() will zap the entire *current* contents of *Cur* (or *Output*)! F can be #Byte (default) or #Char + #Pos (default) or #Len.
Read Str = Ed.Read(BN, S, E, F);	Copies the specified range from BN buffer into a Str and returns it. S (Start) specifies beginning of range, defaults to #Byte pos. E (End) specifies end of range, also defaults to #Byte and #Pos. Both can be in #Char units, and E can be a #Len if indicated by F. E of -1 (default value) means EndPos of buffer (see note above). Ed.Read() will copy the entire contents of *Cur* (or *Output*)! (F can be #Byte/#Char + #Pos/#Len.)
SetPos X = Ed.SetPos(BN, P, F);	Sets Cursor Pos in buffer BN and returns 1. P, if given, defaults to #Byte, but can be #Char if indicated by F. P of -1 means EndPos of buffer (see note above). (F can be #Byte/#Char.)
BLen X = Ed.BLen(BN, S, E, F);	Returns byte len of specified range in BN buffer. S (Start) specifies beginning of range, defaults to #Byte pos. E (End) specifies end of range, also defaults to #Byte and #Pos. Both can be in #Char units, and E can be a #Len if indicated by F. E of -1 (default value) means EndPos of buffer (see note above). (F can be #Byte/#Char + #Pos/#Len, but #Byte is silly here.)
CLen X = Ed.CLen(BN, S, E, F);	Returns char len of specified range in BN buffer. S (Start) specifies beginning of range, defaults to #Byte pos. E (End) specifies end of range, also defaults to #Byte and #Pos. Both can be in #Char units, and E can be a #Len if indicated by F. E of -1 (default value) means EndPos of buffer (see note above). (F can be #Byte/#Char + #Pos/#Len, but #Char is silly here.)
BPos X = Ed.BPos(BN, P, F);	Returns byte pos of specified location in BN buffer. P, if given, defaults to #Byte, but can be #Char if indicated by F. P of -1 means EndPos of buffer (see note above). P of 0 (default value) means *cur CursorPos* of buffer. (F can be #Byte/#Char, but #Byte is silly except if P is 0.)
CPos X = Ed.CPos(BN, P, F);	Returns char pos of specified location in BN buffer. P, if given, defaults to #Byte, but can be #Char if indicated by F. P of -1 means EndPos of buffer (see note above). P of 0 (default value) means *cur CursorPos* of buffer. (F can be #Byte/#Char, but #Char is silly except if P is 0.)
Sel X = Ed.Sel(BN, F)(S, E);	Sets S and E to current *SelRange* in BN buffer and returns 1. If no sel, sets S and E to a 0-len interval at *CursorPos* and returns 0. (F indicates desired #Byte/#Char + #Pos/#Len for S and E.)
Echo X = Ed.Echo(U, V, W...);	Writes out concatenation of all IN args to scEmacs Echo line. Return total number of bytes written.

QueryMC X = Ed.QueryMC(PStr, CStr);	Prompts user for multiple-choice selection in scEmacs Echo line. (*Pauses* script until user enters valid choice or aborts.) Required PStr should present prompt *and* response chars. Required CStr contains *lower-case* response chars in order. Will return 0 if aborted (^G) then M-x script-resume. Will return 1..N index for valid selections from CStr. Example PStr: "Copy selection? [YN.!]:> " and CStr: "yn.!".
Defs:	
Unit Flags	#Byte (default) or #Char
Count Flags	#Pos (default) or #Len
Action Flags	#Show (default, usually) or #NoShow (leave hidden)
Number Flags	#Sep (default) to accept thousands separator, #NoSep if not. (Uses decimal and separator char defined in sc Params!)

Script Package

The *Script* package is used to load and execute sub-scripts from a running script. These were initially designed for system testing and validation, but seem to have acquired a life of their own. Since a sub-script can in turn load and execute other scripts, there can be a hierarchy of running scripts at a given time. This package also has a few helper functions for traversing the script hierarchy and getting parameters, such as `Id` and `Level`.

Note: `Level` indicates the position of the script in the running hierarchy, it is meaningless for a non running script.

Functions:	
Self Sc = Script.Self();	Returns the current running Script (obviously to itself).
Sup SupSc = Script.Sup(Sc);	Returns the superior script of the (optional) Sc script, Int 0 if none. If no Sc, then the superior of the currently running script. Note: Only the current script can have a superior.
Id X = Script.Id(Sc);	Returns id number of the (optional) Sc script. If no Sc, then the id of the currently running script.
Level X = Script.Level(Sc);	Returns level number of the (optional) Sc script. If no Sc, then the level of the currently running script.
Load Sc = Script.Load(FStr);	Read and load the specified script file (FStr) into a var. FStr (required) may include path specifications for .sc or .bsc script. Latter are immediately loaded while former are compiled first.
Exec X = Script.Exec(Sc);	Executes the script loaded into the required Sc var and returns result. (Most scripts end without an explicit Return, so the result is 0).

Note: `Script.Exec` will happily execute its own script (from `Script.Self()`) or even a superior one (from `Script.Sup(Sc)`). In fact, it *clones* the running script, behind the scenes, so it can execute it, without forming a *circular* superior chain, for the very rare circumstances where this is required. But a script executing itself or a superior is usually a *very bad* idea and a good way to get super confused!

Math Package

The Math package in scScript, as well as most arithmetic operators, simply fall through to equivalent `libc` functions and operators. scScript `Ints` and `Flts` are C language `Int64` and `double floats` respectively. Therefore, scScript is subject to the same numerical limitations as C and provides the same `NaN` and `Inf` symbols and semantics as C.

Functions: (Calls into libc functions!)	
Abs R = Math.Abs(F);	Returns absolute value of F. (Will convert arg to Flt if Int.) Note: F, R, and M are Flt, X and S are Int.
Min R = Math.Min(F1, F2);	Returns minimum of F1 and F2. (Will convert F1 and F2 to Flt if Int.)
Max R = Math.Max(F1, F2);	Returns maximum of F1 and F2. (Will convert F1 and F2 to Flt if Int.)
FEq (Fuzzy/Float Eq) X = Math.FEq(F1, F2); X = Math.FEq(F1, F2, M);	Returns 1 if F1 and F2 are very close (0 if not), within margin M. M (optional) is calculated internally as multiple of DBL_EPSILON, but if given M will be *hard coded*. M of 0 or 0.0 will take all *fuzz* out of comparison, same as == operation!
Ln R = Math.Ln(F);	Returns natural logarithm (base e) of F.
Log R = Math.Log(F);	Returns common logarithm (base 10) of F.
Power R = Math.Power(F1, F2);	Returns F1 raised to the power of F2.
Exp R = Math.Exp(F);	Returns e raised to the power of F.
Sqrt R = Math.Sqrt(F);	Returns square root of F.
Floor R = Math.Floor(F); R = Math.Floor(F, S);	Returns the nearest flt not greater than F. S (Scale) power of 10, if given (Int or Flt) will apply before op. S of +3 means go to .001 precision, -3 means go to 1000's.
Ceiling R = Math.Ceiling(F); R = Math.Ceiling(F, S);	Returns the nearest flt not less than F. S (Scale) power of 10, if given (Int or Flt) will apply before op.
Round R = Math.Round(F); R = Math.Round(F, S);	Returns the nearest flt, rounding away from 0 in halfway cases. S (Scale) power of 10, if given (Int or Flt) will apply before op.
Cos R = Math.Cos(F);	Returns cosine of F. Note: All trigonometric angles (F) are expressed in radians.
ACos F = Math.ACos(R);	Returns arc cosine of R.
Cosh R = Math.Cosh(F);	Returns hyperbolic cosine of F.
Sin R = Math.Sin(F);	Returns sine of F.
ASin F = Math.ASin(R);	Returns arc sine of R.
Sinh R = Math.Sinh(F);	Returns hyperbolic sine of F.
Tan R = Math.Tan(F);	Returns tangent of F.
ATan F = Math.ATan(R);	Returns arc tangent of R.

Tanh R = Math.Tanh(F);	Returns hyperbolic tangent of F.
Defs: (Constants courtesy of libc!)	
#Pi	3.1415926535897931 (Flts are generally not accurate beyond 16 places.)
#E (Ok, looks weird, #e then!)	2.7182818284590451
#LogE (#Loge?)	0.4342944819032518
#Ln2	0.6931471805599453
#Ln10	2.3025850929940459
#Sqrt2	1.4142135623730951
#Int_Max	9,223,372,036,854,775,807 (INT_MAX in C) (2^63 - 1)
#Int_Min	-9,223,372,036,854,775,808 (INT_MIN in C) (- 2^63)
#Flt_Max	Approx. 1.7e+308 (DBL_Max in C)
#Flt_Min	Approx. -1.7e+308 (-DBL_Max in C)
#Flt_Small	Approx. 2.22e-308 (DBL_Min in C)
#Flt_Epsilon	Approx. 2.22e-16 or 0x01p-52 (DBL_EPSILON in C)
#NaN	Defined as (0.0/0.0) but prints out as *-nan* from libc. Importantly, (#Nan != #Nan) but (#Nan === #Nan) !!!
#Pos_Inf	Defined as (1.0/0.0). Prints out as *inf* from libc.
#Neg_Inf	Defined as (-1.0/0.0). Prints out as *-inf* from libc.

Log Package

scScript provides a simple and convenient logging mechanism. A running script can open a single *log* file, create time-stamped (`#Zulu` or `#Local` time) entries as necessary, then close the log when done. There can only be a *single* log file at a given time and any sub-scripts that log entries will automatically get redirected to the *same* log file. While the sub-script will *seem* to open and close its own log file normally, these actions will just add entries to the superior script's file, leaving it intact and operational without the sub-script being any the wiser. (The superior script sets the operational flags, sub-script flags are ignored.)

The default name for the log file is simply `Log.txt` and it will be placed in the same directory as the (`.sc` or `.bsc`) script itself. But the script may provide a new filename or path for the log file. It can also specify the `#Unique` flag to append an additional time/date string to the filename, rendering it different and distinct from all other log files that may exist in the same directory.

Since a script is typically run many times, the default behavior is to re-use (`#Use` flag) the existing log file, if found (this can only work with `#Common` naming), and simply add entries to it. But it can zap the old file and create a fresh new one each time if the `#Fresh` flag is set.

While the log is *conceptually* opened (`Log.Open`) just once, the default entry mode is to close the actual data file (and flush it) after each entry. But the `#KeepOpen` flag will direct scScript to keep the data file open for the duration of the executing script, running faster at the expense of fault-tolerance. The default behavior is also to create a new *header* in the log file each time it is *conceptually* opened (not for every entry). But the `#NoHead` flag can be used if a header is not required.

In the case of long-running scripts, the user may remove the log file (assuming it is not #KeepOpen) without interrupting the script. Should the script thus find its log missing mid-stream, the default behavior (#Remake) is to create a new data file and continue logging. But this can be altered with the #Require flag, forcing the script to terminate with an error (as a failsafe mechanism).

It is generally good form for any script that opens a log to create one or more initial entries announcing what the script is and what it does. This will make the stand-alone log files more readable, especially if loading multiple sub-scripts. For example:

```
Log.Open();
Log.Entry("Widget HQ Simulation: V2.0 with CoR");
      .... normal operations and logging...
Log.Close();
```

Functions:	
Open X = Log.Open(FStr, F);	Open *the* Log file and return 1 if successful, 0 if fail. If already opened by superior: succeed but record entry for new open. If already opened by this script: fail with error. FStr can be missing, "", path, filename, or full path+filename. Flags (F) are optional.
Close X = Log.Close();	Closes the open Log. Will return 1 if it *was* open, 0 otherwise. If the log was opened by a superior: do not close, just record an entry for attempt to close and return 1.
Entry X = Log.Entry(U, V, W...);	Strings args together (like Writeln) and timestamps to create an entry. If no args, makes a "No Entry" log entry. Will open the Log file if not already open. Returns byte count written.
EntryE X = Log.EntryE(U, V, W...);	Creates an *extended* entry. Follows previous entry *without* a timestamp. Otherwise works exactly like Log.Entry.
Defs:	
Time	#Zulu (default) or #Local Use global (UTC) or local time for Log.Entry timestamp.
Naming	#Common (default) or #Unique Use given/default filename as is or append UTC to make filename #Unique.
Creation	#Use (default) or #Fresh Create file if missing, but use existing or replace with fresh one.
Entry	#OpenClose (default, safer) or #KeepOpen (faster) Re-open/close on each entry or keep open until script ends.
Header	#Header (default) or #NoHead Write a header with each explicit Log.Open or do not.
Mid-stream	#Remake (default) or #Require Create Log file if missing mid-stream or just fail!

Pat Package

The *Pat* package implements a comprehensive and powerful text matching system. The table below provides a *cursory* summary of its pattern definition and run functions. There is a *great* deal more to Pattern Matching, details along with many examples are given in a later section.

Pat Definition:	
Seq P = Pat.Seq(args...);	Returns pat to match concatenated *sequence* of args (conjunction). Args can be strings, other Pats, flags, and *stash* directives.
Alt P = Pat.Alt(args...);	Returns Pat to match one of the *alternative* args (disjunction). Defaults to *selfish* but can be made *considerate*.
Span P = Pat.Span(args...);	Returns pat to match *span* from StartArg all the way to to EndArg, ignoring everything in between. Defaults to *minimizing* and *selfish*.
Rep P = Pat.Rep(args...);	Returns pat to match the specified (n to m) *repetitions* of its arg(s). Defaults to *minimizing* but *considerate*.
Iter P = Pat.Iter(args...);	Returns special Rep Pat to match exactly n times, special form of Rep.
Char P = Pat.Char(args...);	Returns pat to match the given range of *characters*. Stores *inclusion* set of chars, many sets are predefined.
Any P = Pat.Any(args...);	Returns pat to match *any* character. More efficient version of *Pat.Char(Pat.#All)*!
Go P = Pat.Go(args...);	Returns pat to advance match process (*goto*) to new data location.
At P = Pat.At(args...);	Returns pat that succeeds *only* if *at* specified location!
Pat Match:	
Match Pat.Match(args...)(args...);	Match the given pattern against data in scEmacs buffer. Takes many options and parameters, supports multiple callbacks. Provides rich feedback/debugging displays with compile flags.

Programmers generally view a text pattern search or transformation as a simple character-based linear (left to right) sequential process. Once a basic outline is designed, it is elaborated and flushed out, considering alternatives, exceptions, and other subtle points, much the same way all algorithms and code are conceived and created. But scripting languages typically require mapping from the simple sequential process to a complex *representational* grammar (RegEx, PEG, etc.) which the code then *automatically* interprets/executes at runtime.

Such grammars usually save keystrokes, but at the substantial cost of:

- Being generally write-only and very hard to read/maintain,
- Requiring knowledge of an arcane grammar and its cryptic symbols,
- Knowing how the program will interpret/execute the grammar,
- Awareness of non-obvious (hidden) tradeoffs in execution.

As a C programmer, I prefer to directly *code* a simple match, without resorting to magic runes and occult incantations. Ideally, the program understands and executes the pattern match instructions while taking care of low-level details. Scripting a pattern match should not be any harder than writing an arithmetic function.

To this end, scScript introduces a new, more procedural, *Pattern* (Pat) paradigm, drawing inspiration from SNOBOL as well as Parse Expression patterns. Simple Pat objects are first created, usually assigned to variables, then *matched* against text in editor buffers. This can be as simple as:

```
PHello = Pat.Seq("Hello", " ", "world");
Pat.Match(DataBuffN, PHello, Pat.#Once);
```

A minimal lexicon of Pat types, such as Seq, Alt, Rep, and Char, are predefined. But this simple operational model can easily accommodate more types as well as specialized and custom functionality to suit client applications. Importantly, text strings used to define Pat templates are just that, there are no additional *in-band* magic symbols and escape chars to shield them.

A Pat typically refers to strings, other Pats, and even *callback* script functions (closures) that should be called during a match. Once created, the Pat is simply stored in a variable, passed around, and used/re-used by many other Pats or matching functions.

Better yet, the matching process is *re-entrant*, allowing functional separation of different tasks. It is easy for a callback invoked from one Pat to grab data and match other Pats before returning results for the first Pat. For example, one can find paragraphs first, then check them for repeat words in a separate nested task, instead of coding one pattern to do everything at once!

> Nevertheless, pattern matching is a *complex* process and requires a good deal of thought (and alas learning). The representational paradigm, no matter how elegant or simple, cannot eliminate this complexity. It is still all too easy to write a naive or over-simplified match and get shocked when it fails!

__VV (Verify/Validate) Package

Yes, there is a secret package for internal QA testing! Please refer to source code for details.

Advanced Features

Unpack Operator: @

While arrays obtained from `Sys.GetArgs` *pack* the args passed to the function call, the `@` operator can be used to *unpack* them into another function call.

So if `TheFunc` is called with:

```
TheFunc(MyVar, "This", 3.14, "That", 2.718)
       (OtherVar, MyArr[2], MyD.Size, MyD.Weight);
```

and it later calls:

```
Sys.GetArgs()(InArr, InCount, OutArr, OutCount);
OtherFunc(1, @OutArr);
AnotherFunc(X, Y, @InArr)(Z, @OutArr);
```

The call to `OtherFunc` will be the same as:

```
OtherFunc(1, OtherVar, MyArr[2], MyD.Size, MyD.Weight);
```

Similarly the call to `AnotherFunc` will appear as:

```
AnotherFunc(X, Y,  MyVar, "This", 3.14, "That", 2.718)
           (Z, OtherVar, MyArr[2], MyD.Size, MyD.Weight);
```

> `Sys.GetArgs` will pack its arrays (`InArr` and `OutArr`) with the *actual* variables (parameters) seen inside the caller function (same storage). So setting `InArr[0]` to `Math.#Pi` will set the named first *in* parameter (`MyVar`) to pi. When unpacking such an array into another function's *out* list, the very same storage locations are passed to the function. Of course, the *in* list for the other function is *pass-by-value*, so it only copies the *values*, and allocates its own local storage for them. `GetArgs` uses the same storage, but *in* lists only accept values!

It is also possible to *manually* construct arrays and unpack them into function calls at runtime. This is particularly easy for the in list, as the array only needs the values. But things get slightly complicated on the *out* side, as the called function will be using the very *same* storage locations as the array elts. It is obviously possible to create and initialize a new array. But elts in such a new array *can* be set to use the same storage as existing variables using the Alias (`=&`) operator, outlined in the next section.

scScript can also unpack from a dict. The function being called will see the args in the same order the pairs were initially entered into the dictionary. This is the *opposite* order to what is actually stored in a dictionary (they are LIFO not FIFO) and reverse of how `MapK` will traverse the dict, but scScript adjusts for this. However, it is *not* possible to place a pair on the left of an Alias (`=&`), as the pair needs more data (pertaining to its own dict) than a regular variable or array elt. So it would be impossible to manually construct a dict that can unpack *existing* variables into an *out* list.

Alias Operator: =&

The alias operator was originally designed to assign a *local* name to a global variable:

```
Var MyLocal =& SomeGlobal;
```

Bytecodes represent variables with a single-byte index, so are limited to only 254 available local variables (`0x00` indicates a value on the stack while `0xFF` signifies a *reference* on the stack). But scScript supports up to 65K globals, so bytecodes cannot directly manipulate global variables. Instead, the compiler sets aside a fixed number (currently 7) of implicit local variables as *registers* and writes bytecodes to *set* them (at runtime) to the same storage as the global variable. This way bytecodes can directly access registers as a proxy for globals. But this automatic register *aliasing* is often sub-optimal, especially in loops and with goto. So `=&` was devised as a simple optimization to allow the script writer to explicitly *localize* global variables in performance-critical functions, eliminating automatic bytecode generation to set/reset/recycle the few available registers.

The `=&` operator was later expanded to alias *from* (right side) array and dict elements too, saving the effort of repeated indexing and dict lookup. Once `CurElt` is aliased to `MyArr[12]` or `MyD.Weight`, it will have exactly the *same storage* and can be read, set, and otherwise manipulated just like the array elt or dict pair but *without* the indexing and key lookup. This is a significant step in runtime efficiency and programmer convenience.

> Note: In the above example, if entry `12` is missing from `MyArr` or if `Weight` is missing from `MyD` when aliased, they will be automatically created first!

Finally, the alias operator can also be used to *pack* an *out* array just as `Sys.GetArgs` would. This special alias array will get the *same* elt storage as the target vars (or array elts and dict pairs) and can be directly *unpacked* into the *out* list of another call.

So the sequence:

```
MyArr = Array(3);
MyArr[0] =& OtherVar;
MyArr[1] =& MyD.Size;
MyArr[2] =& MyD.Weight;
SomeFunction(3)(@MyArr);
```

Is the same as calling `SomeFunction` with:

```
SomeFunction(3)(OtherVar, MyD.Size, MyD.Weight);
```

The args passed to a function call are usually set at compile time. But using an array along with the unpack operator allows scScript to set the arg list of a function call at runtime.

As powerful as the alias operator is, it has two, somewhat obscure, limitations:

1. The left-side cannot be a global. Globals may already be automatically aliased (set) to a *register*, and the compiler relies on the validity of these registers for code correctness. If the global variable itself assumes another storage location, its *pre-existing* register will have the wrong storage. While the compiler may find ways to invalidate such registers, nothing can be done if there are *yielded* coroutines that have already *set* the register!

2. The left-side cannot be a dict element. Array elts are simple and use the same storage format as variables. But pairs in dicts have much more information. Therefore, a pair cannot get the same storage as a simple variable. (But a simple variable *can* happily use the larger storage of a dict pair.) So pairs are allowed on the right side, but not the left side, of an alias.

The main body of a script runs as an implicit (nameless) *main* function. This function, by virtue of its top position, *cannot* direcly declare any local vars to alias critical globals (everything it declares becomes another global). While a current limitation, this can be remedied by introducing a {} block in the main function for the express purpose of introducing one or more *local* vars and aliasing globals.

Non-Local Returns

scScript supports an `RC` data structure that represents the *RunContext* of a function (or think of it as *Remote Control* for the function). `MyFunc` can get access to its own `RC` using the `GetRC`: (in `sc` pkg)

```
Var TheRC = GetRC();
```

The hypothetical `MyFunc` can then hand `TheRC` over to a function (`Sub1`) that *it* calls which can hand it to a function (`Sub2`) that *it* calls… on and on to `SubN`. At that point `SubN` can execute:

```
Return X OutOf TheRC;
```

This will immediately return the value of `X` to the *caller* of `MyFunc`, exactly as if `MyFunc` *itself* executed:

```
Return X;
```

This capability has two common uses:

1. Error-return: Lower functions in a call chain can simply exit with an error code, without every higher function having to test, catch, and propagate error codes.

2. Recursion: The lower function in a deep traversal can find the solution and return it without having to pop repeatedly back up the chain.

scScript also optimizes tail-recursion, so the lower function in a tail-recursive chain can implicitly return from up the chain (but *only* if the function does *not* call `GetRC`). In these cases, scScript effectively *eliminates* the recursion itself, so no extra (stack frame) memory is used.

> In early versions, the `OutOf` keyword was given the more succinct and logical name `From`. But it soon because obvious that `From` was *not* a good keyword as it is too common a variable name! Also, the original name for `RC`, perhaps still visible in source codes, was confusingly *RunCall* rather than *RunContext*.

CoRoutines

There is a good deal of theoretical literature on coroutines and their use as light-weight threads. The most common examples tend to depict a server architecture of some kind with one or more *persistent* clients or sources of data. So perhaps a better name would be *PermRoutines*.

Technically, scScript provides *stackfull asymmetrical (full) coroutines*. In a nutshell, this mechanism allows a function to *persist* after its initial call and execution. Therefore, the persistent function can keep its local information (variables and execution state) after returning (initially with *submit*, subsequently with *yield*) then get re-invoked (repeatedly!) from where it left off. Such persistent functions are normally implemented in C as separate threads, but a reasonable facsimile can be emulated using *static* local variables to save state. scScript cannot declare static locals, although it can have private vars in closures. Then again, scScript can make any function persist!

Both the persistent routine (`P`) itself and its caller/master (`M`) should be aware of the situation, so a small initial handshake is required:

- `M` calls `P` like any other function, with the proper *in*/*out* arg list.
- `P` initializes and does a `Submit()` to return *its* `RC` back to `M`.
- `M` stores the `RC` submitted by `P` in a variable (`P_RC`).
- `M` performs its own work...

From now on, `M` can re-invoke `P` with `Resume(P_RC)`:

- `P` resumes execution where it left off (after `Submit`).
- `P` completes its task/computation then does: `Yield Val`.
- `M` takes the yielded `Val` (returned from its call to `Resume`) and performs its own work…
- Later, `M` re-invokes `P` with `Resume(P_RC)` again!
- `P` resumes execution where it left off (after `Yield`) and the *cycle* continues…

The *cycle* terminates when:

- `M` does a `Release(P_RC)` *or*
- `P` does a `Return` instead of a `Yield`.

> If `P` terminates with a `Return`, `M` must *know* not to invoke it again with another `Resume` (as it will fail). So it is generally recommended that persistent functions `Yield` non-zero values but `Return 0` to signal termination. Of course, `P` can establish a much richer exchange with `M` using its *out* params. `Yield` (just like `Return`) can also exit *non-locally* using `OutOf V` where `V` contains the `RC` passed down from `P`. (`P` can gets its own `RC` using `GetRC()`, this will be exactly the same `RC` that `Submit` passed to `M`.)
>
> `Yield`, except in specialized cases, should *always* be called in a loop! It is a very common mistake to forget the loop, thus getting only a single *cycle* out of a persistent function. (Yes, I have fallen in this hole myself... multiple times!)

The following is a somewhat traditional, if playful, example, written with and without coroutines.

```
// Widget factory... no coroutines! (CR1.sc)

Function OrderA()()
{
    Var    I = Sys.Rand(0, 10);

    Writeln("    Supplier: Sending ", I, " part A to factory");
    Return I;
}

Function OrderB()()
{ … }  // Just like Order A

Function OrderC()()
{ … }  // Just like Order A

Var        A, B, C;
Var        Record;

Function Factory()()
{
    Var    Made;

    Writeln("Factory: Part Shelf [", A, " ", B, " ", C, "]");
    // Order Widget parts
    If (!A) Writeln("  Order part A"), A = OrderA();
    If (!B) Writeln("  Order part B"), B = OrderB();
    If (!C) Writeln("  Order part C"), C = OrderC();

    Write("\n  Assembling  ");
    While (A && B && C) {
          Write("*");
          Made += 1;
          --A, --B, --C;
    }

    If (Made)
          Writeln("  Shipping ", Made, " Widgets to HQ!");
    Else
          Writeln("  Failed!!  Nothing to ship...");

    If (Made > Record) {
          Record = Made;
          Write("  The factory is celebrating a new record!\n\n");
    }

    return Made;
}

Function Sales()()
{
    Var I = Sys.Rand(0, 20);

    Write("Sales: Sold ", I, " Widgets!!\n\n");
    return I;
}

Function WidgetHQ()()
{
    Var        Sold;
    Var        Stored;
    Var        Cycles;

    Writeln("ACME Widgets Inc.");
    Write("  Starting Operation...\n\n");

    Do {
          Sold = Sales();

          while (Sold > Stored) {
              Stored += Factory();
              Write("HQ Inventory: ", Stored, " Widgets.\n\n");
          }

          Stored -= Sold;
          Write("** Fulfilled to customer: ", Sold, "!!\n\n");
    } While ((Cycles++ < 10) || Stored);
}

WidgetHQ();
```

```
// Widget factory... WITH coroutines! (CR2.sc)

Function OrderA()()
{
    Var    I = Sys.Rand(0, 10);

    Writeln("    Supplier: Sending ", I, " part A to factory");
    Return I;
}

Function OrderB()()
{ … }  // Just like Order A

Function OrderC()()
{ … }  // Just like Order A

Function Factory()()
{
    Var        Made;
    Var        A, B, C;
    Var        Record;

    Submit();
    while (1) {
          Made = 0;
          Writeln("Factory:");
          // Order Widget parts
          If (!A) Writeln("  Order part A"), A = OrderA();
          If (!B) Writeln("  Order part B"), B = OrderB();
          If (!C) Writeln("  Order part C"), C = OrderC();

          Write("\n  Assembling  ");
          While (A && B && C) {
              Write("*");
              Made += 1;
              --A, --B, --C;
          }

          If (Made)
              Writeln("  Shipping ", Made, " Widgets to HQ!");
          Else
              Writeln("  Failed!!  Nothing to ship...");

          If (Made > Record) {
              Record = Made;
              Write("  The factory is celebrating a new record!\n\n");
          }

          Yield Made;
    }
}

Function Sales()()
{
    Var I = Sys.Rand(0, 20);

    Write("Sales: Sold ", I, " Widgets!!\n\n");
    Return I;
}

Function WidgetHQ()()
{
    Var        Sold;
    Var        Stored;
    Var        Cycles;
    Var        FactoryRC;

    Writeln("ACME Widgets Inc.");
    Write("  Starting Operation...\n\n");

    FactoryRC = Factory();
          Do {
              Sold = Sales();

              While (Sold > Stored) {
                    Stored += Resume(FactoryRC);
                    Write("HQ Inventory: ", Stored, " Widgets.\n\n");
              }

              Stored -= Sold;
              Write("** Fulfilled to customer: ", Sold, "!!\n\n");
          } While ((Cycles++ < 10) || Stored);
    Release(FactoryRC);
}

WidgetHQ();
```

`WidgetHQ` is the main routine in both cases. It calls the `Sales` department, which like a typical internet business, sells widgets it does not have. Then the business scrambles to get the widgets produced at the `Factory` and sent to `HQ` for shipping to customers. But the `Factory` itself has to order parts `A`, `B`, and `C`

from unreliable suppliers who, perhaps suffering from a language barrier, seem to ship a random number (if any) each time. So the `Factory` does its best and builds as many widgets as it can each cycle, bringing joy to this drudgery by celebrating each time a new production record is achieved. All of this gets reported in the `*Output*` window when the script runs.

There are a few slight differences in the two examples. In the coroutine version, `WidgetHQ` nests the main `Do...While` inner loop between a call to `Factory()`, where it stashes the returned `RC`, and the `Release` of `RC`. IButn turn, the persistent `Factory` does an initial `Submit` to kick things off, then performs a `Yield` instead of a `Return`. It also runs in a seemingly endless `While(1)` loop, where it resets the `Made` count each time. Finally, `WidgetHQ` (the *master*) calls `Release` to terminate the `Factory`.

Importantly, the persistent `Factory` routine can store all its own variables and does not need to use globals. This would be a great advantage if the code was more complex and had many more lines, variables, and subroutines of its own. This persistent `Factory` could itself be the *master* for different persistent emulations of remote supplier. Coroutines (or threads) are often used for interfacing to external mechanisms where they maintain complex state while awaiting actual events.

```
Function Factory()()
{
    Var        Made;
    Var        A, B, C;
    Var        Record;
    Var        FRC = GetRC();

    Function Build()()
    {
           Made = 0;

           Writeln("Factory:");
           // Order Widget parts
           If (!A) Writeln("  Order part A"), A = OrderA();
           If (!B) Writeln("  Order part B"), B = OrderB();
           If (!C) Writeln("  Order part C"), C = OrderC();

           Write("\n  Assembling  ");
           While (A && B && C) {
               Write("*");
               Made += 1;
               --A, --B, --C;
           }

           If (Made)
               Writeln("  Shipping ", Made, " Widgets to HQ!");
           Else
               Writeln("  Failed!!  Nothing to ship...");

           If (Made > Record) {
               Record = Made;
               Write("  The factory is celebrating a new record!\n\n");
           }

           Yield Made OutOf FRC;
    }

    Submit();
    while (1) Build();
}
```

As previously mentioned, a non-local `Return` or `Yield` is just a matter of passing the `RC` down a *call chain* and using the `OutOf` modifier. In the above example (`CR3.sc`), `Factory` uses a separate `Build` function to illustrate the point.

Private Variables

scScript only provides local and global variables. But thanks to the magic of closures, local variables can persist outside their original scope. Better yet, multiple closures can have simultaneous private access to the same local variables, making them an otherwise *hidden* shared state or comm channel.

```
// Private Variable!  (PV.sc)

Var FuncA, FuncB;
Var I;

Function MakeFunc()()
{
    Var PriVar = 1;

    FuncA = Function()()
        Writeln("FuncA --> ", PriVar *= 2);

    FuncB = Function()()
        Writeln("FuncB --> ", PriVar += 1);
}

MakeFunc();
For (I = 0; I < 8; I++) {
    If (Sys.Rand() & 0x01)
        FuncA();
    Else
        FuncB();
}
```

```
*Output*


FuncA --> 2
FuncA --> 4
FuncB --> 5
FuncB --> 6
FuncA --> 12
FuncA --> 24
FuncB --> 25
FuncA --> 50



*******************************************
   SCRIPT RETURN -->  0
*******************************************
```

In the above example two functions, `FuncA` and `FuncB`, are defined within `MakeFunc`. When `MakeFunc` is called in the script, it creates two closures for them and assigns them to their respective named global variables. But both closures *capture* `PriVar`, keeping the storage location alive for a local variable that existed briefly only while `MakeFunc` ran. So while `FuncA` and `FuncB` remain, they trivially access the storage location for `PriVar`. But no other code has access to `PriVar` or its storage, as `MakeFunc` no longer exists!

If the variables for `FuncA` and `FuncB` get re-assigned, as in:

```
FuncA = FuncB = 0;
```

the two closures will be lost and the storage originally assigned to `PriVar` will (finally) be freed.

> Non accessible storage will be freed immediately if *RefCounting* is in effect (it can be turned off with a compile flag). Otherwise, the system will free them when Mark/Sweep GC is eventually called to clean up memory! (The user can explicitly invoke Mark/Sweep GC with `Sys.GC()`.)

Map Iteration

scScript currently provides two primitives, `MapI` and `MapK`, to map (iterate) over arrays and dicts:

```
MapI(S)(V, I) Operation;
```
or
```
MapI(S)(V, I) {Operations...}
```

`MapI` (Map Index) will perform `Operation(s)` for every single *cell* of `S`, which is usually an array or dict. If the *out* variables `V` and `I` are provided, they will get the value of individual array elt or dict pair and the `0`-based index respectively. New arrays are generally created with missing elt, but `MapI` will still *index* every cell, it will execute `Operation(s)` and set `V` to `Int 0` for empty cells. Similarly, `MapI` will index through the closed hash array for dicts, going down every individual cell, even the ones that do not indicate a pair. Since there is no rhyme nor reason to hash indices, a MapI traversal of a dict will not visit its pairs in any user-discernible order.

```
MapK(S)(V, K) Operation;
```
or
```
MapK(S)(V, K) {Operations…}
```

`MapK` (Map Key) will perform similar to `MapI`, but will *key* only on non-empty cells in arrays and dicts. In the case of arrays, `MapK` executes `Operation(s)` and sets `V` and `K` for each elt. When it comes to dicts, `MapK` traverses the *chain of pairs*, ignoring the hash array completely. If provided, `V` will get the value and `K` the storage key for the pair. Interestingly, dicts are LIFO, so the last pair to be added will be traversed first with `MapK`. In general, while both `MapI` and `MapK` make sense for arrays, but only `MapK` is used for dicts.

Both `MapI` and `MapK` traverse other data, such as numbers or strings, in a *single* step. So the `Operation(s)` in the body of the `Map` will be performed only once and `I` or `K` will be set to `Int 0`.

> MapI (and MapK) will place the *value* of the array elt (or dict pair) in the V variable. They will not *set* V (alias it) to have the same storage as the elt (or pair value). While *aliasing* was initially contemplated for `MapI`, it was soon overruled because the user cannot easily know if an elt is missing or really has an `int 0` value. If the cell was missing, a new temporary variable would have to be created to satisfy the alias, and (confusingly) assigning to it would *not* change the array! (Alternatively, all elts of an array *could* be populated when Mapping, but this would be inefficient; besides, dicts rely on empty cells for their hash storage algorithm.)

While `Map` iterators look like scScript function calls, they are internally implemented as IFIDs, sequences of bytecodes that are added inline with no function setup/call overhead. The bulk of the work is done by efficient low-level `IMap` and `KMap` bytecodes, forming loop centers.

Implementation Details

The following example shows the interaction between `MapI` and coroutines with non-local exits. `PFunc` defines an array, stashes its own `RC`, then `Submits` to its caller, the main script. The script will in turn `Resume PFunc`, which in turn calls `SFunc`, handing it the array and its `RC`. This subordinate function will perform the `MapI`, and `Yield OutOf`

PFunc to the main script. The complexity becomes obvious when the script shows the PRC contents, which now has 3 parts:

- Main *RunRecord:* indicates PC (*ProgramCounter*) 52, the instruction *after* *FCal* to SFunc in PFunc.
- *Resume RunRecord*: indicates PC 47, the instruction immediately after Yield in SFunc. (It is actually a one-byte *No-op*, followed by a *Jmp*!)
- *Stack Copy*: contains 4 elements placed on the run-stack by MapI for its continued operation. These have to be restored when jumping back in.

```
// MAP and CoRoutines!  (CRMap.sc)

Function SFunc(A, RC)()
{
    Var V, I;

    // Will yield mid-MapI
    MapI(A)(V, I) {
        Writeln(I, " --> ", V);
        if (I == 1) Yield V OutOf RC;
    }
}

Function PFunc()()
{
    Var TheArr = ["First", "Second", "Third"];
    Var FRC = GetRC();

    Submit();
    SFunc(TheArr, FRC);
}

Var PRC, Res;

PRC = PFunc();      // Gets RC from PFunc Submit!
Res = Resume(PRC);  // SFunc will Yield mid-MapI...
Show.Vars(Show.#Max)(Res, PRC);

Resume(PRC);        // Will Return from PFunc and end.
```

The following is the *Output* of the above script:

```
0 --> First
1 --> Second
    Var (Ref:1) [Entry 000048 (16 Bytes) on Mem:1 (VEnt) Slab:1]
        --> Str (Ref:4) [Entry 020032 (32 Bytes) on Mem:2 (BEnt) Slab:1]
            --> 6 Bytes [Entry 003376 (24 Bytes) on Mem:3 (SDat) Slab:1]
                "Second"

    Var (Ref:1) [Entry 000032 (16 Bytes) on Mem:1 (VEnt) Slab:1]
        --> RC (Ref:5) [Entry 020416 (32 Bytes) on Mem:2 (BEnt) Slab:1]
            --> Run PC:52 [Entry 008336 (72 Bytes) on Mem:5 (Chnk) Slab:1]
                --> Code:2 (Ref:2) [Entry 000224 (104 Bytes) on Mem:5 (Chnk) Slab:2]
                --> 001 Loc Var (Ref:1) [Entry 000064 (16 Bytes) on Mem:1 (VEnt) Slab:1]
                --> 002 Loc Var (Ref:1) [Entry 000080 (16 Bytes) on Mem:1 (VEnt) Slab:1]
            Resume RunRecord
            --> Run PC:47 [Entry 008408 (88 Bytes) on Mem:5 (Chnk) Slab:1]
                --> Code:1 (Ref:2) [Entry 000120 (104 Bytes) on Mem:5 (Chnk) Slab:2]
                --> 001 In Var (Ref:1) [Entry 000160 (16 Bytes) on Mem:1 (VEnt) Slab:1]
                --> 002 In Var (Ref:1) [Entry 000176 (16 Bytes) on Mem:1 (VEnt) Slab:1]
                --> 003 Loc Var (Ref:2) [Entry 000192 (16 Bytes) on Mem:1 (VEnt) Slab:1]
                --> 004 Loc Var (Ref:2) [Entry 000208 (16 Bytes) on Mem:1 (VEnt) Slab:1]
            Stack Copy
            --> 4 Elts [Entry 004376 (80 Bytes) on Mem:4 (ADat) Slab:1]

2 --> Third

   ******************************************
   SCRIPT RETURN -->  0
   ******************************************
```

Interestingly, the FRC variable in PFunc and PRC returned by Submit are exactly the same RC, as every function has one and only one RC. But *Resume RunRecord* and *Stack Copy* would be NULL (and therefore not shown) if Show.Vars were to be called from PFunc, because it would be running and not (suspended) awaiting resumption.

When the second Resume is called in the main script, control flow returns to SFunc immediately after Yield. The third and final iteration of MapI would call Writeln and SFunc would terminate, returning to PFunc which would also return.

Normally, SFunc would be a *tail call* from PFunc, and scScript would optimize it so SFunc returns directly to the main script. But tail call optimization is *turned off* for coroutines and where GetRC() is called for non-local returns.

Comparing and Counting

The C programming language has familiar == and != operators to compare scalar values as well as pointers. But scScript does not expose pointers, so provides special === and !== operators to compare *storage locations* instead. Given:

```
Var A = "Yes", B = "Yes";
```

All the following will be true:

```
(A == B)
(A == "Yes")
(B == "Yes")
```

because A and B will have the exact same value, "Yes" (an immutable string).

But all the following will be false:

```
(A === B)
(A === "Yes")
(B === "Yes")
```

because A and B are *different* variables with different storage locations. Even more, while the *value* of A is "Yes", the variable A has its very own storage location, which is *not* the same as the literal string "Yes". In that sense, X === Y really tests that X *is* Y. (Recall that X =& Y makes X *become* Y.)

> While === generally tests if both sides have the same storage location, it is smart enough to do a value comparison if one (or both) sides are just constant numbers without a *storage location*. So K === 12 returns 1 if the *value* of K is 12. Importantly, this test also works for NaN, since it does a bit-image comparison and not a numerical one. Therefore K === Math.#Nan is a good test to see if the value of K has left the realm of numbers. (Some languages require an obscure K != K test for this!)

Arrays, unless created with a [] constructor, are created with missing (*unpopulated*) cells. Dicts, by definition, only have a few pairs and are missing a whole infinity of possible key-value pairs. When comparing or using values, scScript will automatically *populate* any missing elts or pairs. It initializes these empty storage cells to contain the value Int 0, so == and other operations get proper values to compare and use. This *auto-populate* may also involve expanding the array or dict as necessary.

So a simple test, such as

```
(MyDict.George == 0) or
(MyArr[12] == 0)
```

will in fact add a pair to `MyDict` and an elt to `MyArr` if they did not have them before. Therefore an equality test for `MyArr[240000]` will cost an immense amount of memory as `MyArr` is expanded!

But things are different for `===` as it only cares about storage locations, not values. To that end, `===` does *not* populate missing elts or pairs; it just treats them as missing (*null*) locations. But to get the proper semantics, `===` returns `0` if they are missing! So `A[x] === A[y]` is always `0` when *one* or *both* are missing. Interestingly a statement of the form `A[x] == A[y]` will automatically create both missing elts (and succeed as both will be `0`), but these will still have different storage locations, so the `===` test, if subsequently applied, will once again return `0`.

scScript provides two predicates to ascertain the size of an array or dict:

- `SizeOf(A)` returns the number of cells the array or dict *can* currently hold, even if unpopulated. This number will increase if the array or dict are expanded (capacity of the bus).
- `CountOf(A)` returns the number of *populated* elts/pairs currently in the array or dict. This number will increase as new ones are added or auto-populated (how many *in* the bus).

```
// Size and Count test!  (SCTest.sc)

Var A = Array();
Var D = Dict();

A[5] = 0;
D[5] = 12, D["Yes"] = 144, D["No"] = "Yes";

Writeln("A Size ", SizeOf(A));
Writeln("A Count ", CountOf(A));
Writeln("A[5] Count --> ", CountOf(A, 5));
Writeln("A[4] Count --> ", CountOf(A, 4));
Writeln("-----------");

Writeln("D Size ", SizeOf(D));
Writeln("D Count ", CountOf(D));
Writeln("D.Yes Count --> ", CountOf(D, "Yes"));
Writeln("D[4] Count --> ", CountOf(D, 4));
Writeln("-----------");

Writeln("A[5] == A[4] --> ", A[5] == A[4]);
Writeln("A[5] === A[4] --> ", A[5] === A[4]);
Writeln("A[5] === A[5] --> ", A[5] === A[5]);
Writeln("A[5] === A[6] --> ", A[5] === A[6]);
Writeln("A[3] === D[3] --> ", A[3] === D[3]);
Writeln('D["No"] == "Yes" --> ', D["No"] == "Yes");
Writeln('D["No"] === "Yes" --> ', D["No"] === "Yes");
Writeln('"Yes" === "Yes" --> ', "Yes" === "Yes");
Writeln("-----------");

Function Test()(OutA, OutD)
{
    Var   LocA =& OutA;
    Var   LocB = LocA;

    Writeln("OutA === A --> ", OutA === A);
    Writeln("OutA === LocA --> ", OutA === LocA);
    Writeln("LocA === A --> ", LocA === A);
    Writeln("LocB === A --> ", LocB === A);
    Writeln("-----------");

    LocA[9] =& OutD["Yes"];
    Writeln("LocA[9] --> ", LocA[9]);
}

Test()(A, D);
Writeln("A[9] --> ", A[9]);

Writeln("-----------");
Show.Vars(Show.#MAX)(A, D);
```

```
A Size 8
A Count 1
A[5] Count --> 1
A[4] Count --> 0
-----------
D Size 16
D Count 3
D.Yes Count --> 1
D[4] Count --> 0
-----------
A[5] == A[4] --> 1
A[5] === A[4] --> 0
A[5] === A[5] --> 1
A[5] === A[6] --> 0
A[3] === D[3] --> 0
D["No"] == "Yes" --> 1
D["No"] === "Yes" --> 0
"Yes" === "Yes" --> 1
-----------
OutA === A --> 1
OutA === LocA --> 1
LocA === A --> 1
LocB === A --> 0
-----------
LocA[9] --> 144
A[9] --> 144
-----------
    Var (Ref:1) [Entry 000032 (16 Bytes) on Mem:1 (VEnt) Slab:1]
        --> Array (Ref:1) [Entry 000064 (16 Bytes) on Mem:1 (VEnt) Slab:1]
            --> 10 Entries [Entry 009536 (96 Bytes) on Mem:4 (ADat) Slab:1]
                000 ---
                001 ---
                002 ---
                003 ---
                004 ---
                005 Var (Ref:1) [Entry 000000 (16 Bytes) on Mem:1 (VEnt) Slab:1]
                    Int 0
                006 ---
                007 ---
                008 ---
                009 Pair (Ref:2) [Entry 016608 (32 Bytes) on Mem:2 (BEnt) Slab:1]
                    Int 144

    Var (Ref:1) [Entry 000016 (16 Bytes) on Mem:1 (VEnt) Slab:1]
        --> Dict (Ref:1) [Entry 017184 (32 Bytes) on Mem:2 (BEnt) Slab:1]
            --> 16 Entries [Entry 009392 (144 Bytes) on Mem:4 (ADat) Slab:1]
                Filled Dictionary:  3 of 16
                Total Probes:3 (Avg:1.000000)
                - * - - - - - - * - - - - *
                Pair (Ref:1) 0001 Key:"No" Val:"Yes"
                Pair (Ref:2) 0002 Key:"Yes" Val:144
                Pair (Ref:1) 0003 Key:5 Val:12
```

Importantly, the special *dyadic* function `CountOf(TheArr, Index)` or `CountOf(TheDict, Key)` will return `0` if the indexed elt is *not* populated or the indicated pair is *not* in the dict. But it is imperative to

use this *2-arg* format, otherwise `CountOf(TheArr[Index])` will automatically populate `TheArr[Index]` when the argument for `CountOf` is being processed!

The sample code above left, and its output on the right, illustrates many of these points. The `Test()` function above demonstrates the implicit *alias* mechanism of the *out* arg list as well as the explicit storage assignment of the `=&` operator itself. In both cases, variables are assigned pre-existing storage locations belonging to another, a relationship that can be verified using `===`.

Goto

Just like C, scScript supports the use of `goto` operators. While it is easy to turn a structured routine into unreadable spaghetti with indiscriminate use, `goto` jumps, when called for, can also make the code much cleaner and simpler. The C source code for scScript proudly uses very many `goto` operators!

> In case there were any questions: judicious goto jumps are great, camel-case is clearly superior, and object-oriented is a horrible evil.

scScript imposes 2 limitations on its `goto` operators:

1) All jumps must stay in the same routine. As in C, `goto` cannot be used to exit a function or enter another one.

2) `Goto` cannot jump into the body (block) of a `MapI` or `MapK` loop. These iterators place some (currently four) required values on the stack and cannot operate without them. So while a `goto` can jump into the body of a `For` loop and happily generate possibly erroneous results, a jump into a `MapI` block will result in a crash and is prohibited!

   Interestingly, jumps *over* one or more `Map` blocks are perfectly fine. The compiler will also correctly process `goto` jumps *out* of (even multiple nested) `Map` loops. These require popping off the extra stack values. In fact, the compiler initially places a placeholder `No-op` bytecode before *every* goto `Jmp` just in case it needs to add an extra `PopN` later in the *assembly* phase.

Argument Execution Order

scScript supports three operative forms: Functions, IFIDs, and Operators. All three return a value that can be stored/assigned and are able to accept extraneous operands (albeit with the help of comma for operators). It is therefore possible to write contrived, unreadable, and degenerate code where these operands have mutual side-effects.

For example: (*bad* code!)

```
MyArr[I] = MyFunc(A, B, C, I += 3); or
MyDict[K] = GetRC(K = SomeArr[I+=2]);
```

The relative execution order of the (left) assignment arg, expected *in* args, and extraneous *in* args should *not* be relied upon and may in fact differ between the three forms! While execution order can be standardized, the extra cost (in code size, runtime speed, and testing) to harmonize degenerate code across operative forms does not seem worthwhile. A better approach is to avoid writing such *bad* code.

Tail Call Optimization

Each function call in scScript will normally create a new `RunRecord` (same concept as *stack frame* in C) that points to its caller's `RunRecord`. These data structures (allocated on `Chunk` slabs) will hold all local variables, call parameters, and registers. So a recursion that goes `10,000` levels deep will normally require a chain of `10,000` such `RunRecords`.

But it makes no sense to maintain all this if the recursive calls (or *any* call for that matter) is the *very last step;* so no local variable, call parameter, or register will be needed from the caller during the return. To that end, scScript detects *tail calls* and recycles the current `RunRecord` for the next call. So instead of the long chain, and megabytes of RAM used, there will only be a single `RunRecord`!

Tail call optimization can only work if the script function *ends* with:

- `SomeFunc(...);`
- `Return SomeFunc(...);` or
- `X = SomeFunc(...); Return X;`

  Note: `X` must be a *caged* local. It cannot be an *out* parameter, aliased to a global, or stashed in a closure, as those have non-local (*escaped*) side-effects and must be set (by the caller) *after* `SomeFunc` returns. `X` can also not be an array elt, as it *may* escape when the array does (better safe than sorry)!!

scScript is even smart enough to handle the `X = SomeFunc` case above if a `jmp` bytecode intervenes, as tail calls are identified at runtime! The `jmp` could be the result of a hard coded `Goto` or the function call occurring in a conditional (`If/Else`) clause that leads to the end. (scScript also automatically collapses multi-step unconditional hops into a single jump.)

However, the caller's `RunRecord` cannot be eliminated if it obtains an `RC` (whether through `Submit` or `GetRC()`) or if the tail call to `SomeFunc` is followed by `Return; Return Y;` or any other operation!

The `Return;` on its own is simply shorthand for `Return 0;` which is really no different than `Return 42`. This requires the function to push `0` (or `42`) on the stack upon returning from the recursive call. While seemingly trivial, this simple act precludes eliminating the function's `RunRecord` from the run chain! The caller's `RunRecord` can only be eliminated if it does absolutely *nothing* after the call returns.

Just to be clear, the following *will* optimize the call to `MyFunc`, but only as long as `X` is *caged*!

```
Function MyFunc(In1)()
{
    Var X;
    ... blah blah blah....

    if (SomeCondition...) {
            ... Some things...
            X = MyFunc(++In1);
    } else {
            ... Other things...
    }

    return X;
}
```

Pattern Matching

scScript implements a comprehensive and powerful text matching system. The table below summarizes its functions, pre-defined flags, directives, accessors, and templates. Detailed design and operation, along with multiple examples, are documented in the text that follows. Buckle up!

Note: It is easier to just scan over these tables in the first reading; it will all make more sense later.

Pat Functions: (Defines/Creates Pats) Create a Pat first, then Match against data!	
Seq P = Pat.Seq(F1, A1, A2, ..., F2, A10, A11, ..., ... Dir1, Arg1, Dir2, Arg2, ...);	Returns pat to match concatenated *sequence* of args (conjunction). F1, F2, ..., are Flags that affect subsequent args. A1, A2, ..., are args. These can be Strs, pats, or *PFunc* (closures) Dir1, Dir2, ..., are directives of where to stash, etc. Arg1, Arg2, ..., are args for the directives, one per.
Alt P = Pat.Alt(...same as Seq...);	Returns pat to match one of the *alternative* args (disjunction). Defaults to #SELF.
Span P = Pat.Span(F1, SArg, F2, EArg, Dir1, Arg1, Dir2, Arg2, ...);	Returns pat to *span* from SArg (start) all the way to to EArg (end), skipping everything in between. Only accepts flags and 2 string/pat args, followed by stash directives. Defaults to #MIN and #SELF.
Rep P = Pat.Rep(Min, Max, ...same as Seq...);	Returns pat to match the specified *repetitions* of the arg(s). Min/Max can be dynamic values (from PDict) if given strings. Min/Max can also be VFunc callbacks, return value at runtime. Only #CONS/#SELF flags preceding first match arg are honored. Defaults to #MIN and #CONS.
Iter P = Pat.Iter(Cnt, ...same as Seq...);	Returns special Rep Pat, with MIN and MAX both set to Cnt! Will match args exactly Cnt times. Cnt can only be Int, use Rep for other options!
Char P = Pat.Char(F1, A1, A2, ..., F2, A10, A11, ..., ... Dir1, Arg1, Dir2, Arg2, ...);	Returns pat to match the given range of *characters*. Args (A1, A2, etc.) are strings (perhaps pre-def) or other Char pats. Strs list chars, can have ranges ("c-q") too. ('-' first if *not* range) No PFunc and no #Sen/#InSen flags, succeeds if char is in the Pat. Stores *inclusion* set only, #Neg Str/Pats must come *after* #Pos ones. When matched, #S_Cnt is Unicode # for char, #S_Len1 is CharLen. Default is #Smart (À matches if A is in), use #Exact for *exclusion*.
Any P = Pat.Any(F1, F2, ..., Dir1, Arg1, Dir2, Arg2, ...);	Returns pat to match *any* character. More efficient version of Pat.Char(Pat.#All)! Used to *eat* characters during a match. Stashes everything Char does, *except* #S_Cnt will be UTF-8 CharLen.
Go P = Pat.Go(Loc, ...Dir...);	Returns pat to move match process (*goto*) to new data location. Loc can be Int, Str (dynamic), or VFunc. Loc defaults to #Rel and #H (chars), can also be #Abs and/or #V (lines).
At P = Pat.At(Loc, ...Dir...);	Returns pat that succeeds *only* if *at* specified location! Loc can be Int, Str (dynamic), or VFunc. #ABS Loc is from Buf beginning if >= 0, or Buf end if < 0. #REL Loc is from FirstPos if >= 0, or LastPos if < 0. #H deals with Pos, V deals with lines. Does *not* stash any data, but can handle other directives.
Run Functions: (Make and Match)	
Make P = Pat.Make(Type, a1, a2, ...);	Return a pattern (like above), but takes *Type* as arg.

Match Pat.Match(Bn, Pat/Str, Mode/MFunc, StartPos, FirstPos, LastPos, RFunc, GapRFunc) (PatDict, MStartPos, MEndPos);	Match the given pattern against data. BN (Required) BufferNumber from scEmacs. Pat/Str (Required) is what is being matched. Mode (Opt, Default #Once) determines Match operation mode. StartPos (Opt, Default 0 is FirstPos) Pos to start matching. FirstPos (Opt, Default 0) First permissible Pos in Buffer. LastPos (Opt, Default 0 is Buf End) one past last permissible Pos. RFunc (Opt callback) Called if match(s) succeeds (only for #Res Pats). GapRFunc (Opt callback) Called for regions Match scans over. PatDict (Opt) Set to internal PDict, or use PDict if starting with one. MStartPos/MEndPos (Opt) Set to Start/End pos of successful match.
Flag: (Pat flags are sticky, once set, they apply to *all* subsequent args!)	
Match Mode	#Anchor, #Once, #Slide, or #Skip (also #Custom used by MFunc only)
Pat Types	#Seq, #Alt, #Span. #Rep, #Char, #Any, #Go, or #At (Iter is just Rep)
Op Flags (Rep and Span)	#Min (default) or #Max
Mode Flags (Alt, Span, and Rep)	#Self (default) or #Cons (default ONLY for Rep)
Result Flags	#NoRes (default) or #Res (Store in RStack for RFunc)
Case Flags (Sticky!)	#Insen (default) or #Sen
Char Flags (Sticky!)	#Smart (default, root char can match accented version) or #Exact
Polarity Flags (Sticky!)	#Pos (default) or #Neg (See Note ++++ in *Format* section)
Position Flags (Sticky!)	#Rel (default) or #Abs
Direction Flag (Sticky!)	#H (default) or #V
Sense Flag (Sticky!)	#Str (default, string) or #Dyn (dynamic! Key for dict lookup)
Directives: (Do *when* Pat matches.)	
#S_Pos (Key)	Stash match position in PDict under Key (Str).
#S_Str (key)	Stash match string in PDict under Key.
#S_Ord (Key)	Stash match order in PDict under Key.
#S_Cnt (Key)	Stash match count in PDict under Key.
#S_Len1 (Key)	Stash match Len1 in PDict under Key.
#S_Len2 (Key)	Stash match Len2 in PDict under key. (Only used by Span)
#S_Func (SFunc)	Call SFunc (Closure) to store in PDict or affect client data.
#S_Name (NameStr) (Pat Def time)	*Attach* Name to Pat, for display and debugging!
#S_Data (Val) (Pat Def time)	*Attach* Data to Pat, can be accessed by callbacks.
Accessors: (Called from callbacks to get data during match) (No Pat, Name, Data, or Type when called from GapRFunc!)	
GetRange	Returns current match range as (Int: (EndPos << 32) + BegPos).
GetStr	Returns current match string (Str).
GetOrd	Returns current match order (Int).
GetCnt	Returns current match count (Int).
GetLen1	Returns current match Len1 (Int).
GetLen2	Returns current match Len2 (Int). (Only used by Span)
GetPat (Not for GapRFunc)	Returns Pat that is/was matched or 0 if called from GapRFunc.
GetName (Not for GapRFunc)	Returns #S_Name attached to Pat (Str, or 0 if none).
GetData (Not for GapRFunc)	Returns #S_Data attached to Pat (Val, or 0 if none).
GetType (Not for GapRFunc)	Returns Type of Pat that is/was matched or 0 if called from GapRFunc.
GetBuf	Returns Buf Number for this match.
GetBufRange	Returns permissible use range in buffer (Int: (EndPos << 32) + BegPos).

Callback Funcs (Different function types, different args!)	
MFunc MFunc(PDict)(NextPos, SPos, EPos) {...}	*Match* Func (Called by Match after success) Decides whether to do *another* match, and from where! Returns: #Done to stop the match. #Custom, to continue match from NextPos! #Skip, to continue match after EPos. #Slide, to continue match from SPos+1 (will adjust for UTF-8). SPos/EPos are start/end of current match.
VFunc VFunc(PDict, VOrd)() {...}	*Value* Func (Provided in Pat definition to set required Op values) Return Int value for VOrd Op arg. VOrd is 1 for first Op arg, 2 for 2nd, etc. VFunc is called at Match time, *NOT* when define/create Pat!
SFunc SFunc(PDict, Data)() {...}	*Stash* Func (Provided in Pat definition, after #S_Func directive) Returns Int 0 to abort whole match, everything else means proceed. Data is from #S_Data, or Int 0. Called *after* all other stash directives.
PFunc PFunc(PDict)(Pos) {...}	*Pat* Func (Provided in Pat definition in place of sub-Pat or Str) Returns Int 0 if match fails, Non-0 Int if success. Otherwise returns Str or Pat that needs matching. NOT for Char Pats!
RFunc (or GapRFunc) RFunc(PDict, SPos, EPos, ROrd, Numb, PatP, PType)() {...}	*Res* Func (Called by Match to inform script of each succ. match) Only works for Pats with #Res flag! **** Return 0 to abort/terminate, non-0 to continue. SPos and EPos are matched byte range. ROrd is 1-based order in RStack. Numb is which Elt (of Alt), which Rep, or what Span size matched. PatP is the one matched and PType is its Pat type. (ROrd, Numb, PatP, and PType are meaningless in GapRFunc.)

**** RFunc, like SFunc, are callbacks to convey match results to the controlling script. The process of matching a #Cons Pat normally involves searching, going down blind alleys and reversing course. This could be very confusing for the calling script, as the results communicated by SFunc callbacks could be subsequently invalidated. Instead, Match maintains an *RStack* (Result Stack) and can call the RFunc (instead of SFunc) *after* everything is done, using final results stored on the RStack.

Match invokes GapRFunc on text it scans over (*gap*) before matching (or matching again).

Char Range Defs:	
#Alpha	Alphanumeric A-Z, a-z, and 0-9
#Letter	Letters A-Z, and a-z
#Upper	Uppercase letters A-Z
#Lower	Lowercase letters a-z
#Symbol	Printing symbols in lower ASCII !-/, :-@, [-', and {-~" (Stops at Del)
#Digit	Decimal digits 0-9
#Binary	Binary digits 0 and 1
#Octal	Octal digits 0-7
#Hex	Hex digits A-F, a-f, and 0-9
#Quote	Quote chars ' and "
#WSpace	Whitespace chars, HTab, LF, CR, and Space
#Ctrl	Control chars ASCII 0-31 and Del
#Print	Printable ASCII !-~
#Punc	Punctuation !,.:;? (excluding quotes)
#EPunc	End-of-sentence punctuation !.?
#All	All chars, ASCII + Unicode

#ASCII	All ASCII chars \x01 - \xFF
#LoASCII	Lower ASCII chars \x01 - \x7F
#HiASCII	Higher ASCII chars \x80 - \xFF
#Unicode	All Unicode chars \x80 - \xF4\x8F\xBF\xBF (Last Unicode is 0x10FFFF)

Design and Basics

Regular Expressions (RE) have traditionally been used for matching text patterns. While an extremely compact and powerful pattern representation, RE are conceptually abstruse with a terse and effectively *write-only* syntax. In that sense, RE are perfectly fine for the machine, but not so much for the human being who writes them, or worse has to read and comprehend them.

scScript introduces a new, more procedural, *Pattern* (Pat) paradigm, drawing inspiration from *SNOBOL* as well as *Parse Expression* patterns. Recursive Pat objects are first *defined* and created, then *matched* against text from editor buffers. Only a few pre-defined Pat types are required, but this simple model can easily accommodate more types as well as specialized and custom functionality to suit client applications. Importantly, strings used to define Pats are simple text, there are no additional *in-band* cryptic symbols and no forests of escape chars.

A Pat typically refers to strings, other Pats, and even *callback* scScript functions that should be called during a match. Once created, the Pat is simply stored in a variable. It is thus passed around and used/re-used by many other Pats or matching functions. The matching process is also re-entrant. So a callback (scScript function closure) invoked from one Pat can itself match other Pats before returning results to its caller.

The simplest Pat just matches a string:

```
MyPat = Pat.Seq("hello world");
```

But a single string is trivial and not very interesting. The matching process (`Pat.Match`) can in fact accept a string directly, so `MyPat` is also superfluous! Perhaps a better example would be:

```
MyPat = Pat.Seq("hello", " ", "world");
```

where Seq is just a (conjunctive) *sequence* of strings that it will match in order.

But this being English, the first word can be capitalized! So a better Pat would be

```
MyPat = Pat.Seq(Pat.#InSen, "hello", " ", Pat.#Sen, "world");
```

where the `#Sen`/`#InSen` are flags that modulate case sensitivity for subsequent args. There are many possible flags, but they all work the same way. When a flag appears it sets the relevant parameters for all *subsequent* args, until another flag countermands it. The args say *what* to match, the flags say *how*!

Similarly, a `#Neg` (or `#Pos`) flag can be used to explicitly exclude/include subsequent args. So

```
MyPat = Pat.Seq("hello", " ", Pat.#Sen, "world", #Pat.Neg, ",")
```

would match `"Hello world!"` or `"Hello world."` but *not* `"Hello world, I am so happy."`!

Note: Both `#Pos` and `#Insen` are defaults, therefore elided at the beginning.

So far, MyPat has been quite rigid about the space char. How would this Pat match if there was a linebreak or even a tab character for *whitespace*? A better approach would be:

```
WSPat = Pat.Char(Pat.#WSpace);
MyPat = Pat.Seq("hello", WSPat, Pat.#Sen, "world");
```

Char is a Pat that matches any character within its defined range, here using the pre-defined set of *whitespace* characters. So MyPat would easily match:

"Hello world" or "Hello<tab>world", etc.

But this needs another modification, as there could be more than one space or tab:

```
WSPat = Pat.Char(Pat.#WSpace);
MyPat = Pat.Seq("hello", Pat.Rep(1, 0, WSPat), Pat.#Sen, "world");
```

> Ideally, Rep would be `Pat.Rep(1, 0, Pat.#Self + Pat.#Max, WSPat)` so it *selfishly* and *maximally* matches the entire whitespace without looking to see whether the Seq matches. More on this later.

The Rep above repeats its args one *or more* times (Min is 1 and Max is 0, meaning there is no maximum limits), so MyPat can match any reasonably spaced version of

"Hello world" or even "Hello<tab><space><space><newline><tab>world".

But of course, language is full of twists and turns... and sometimes *Hello* becomes *Greetings* and *World* becomes *Everyone*! The following can easily accommodate such a permutation:

```
WSPat = Pat.Char(Pat.#WSpace);
MyPat = Pat.Seq(Pat.Alt("hello", "hi", "greetings"),
                Pat.Rep(1, 0, Pat.#Self + Pat.#Max, WSPat),
                Pat.Alt(Pat.#Sen, "world", "everyone"));
```

Finally MyPat can match against any reasonable first greeting as Alt are disjunctions, choosing the first constituent *alternative* that matches; whereas Seq are conjunctions, all must match in sequence.

> Pat1.sc in the SampleSC directory will have most of the above examples, matching against Sample_BigC.txt, a fairly large sample data file (based on old source codes for scScript).

So the Pat here, ignoring case sensitivity, is defined as

- "hello", "hi", or "greetings", followed by
- one or more whitespace chars, followed by
- "world" or "everyone".

scScript Pats obviously requires a few more characters than an equivalent RE, but humans can read/write remarkably quickly when it actually makes sense!

> A very long time ago, a programming language called APL was created. It was based on the Greek alphabet and designed for linear algebra. One could boast of writing a whole APL program in one line!! But of course it took many hours of study to write that one line, and few could decipher it afterwards.
>
> To this day, some consider APL genius and an incredible tool for parsimonious expression.
> It makes them feel special and they will happily dedicate long hours to master it.
>
> Others just shake their head and walk away.
> The pattern matching in scScript was designed for us *others*.

Format

scScript is very flexible with its handling of function args, so Pat definition functions, like `Pat.Seq` or `Pat.Rep` can take very many arguments. These are broken into 3 sets: (appearing in order)

1) *Op Args*, (Min, Max, etc.) mandatory initial args for a few Pat types.
2) *Flags/Pat Args*, (Flags and Str/Pat/PFunc args) defining the *meat* for most Pat.
3) *Stash Directives*, (each also needs a value) for any Pat that needs them.

Op Args (1) always come first. They are not optional, they *must* be present and exactly as specified for the particular Pat types that require them. Most Pats do not require any and should not have them. So Rep needs Min and Max to establish the bounds of repetition (e.g. from 1 to 27 reps), and Iter needs the Cnt (to set both Min and Max), but Seq or Alt take no such Op Args.

Pat Args (2) define *templates* that actually match against text data. These come just after the Op Args, if any, and define what the Pat is looking for. Pat Args are usually Strings or Pats, but can also be callback scScript PFuncs (closures) that can succeed/fail directly (return `0` to fail or non-`0` Ints to succeed), or return other Strings or Pats for matching.

Integer *Flags* are also mixed in with Pat Args. They are easy to spot, as they are pre-defined integers, like `#Cons` or `#Neg`. Every time a Flag is encountered, it affects the operation of all *subsequent* Pat Args (or the entire Pat), so it is very typical to have one or more Flags at the beginning. Flags can be listed individually or combined together with `|` (bit-or) or `+` (add), as they are bit encoded and additive. Default Flags are normally left out, but are necessary if a *previously* set Flag has to be countermanded for subsequent args.

Note: Op (`#Min`, `#Max`), Mode (`#Self`, `#Cons`), and Result (`#NoRes`, `#Res`) flags should come *before* any Pat Args as these apply to the Pat itself, and not its args!

Pat Flags are *sticky*, once set they apply to all subsequent args, until countermanded. This is especially important for `#Dyn` and `#Neg`, as they change the *interpretation* and *logic* of all following ops.

++++ Flags in general only operate *inside* the Pat that contains them. They cannot *penetrate* sub-Pats to alter their behavior. For example `Pat.Seq(Pat.#Exact, Pat.Char("aA"))` *will* match all accented forms of the letters `a` or `A`, because Pat.Char will happily match them despite the `#Exact` flag in the superior Pat.Seq. (Despite appearances, `#Neg` is no exception, it just negates the result of the sub-pat match! So in `Pat.Seq(Pat.#Neg, Pat.Char(...))`, it will make the Seq fail if the Pat.Char succeeds!)

Stash Directives (*Dirs* for short) (3) are typically instructions to store (stash) information *when* the Pat matches (or in the case of *Location Directives*, when the Pat is defined). The matching process maintains a *PDict* to store this information and its contents can be accessed during or after the match. The initial purpose was to store useful bits of the match for the calling script, but it turns out the match process itself (and callbacks) want this data too! So this information is stashed as soon as an individual Pat matches, even if in the big picture, the *outermost* Pat has not matched yet. The to-and-fro searching behavior of *considerate* Pats means this data may be stashed (and possibly become invalid) many times!

A good use case is stashing info in one sub-Pat and using it *dynamically* in a subsequent one. Consider a Pat to match scScript *long* strings. These begin with "|>" with 0 to 8 intermediate "-" (dash) characters, such as "|--->". The string must end with the *exact* mirror image delimiter.

A simple Pat to match the opening delimiter is:

```
POpen = Pat.Seq("|", Pat.Rep(0, 8, Pat.#Self + Pat.#Max, "-"), ">");
```

But how would a PClose know what the POpen matched?

The easiest approach is for POpen to *stash* the match string using an SFunc callback:

```
POpen = Pat.Seq("|", Pat.Rep(0, 8, Pat.#Self + Pat.#Max, "-"), ">",
                Pat.#S_Func, StashRev);
```

StashRev is a one-line SFunc written to store the *reversed* matching string in PDict under "EndLS":

```
Function XStashRevX(PDict)() {PDict.EndLS = Sys.Reverse(Pat.GetStr()); Return 1;}
```

Pat.GetStr is an *accessor* function for use inside callbacks. As the name suggests, it returns the Str matched in the current Pat. But unfortunately, *reversing* takes a little more work, as |--> reversed would just be >--| and the *arrow head* needs flipping around! A correct approach would be to get the string CLen and construct a delimiter string:

```
Function StashRev(PDict)() {
    PDict.EndLS = Str.Form("<", Str.CLen(Pat.GetStr()) - 2, "-", 1, "|");
    Return 1;
}
```

Str.Form concatenates 1 copy of "<" to N-2 copies of "-" (where N is the opening string length) and 1 copy of "|", thus forming the ending delimiter for long comments. Now PClose can be written as:

```
PClose = Pat.Seq(Pat.#Dyn, "EndLS");
```

This will use the #Dyn key (EndLS) to find the data string "<---|" in PDict, then use *it* to match buffer data. So the PClose will not match against "EndLS", but the *value* stored in PDict.EndLS.

An improved approach would be to stash the S_Cnt in Rep and use it directly instead of counting chars in the POpen string! (This is also shown in SampleSC/Pat1.sc.)

Note: #S_Name and #S_Data are L*ocation Directives* and associate the Name and Data with the Pat at *definition* time, regardless of whether it matches anything. The former is quite useful for debugging and display while the latter can carry a payload into the match!

Paradigm and Performance

Pattern matching, while appearing simple, can in fact be quite complex and require an inordinate amount of processing, searching, and backtracking. While the traditional RE approach hides the *process*, the Pat system in scScript, like C itself, endeavors to be more procedural, allowing the script writer more insight and control over the matching process and the steps therein. A lot of work can also be offloaded to various callback routines which can themselves create/use Pats and call the (*re-entrant*) matcher! So a little theoretical understanding of the process goes a long way to ensure correct results and optimize performance. There are often *hard* ways and *easy* ways of doing things.

```
Pat.Match(Buffer Pat/Str Mode/MFunc StartPos FirstPos LastPos RFunc GapRFunc)(PatDict MStartPos MEndPos)
           REQ    REQ       opt       opt      opt     opt    opt     opt      opt      opt      Opt
```

The `Pat.Match` function takes a specified Pat or Str and matches it against text from an scEmacs buffer. It requires only the first two arguments, all the rest are optional. The match process begins at `StartPos` (defaults to `0`); if a match is found, it will call the `RFunc` callback and set the *out* variables `MStartPos` and `MEndPos` to the beginning and end of the match respectively. `Pat.Match` deals with byte pos, not char counts, and it is important to input byte pos that do *not* straddle `UTF-8` chars. `GapRFunc`, if provided, is a callback (like `RFunc`) that `Pat.Match` calls for *non-matching* text it *gaps* over as it scans the buffer.

The optional `FirstPos` and `LastPos` *in* variables delimit the permissible `Buffer` boundaries. This might seem unnecessary, but a Pat and its sub-parts *can* move to and fro within the data and it is sometimes beneficial to explicitly set the boundaries. (`FirstPos` is inclusive, `LastPos` is exclusive.)

If a `PatDict` *out* variable is provided, it will be set to the PDict created by the match process. Then again, if `PatDict` *already* holds a dict, `Match` will simply use it and add/alter pairs as necessary. While PDict will always be passed on to callback functions, a `PatDict` *out* variable (for `Pat.Match`) is necessary if the calling script itself wants to access it after the `Match` function returns.

The optional Mode *in* var determines how Pat.Match operates:

`#Anchor:`	Match *at* and only at `StartPos`. Do not look anywhere else!
`#Once:`	(Default mode) Match *starting at* `StartPos`. If the Pat/Str does *not* match, look at the next pos and try again! Continue *until* a match is found or reach `LastPos` (default is end-of-buffer).
`#Slide:`	Start like `#Once`. But after a successful match is found, *slide* over one character (multiple Pos if sliding over a `UTF-8` char) and go again!
`#Skip:`	Start like `#Once`. But after a successful match is found, *skip* over it, start from the first Pos *after* the match, and go again.
`MFunc:`	MFunc callback will determine *whether* and *where* to match from again.

The first two options can result in at most a single match, `Pat.Match` will return `int 0` if it fails and `int 1` if it succeeds (will also call `RFunc` if given). The `#Anchor` option will be faster in case of failure, as there is no scanning for the next potential match.

The next two modes can match very many times and `Pat.Match` will *not* return until all is done. Each time a successful match is found, it will invoke the optional `RFunc++` callback (if given), set `MStartPos`

and MEndPos, as well as stash data in PDict. But it will then #Slide or #Skip to the next starting position and continue to scan for another match. In that sense, these modes will continue finding *all* possible matches until the very end. As expected, #Skip will be faster, since it will scan fewer starting positions. When done, Pat.Match will return with the *number* of successful matches.

++ While RFunc, especially for sub-Pats, *appears* to be called during a match, it is actually called *after* the top level pattern match succeeds! Match simply pushes each intermediate success result on to an RStack and calls RFunc on its contents *after* all the searching and backtracking is done. This is done so the calling script does not get buffeted with RFunc results that become invalid because of #Cons backtracking.

Pat.Match can also allow a callback MFunc to dictate its operating mode, instead of using hard coded values. This is useful if the decision whether to continue (and from whence) depends on the data itself. The callback can return #Done to mean Pat.Match is done (it must have already matched once), #Slide or #Skip to continue at the respective next location, and #Custom if the callback itself sets the *next* StartPos where matching should resume.

> The Match apparatus is re-entrant, so an MFunc may launch another Match to determine the value for StartPos, where *the current* Match should resume!

The Pat being matched will have a big impact on the efficiency of the match operation. A simple search for the word "Hello" can be done with:

```
Pat.Match(BN, "Hello");
```

While not a Pat *per se*, a string is the simplest pattern there is. So this trivial match call will default to #Once mode and quickly search from the beginning of BN for "Hello".

But one can also use a slightly more complex *self-scanning* Pat:

```
PScan =  Pat.Rep(0, 0, Pat.Char(Pat.#All));        // Will match anything!
PHello = Pat.Seq(PScan, "Hello");

Pat.Match(BN, PHello, Pat.#Anchor);
```

PHello will match *any* sequence of characters that ends in "Hello"! In that sense, this Pat will *itself* scan down the buffer within a *single* match. So using #Once mode (instead of #Anchor) with such a Pat would be extremely wasteful *if* there is no match. In that case, Match would start at pos 0, search the entire buffer for "Hello", then move over to pos 1, and repeat the process, again and again. If BN has a modest 10K of data, #Once will look for "Hello" more than 50 million times.

> A simpler version of Pat.Char(Pat.#All) is Pat.Any() (of course, Pat.Go(1) is also an option!). *Pat.Any()* was a relatively late addition, as the need for efficiently searching over all chars became apparent only with use. So PScan = Pat.Rep(0, 0, Pat.Any()); is an improvement.

Interestingly, PHello works *almost* as fast as the direct Match of "Hello". Matching in #Anchor mode, PHello will do the scanning and Pat.Match will succeed immediately when PHello strikes. But Pats do a lot of internal work in addition to string matching (stashing data, handling callbacks, etc.) so the direct (string) Match of "Hello" will *always* be faster!

Most Pats, like simple strings, either match or not depending on their contents, they have no say or leeway in how they operate. But some do have *latitude* in how and what they match. For example, given repeating data, Rep can decide, on its own, *how many times* to repeat!

There are two theoretical dimensions for Pats that enjoy matching latitude:

1) *Greediness*: How many strings, characters, iterations? (`#Min` or `#Max`)
2) *Kindness*: Selfish or considerate of subsequent Pats? (`#Self` or `#Cons`)

Matched *on its* own, the simple `PScan` Rep (from above) will immediately succeed on any text in the buffer. The `Pat.Any` component will match *any* char, but worse `Pat.Rep` will *always* match with `0` reps! In fact, a Pat of the form: (assuming it is not part of another Pat and not being *considerate*)

```
Pat.Rep(0, 0, "xyz")
```

will instantly match with `0` reps! (scScript issues a warning for such a Pat.)

`Pat.Rep` defaults to `#Min` (*minimalist*) behavior. If its Min parameter is `0`, as in this case, it simply declares itself successful and stops *without* matching anything! But if given a Min greater than `0`, it will actually try to match repeatedly until the Min is satisfied. The `#Max` (*Maximalist*) flag changes all that, it makes `Pat.Rep` *greedy.* It will continue to iterate even after a successful match, again and again, until the repetition count equals its Max parameter (`0` here means there is *no* Max limit) or it runs out of data (buffer) to match. So the *maximalist* Rep will succeed with the highest rep count it can, between its Min and Max limits, inclusively.

A minimalist `PScan` (as written)

```
PScan = Pat.Rep(0, 0, Pat.Any());
```

if Matched on its own, will succeed immediately without *eating* any chars. A maximalist version:

```
PMaxScan = Pat.Rep(0, 0, Pat.#Max, Pat.Any());
```

if matched on its own, will eat every *last* char in the buffer. If stashing `#S_Cnt`, the Rep would therefore store the char count for the whole buffer. `#Max` mode is clearly slower than `#Min`, as the Pat will keep going, even after a match succeeds! (`SampleSC/Pat2.sc` will have these examples.)

Pats with operating latitude generally default to *selfish* (`#Self`) but can also be *considerate* (`#Cons`), the default case for Rep. A selfish Pat only cares about itself, whether greedy or not, it seeks to match on its own and be done. A selfish version of the example above, `PScanS`, *whether* `#Min` or `#Max`, will fail (with one exception)!

```
PScanS  = Pat.Rep(0, 0, Pat.#Self, Pat.Any());
PHelloS = Pat.Seq(PScanS, "Hello");
```

A minimalist `PScanS` will *eat* no chars, so `PHelloS` will fail unless the buffer *starts* with "`Hello`". Worse, a maximalist `PScanS` will eat *all* chars, so `PHelloS` fails with nothing left to match!

A `#Cons` Pat, unlike the `#Self` variant, is a team player. It will look to subsequent Pats and succeeds *only if* they too can succeed. So it does a *preliminary* match, then checks with every subsequent Pat in the Match hierarchy (in this case `PHellos`) to verify they too can be satisfied! This takes a lot of processing, but scScript has clever optimizations to avoid duplicate checks. (Yes, nasty complex code!)

The considerate (original) `PScan` in `PHello` will only *eat* until it reaches `"Hello"`. Such a minimalist and considerate `PScan` will go until the *first* `"Hello"`, but if `PScan` was maximalist, it would look past the first match, and *eat* until the very last `"Hello"`! In most cases `#Cons` slows down the match, but is semantically required. However, `#Max` and `#Cons` together are *often* a semantic error.

> Pattern matchers have historically differentiated between *lazy* and *greedy* for iteration. But the traditional categorization defined *greedy* as maximalist *and* selfish, with *lazy* being minimalist and considerate. So *lazy* traditional patterns would often do more work than their *greedy* counterparts!

The most important rules of thumb:

- Rep (defaults to `#Cons` and `#Min`)
  - Make it `#Self` when possible (usually `#Self` and `#Max`).
  - Make it `#Max` if required,
  - Avoid `#Cons` and `#Max` together!
  - NEVER Rep with only `#Neg` Pats (It will loop forever)!!

- Span (defaults to `#Self` and `#Min`)
  - Make it `#Cons` if necessary.
  - Make it `#Max` if required.
  - But avoid `#Cons` and `#Max` together!

- Alt (defaults to `#Self`)
  - Sometimes needs `#Cons`.

- Avoid 2 or more sequenced `#Cons` Pats, unless *absolutely* necessary.

It is best to make everything `#Self` unless `#Cons` are necessary. `#Cons` and `#Max` together is usually a mistake while multiple `#Cons` Pats will require huge resources (memory and time) to match.

> Rep Pats default to `#Cons` because they are often used as *fillers* where the container Pat specifies the end point. But Span specifies its own beginning and end patterns, so it defaults to `#Self`. Alt will also default to `#Self` as it is just a disjunctive version of Seq, so usually seeks to affirmatively match its own template. The big picture is easy to remember: everything is `#Min` and `#Self`, only Rep is `#Cons`.
>
> A Pat of the form `Pat.Rep(0, 0, Pat.#Neg, AnyPat)` will generally *not* work! `Rep` must *advance* from one char to the next, but a failing `AnyPat` will *not*. So the `Rep` will loop forever!
>
> One solution is `Pat.Rep(0, 0, Pat.#Neg, AnyPat, Pat.#Pos, Pat.Go(1))`. Here `Go(1)` will force an *advance* and avoid the infinite loop. (An automatic aid might help here, but would cause problems and complications in other cases; best to stick with the simplest paradigm.)

A good exercise is using Pats to count words in a document. A word can be (over) simplified as one or more contiguous alphanumeric letters (plus `underscore` for us programmers). These are then separated by a *non-word* consisting of one or more contiguous non-letter characters. Obviously, the buffer may start with a *non-word* (or not), so it is best to set the `PNotWord` `Min` to `0` instead of `1`.

```
PLetter  = Pat.Char(Pat.#Alpha, "_");
PWord    = Pat.Rep(1, 0, Pat.#Self + Pat.#Max, PLetter);
PNotWord = Pat.Rep(0, 0, Pat.#Self + Pat.#Max, Pat.Char(Pat.#Neg, PLetter));
```

`PWord` is defined as one or more contiguous `PLetter` characters, and `PNotWord` (the stuff between words) is defined as zero or more contiguous non-`PLetter` characters. Note the use of `#Neg` in the

definition of PNotWord. Both Reps are #Self and #Max, they just need to *eat* all that matches without worrying about the next Pat.

> Pat.Char cannot accept PFunc callbacks. These Pats look at one character at a time and decide whether it is in their *inclusion set*. Not only would the semantics of a PFunc callback not be clear, the performance hit of invoking it many thousands of time, even on small data files, would be prohibitive.
>
> Pat.Char only maintain *inclusion* sets of characters, not *exclusion* sets! When defining a Char Pat, #Neg chars (from a Str or another Char Pat) should only come *after* the #Pos ones. In a mathematical analogy, it only stores a positive *sum*; so positive numbers must be added in *before* the negative ones! Fortunately, scScript is smart enough to automatically include #All characters if the very first Str/Pat is #Neg. Char can accept an #Exact flag, but is #Smart by default, so À in data will match against A if À itself is *not* included in the Char.
>
> (While it is theoretically possible to also maintain separate *exclusion* sets in Pat.Char, it would double the workload *and* present semantic issues of priority in case of conflict!)

A very efficient way to count words is to ignore non-words and just let Pat.Match scan the buffer and do the counting, it will simply skip over any word *gaps*:

```
N = Pat.Match(BN, PWord, Pat.#Skip);
```

This function will scan forward to match PWord, then skip the matched word and scan until it hits again, this repeats until the buffer end. The returned Match count will in fact be the word count! But it is critical to use #Skip, as #Slide would count each word multiple times (once for each letter!).

> The distinction between #Skip and #Slide emphasizes the importance of *procedural* understanding when pattern matching, another reason why obfuscating or concealing the *process* is counter productive. Programmers (especially C programmers) need to understand the operational paradigm, what things mean and how they operate; hiding this behind a syntactic formulation (or opaque object!) just leads to problems. One can also use clever defaults and aids to catch common problems, but they will complicate the user paradigm and confuse even more when there is the inevitable failure.

It is also possible to have a *self-scanning* PCount pattern:

```
PCount = Pat.Rep(0, 0, Pat.#Max, PNotWord, PWord, Pat.#S_Cnt, "WCnt");
```

> Rep accepts a core Pat or Str to repeat. But it also accepts more than one and will implicitly sequence them, just like Seq Pats. Here, this works as if Pat.Rep was given Pat.Seq(PNotWord, PWord)!

Each PCount repetition first *eats* up the non-word letters, then matches a full word. The goal is to have as many reps as possible, so it is #Max. The default for Rep is #Cons, but there are no *subsequent* Pats, so nothing to be *considerate* of!

The actual match for PCount, like all #Anchor matches, is quite efficient:

```
Pat.Match(BN, PCount, Pat.#Anchor)(PDict);
```

But PDict is a required *out* arg in Pat.Match because the desired word count will be stashed there under "WCnt". Once the function returns, the script can simply print out PDict.WCnt or assign it to a variable.

The file Pat2.sc in SampleSC lists a few variations of this match to experiment with. But note that PWord and PNotWord are exact opposites *only* for ASCII characters. Things get more complicated for higher Unicode (UTF-8) characters, especially since the default #Smart conversion is in effect here. For example Ā is obviously *not* in #Alpha (as it is not LowASCII), so the #Smart regime will look at its *root* (A) and match it successfully for PWord. But PNotWord is explicitly the set of all chars, except #Alpha, so will positively match Ā itself. The system will therefore not bother to extract the *root* and simply declare Ā as being in PNotWord. Pat2.sc lists a very simple way to get the exact (desired) symmetry, so Unicode characters do *not* end up being in PWord *and* PNotWord at the same time:

```
PNotWord = Pat.Rep(0, 0, Pat.#Self + Pat.#Max + Pat.#Neg, PLetter, Pat.#Pos, Pat.Go(1));
```
(A positive Go(1) is required to *advance*, Rep would just be an infinite loop without it!!)

A good way to test such Pats during development is to have it write the matching words to *Output*. This can be achieved easily with a callback SFunc:

```
Function WriteWord()() {Write(Pat.GetStr(), " "); Return 1;}
```

Finally, PWord must be modified to invoke WriteWord after each successful match:

```
PLetter  = Pat.Char(Pat.#Alpha, "_");
PWord    = Pat.Rep(1, 0, Pat.#Self + Pat.#Max, PLetter, Pat.#S_Func, WriteWord);
PNotWord = Pat.Rep(0, 0, Pat.#Self + Pat.#Max + Pat.#Neg, PLetter, Pat.#Pos, Pat.Go(1));
PCount   = Pat.Rep(0, 0, Pat.#Max, PNotWord, PWord, Pat.#S_Cnt, "WCnt");

Pat.Match(BN, PCount, Pat.#Anchor, 0, 0, 1186)(PDict);
```

It is a very good idea to limit the range of a test match, especially one that prints a lot of info, when dealing with large files (Sample_BigC.txt in SampleSC is a huge file). And if word-by-word verification is desired, WriteWord can be made to call Ed.QueryMC!

While writing word after word to *Output* will significantly slow things down, it also provides an opportunity to abort (Ctrl-G) out of the process. The exact timing will vary, but this tends to occur during the SFunc callback (WriteWord) of PWord. If aborted, PCount will never succeed (and *finalize*), so WCnt will remain 0. If the count is critical, then WriteWord itself must record a running tally.

The above have illustrated three of the four common techniques for getting match results:

1) Use match count returned by Pat.Match itself.
2) Read PDict after Stash Directives store info/results.
3) Process results with SFunc callbacks invoked by Pats.
4) Process results with RFunc callbacks invoked by Pat.Match call.

RFunc and SFunc serve a similar purpose, both are callbacks used by Pat.Match to inform the scScript and process the match. But SFunc is often called *redundantly* for #Cons matches that are later *invalidated*, while RFunc is only invoked for (*finalized*) successful top-level matches.

SFunc, like other stash directives in individual Pats, are invoked when the particular Pat succeeds. But the Pat may have a #Cons flag and therefore match only *provisionally*, at the whim of subsequent Pats. So if a subsequent Pat fails, the Match mechanism will *backtrack*, consider the previous success (that called the SFunc) invalid, and look at another match for the Pat. This cannot be avoided because SFunc and other stash directives for a #Cons Pat cannot be delayed until subsequent ones are resolved,

as those later Pats may in fact rely on preceding stash directives for their template data (`#Dyn` data) or PFunc. But the process of searching, going down blind alleys, and reversing course for a `#Cons` Pat may be too much for the calling scScript (and its callbacks). A simple script expects only *final* information and cannot handle the to and fro that comes from `#Cons` pattern matching.

scScript provides an RFunc callback to avoid this problem. When an RFunc is present, `Pat.Match` will maintain an internal `RStack` data structure to record what was matched and in what order. This `RStack` will carry the burden of searching and backtracking, always storing the *final* path in the successful match, as invalidated values are popped off. So *after* the whole match is done, `Pat.Match` calls RFunc iteratively on each element of the `RStack`, giving the calling script the *illusion* of a straightforward (prescient!!) path without any backtracking.

There are a few operational points when matching complex multi-level Pats:

- It is imperative to give Pats a `#Res` flag *if* they need RFunc called for them. All Pats default to `#NoRes`, otherwise `Match` would be storing `RStack` info and invoking RFunc for even the most insignificant Char patterns that might get called 100K times on a small data file.

- While RFunc are *not* called redundantly for invalidated/obviated matches, their exact call order is difficult to anticipate. Because of the iterative unwinding of recursive match trees (internal `MStack`), `#Cons` matches tend to trace a path opposite what is naively expected. (Sadly, *everyone* is naive until they implement a pattern matcher!)

An excellent variation of the previous examples, that will (eventually) illustrate the dangers of SFunc calls, is counting comment words in C source files (and now you know why the sample data is C code and not a chapter of War and Peace). There are two comment types in C, `//` initiated *line* comments as well as `/*` and `*/` delimited *block* comments. Naturally, comment delimiters should be ignored if in a string, just as string delimiters are ignored in a comment.

A logical way to start such an endeavor is by writing a Pat outline first:

```
PStr    = Pat.Span('"', '"');
PLCom   = Pat.Span("//", "\n");
PBCom   = Pat.Span("/*", "*/");
PCCnt   = Pat.Alt(PStr, PLCom, PBCom);
```

The repeated application of `PCCnt` (in #Skip mode) will find all `PLCom` and `PBCom` without looking inside C strings! But there are two problems:

1) C sources may also include *char specs* for quotes (`'"'` or `'\"'`).
2) Strings may include *escaped* quotes!

Problem (1) can be easily solved by manually including char specs (2 versions) as the first Alt args, so as to *catch* them first before they can gum up the works:

```
PCCnt = Pat.Alt(|>'"'<|, |>'\"'>|, PStr, PLCom, PBCom);
```

The *long* string format is used here to avoid an extra escape layer required in *short string* formats. Otherwise the backslash itself (and possibly the double quote) in the second string would need an escape. It would have to be written as `'\\"'` or worse `"\\\""`, making it very hard to read. (Shades of RE!)

The solution for problem (2) starts with a more flexible Pat to *eat* chars between quotes. Then it becomes (seemingly) easy to match a C Str despite included *escaped* quotes:

```
PAnyC   = Pat.Rep(0, 0, Pat.Any());
PStr    = Pat.Seq('"', PAnyC, Pat.#Neg, |>\"<|, Pat.#Pos, '"');
```

The idea is to start with one quote, skip any \", but stop at the closing quote. Sadly, the clumsy formulation above will fail, even if there is only a single escaped quote! `PAnyC` will expand to eat just the backslash "\" , leaving the quote immediately following it to prematurely end the string.

A working, but convoluted, approach would be:

```
PStr    = Pat.Seq('"', PAnyC, Pat.Rep(0, 0, |>\"<|, PAnyC), '"');
```

This breaks the target string into a number of *segments*, each separated with an escaped double quote. So the inner Rep will match any number of segments following the initial one, such as:
`"----------"`, `"----\"----"`, `"----\"----\"----"`, etc.

The segmenting approach works, but an elegant and slightly faster solution uses Alt to specifically eat *escaped* quotes (\") inside the C String: (given in `SampleSC/Pat3.sc`)

```
PInStr  = Pat.Rep(0, 0 Pat.Alt(|>\"<|, Pat.Any()));
PStr    = Pat.Seq(|>"<|, PInStr, |>"<|);
PLCom   = Pat.Span("//", "\n");
PBCom   = Pat.Span("/*", "*/");
PPCnt   = Pat.Alt(|>'"'<|, |>'\"'>|, PStr, PLCom, PBCom);

Note:   Pat.Match(BN, PPCnt, Pat.#Skip) will return a large number, it counts all hits on PPCnt!
```

This works well and leaves `PLCom` and `PBCom` to capture comments for word counting. Word counts are easy, they were done before:

```
PLetter  = Pat.Char(Pat.#Alpha, "_");
PWord    = Pat.Rep(1, 0, Pat.#Self + Pat.#Max, PLetter, Pat.#S_Func, WriteWord);
PNotWord = Pat.Rep(0, 0, Pat.#Self + Pat.#Max + Pat.#Neg, PLetter, Pat.#Pos, Pat.Go(1));
PCount   = Pat.Rep(0, 0, Pat.#Max, PNotWord, PWord, Pat.#S_Cnt, "WCnt");
```

It seems easy to transform `PLCom`/`PBCom` for counting:

```
PLCom   = Pat.Seq("//", PCount, "\n");
PBCom   = Pat.Seq("/*", PCount, "*/");
```

But this approach will fail for subtle but important reasons:

1) `PCount` has to be `#Cons` as it has to bridge from the opening delimiter to the closing one in `PLCom`/`PBCom`. So `PCount` first tries `0` words, then `1`, then `2`... until finally reaching where the calling Seq would match the closing delimiter. But each time `PCount` tries a Rep, it will invoke its SFunc thinking it has succeeded, only for the calling Seq to fail and dash its hopes.

2) `PNotWord` also has a problem. It is `#Self` and `#Max`, so what stops it from *eating* the closing delimiters? A comment line can end with a punctuation (or extra space) char; in that case

PNotWord will read it, the subsequent Newline, and the beginning comment characters of the next line! It is also quite common in C to have line comments with no words at all, just a lot of stars, dashes, or other ornamentation.

One solution is to restore PLCom/PBCom back to Span again, give them both an SFunc (not a problem as both Spans will be #Self), and launch a new Match from inside the SFunc:

```
PInStr  = Pat.Rep(0, 0, Pat.Alt(|>\"<|, Pat.Any()));
PStr    = Pat.Seq(|>"<|, PInStr, |>"<|);
PLCom   = Pat.Span("//", "\n", Pat.#S_Func, CountWords);
PBCom   = Pat.Span("/*", "*/", Pat.#S_Func, CountWords);
PPCnt   = Pat.Alt(|>'"'<|, |>'\"'<|, PStr, PLCom, PBCom);


Function CountWords()()
{
    Var Range = Pat.GetRange();
    Var PosS = Range & 0x00FFFFFFFF;
    Var PosE = Range >> 32;

    WrdCnt += Pat.Match(BN, PWord, Pat.#Skip, PosS, PosS, PosE);
    Return 1;
}
```

Warning: It is imperative to include "Return 1;" at the end of *every* callbacks. Otherwise, the match will fail and the results will be hard to comprehend and debug! Details of Pat.Match behavior are generally quite cryptic, but can be illuminated by turning on debugging flags in scScript sources.

The above relies on Match being re-entrant and working from inside (one of its own) callbacks. But this is very much in the spirit of C, just break the problem down and let brute force have its day without much further thought from the script writer.

Another, perhaps more elegant, solution is to introduce a PNLet, where PCount happily loops on *non-letters* (individually) *or* PWord Pats:

```
PInStr  = Pat.Rep(0, 0, Pat.Alt(|>\"<|, Pat.Any()));
PStr    = Pat.Seq(|>"<|, PInStr, |>"<|);

PLetter = Pat.Char(Pat.#Alpha, "_");
PNLet   = Pat.Seq(Pat.#Neg, PLetter, Pat.#Pos, Pat.Go(1));
PWord   = Pat.Rep(1, 0, Pat.#Self + Pat.#Max, PLetter, Pat.#S_Func, GotAWord);

PCount  = Pat.Rep(0, 0, Pat.Alt(PNLet, PWord));
PLCom   = Pat.Seq("//", PCount, "\n");
PBCom   = Pat.Seq("/*", PCount, "*/");
PPCnt   = Pat.Alt(|>'"'<|, |>'\"'<|, PStr, PLCom, PBCom);
```

Each call to GotAWord would simply increment the word count! This is dead simple as PWord is #Self and matches with finality. (Incidentally, a call to Pat.GetCnt() from inside GotAWord would return the char count in the given word. So one could tally exactly what percentage of the source file has been allocated to comments!)

The #Neg logic flag, used here and previously, seems quite simple, but introduces some surprising complexity, especially when applying to multiple args. The simple Pat:

```
P1 = Pat.Seq("A", "B", "C", Pat.#Neg, "D");
```

will happily match "ABC", but not "ABCD". In that sense P1 does not care if "ABC" is followed by another character, as long as it is not "D".

But things get more complicated with:

```
P2 = Pat.Seq("A", "B", "C", Pat.#Neg, "D", "E", "F");
```

Once again it will happily match "ABC". But rather than *specifically* excluding "ABCDEF" as one might naively expect, P2 really excludes "ABCD" (and by extension "ABCDEF"), "ABCE", and "ABCF". So while the Seq forms a *conjunction* of its #Pos templates, it forms a *disjunction* of its #Neg ones! Perhaps surprising, this is in fact quite logical and practical, allowing:

```
P3 = Pat.Seq("A", "B", "C", Pat.#Neg, "D", "E", "F", #Pat.Pos, Pat.Any(), "X", "Y", "Z");
```

to match any string of the form "ABC*XYZ" as long as * is not "D", "E", or "F". If one really does want "ABC" but not "ABCDEF" (the naive initial expectation of P2), it can be written as:

```
P4 = Pat.Seq("A", "B", "C", Pat.#Neg, Pat.Seq("D", "E", "F"));
```

so the inner Pat.Seq creates a conjunction of "DEF", which is then explicitly negated *en masse* for the outer Seq. Of course, this trivial case is best written as:

```
P4a = Pat.Seq("ABC", Pat.#Neg, "DEF");
```

#Neg flags are seldom used with Alt and Span because exclusionary templates are often *too* willing to match on their own. But this excess promiscuity is greatly mitigated in #Cons matches where sub-patterns effectively become part of a larger ensemble that must match.

Exclusion, and therefore #Neg flags, do play a large role in Char Pats. Each such Pat is really just a set of characters, and it is often convenient to define a larger set first, only to then exclude a few exceptions. While Char should logically require a #Pos set before losing characters to #Neg, it is smart enough to assume a base of #All characters if its definition *starts* with an exclusion (but this too can lead you astray if that was *not* your intention, especially when dealing with #Smart root considerations for UTF-8 chars).

> #Smart and #Exact flags were late additions to the pattern matcher. These became necessary as scEmacs was expanded to understand Latin Unicode chars. So scScript now uses an internal table that represents accented chars, their *root* ASCII letters, and whether they are upper or lower case. As a result, the system can match *entree* against *entrée* or indeed against *ĔṇŤṛĒĖ*. The default #Smart flag means root (base ASCII) letters in a Pat can match their accented brethren (subject to case #Sen or #Insen flags) as well as their plain unadorned versions. But #Exact forces base (unaccented) letters to *only* match identical (again subject to case flags) base letters. Whether #Smart or #Exact, accented letters in the Pat can *only* match identically accented ones in data; so *entrée* will always match *entrée* and never *entree*. (If explicitly excluding a Unicode char with a Char Pat, make it #Neg and #Exact, otherwise its root may match anyway!)

Another useful example (given in Pat4.sc of SampleSC), builds on previous work to print out a C source file without its comments: (Lawyers sometimes ask for this!!)

```
PInStr  = Pat.Rep(0, 0, Pat.Alt(|>\"<|, Pat.Any()));
PStr    = Pat.Seq(|>"<|, PInStr, |>"<|);
```

```
PLCom   = Pat.Span("//", "\n");
PBCom   = Pat.Span("/*", "*/");
PCopy   = Pat.Alt(Pat.#Res, |>'"'<|, |>'\"'<|, PStr, PLCom, PBCom);

Function DoCopy()()
{
    Var Ord = Pat.GetOrd();             // 0 if GapRFunc

    If (Ord == 4)                       // PLCom
        Write("\n");
    Else If (Ord < 4)                   // GapRFunc, charspecs, or PStr,
        Write(Pat.GetStr());            //   but NOT a comment!

    Return 1;
}

Pat.Match(BN, PCopy, Pat.#Skip, 0, 0, 0, DoCopy, DoCopy);
```

`Pat.Match` goes through the entire buffer, matching against `PCopy`. Since `PCopy` is the only `#Res` Pat, the RFunc (`DoCopy`) will be called each time it matches. `DoCopy` uses `GetOrd()` to determine which component of Alt matched and what to write out. The char specs and `PStr` are written out verbatim, text matching `PLCom` is reduced to just a `<newline>` char, and `PBCom` strings are ignored completely.

The secret sauce comes from the second `DoCopy` in the `Pat.Match` call. This `DoCopy` is the GapRFunc callback, it is called on text the matcher gaps *over* on the way to a match. Here the GapRFunc `DoCopy` will transcribe the unmatched lines of C code, not quotes, not char specs, nor comments. So the unmatched text will be written out as `GetOrd()` (and also `GetPat()`) will return `0` when called from inside a GapRFunc.

> The above operation will likely leave a great deal of whitespace. A good approach is to use another pass to remove excess whitespace. In general, multiple simple passes are easier to code, test, and run than a single complex one that tries to do too much.
>
> The following is sample code to remove excess whitespace. It reads from `OBN` (assuming a version of the above code uses `Ed.Write` to write to an `OBN` buffer instead of `Write` to `*Output*`) and `Xfers` the new version to the `OOBN` buffer.
>
> ```
> // Discards the extra blank lines
> Var PWS     = Pat.Rep(1, 0, Pat.#Max, Pat.Char(Pat.#WSpace));
> Var PBlank  = Pat.Seq(Pat.#Res, PWS, "\n");
>
> // Catches the single blank line + any code.
> Var PC      = Pat.Rep(0, 0, Pat.#Max, Pat.Char(Pat.#Neg, "\n"));
> Var PCode   = Pat.Seq(Pat.#Res, PC, "\n");
>
> Function CodeRFunc(Null, SPos, EPos, Null, Null, ThePat)()
> {
>     If (ThePat == PBlank)
>         Ed.Write(OOBN, "\n");
>     Else
>         Ed.Xfer(OOBN, OBN, SPos, EPos);
>     Return 1;
> }
>
> Pat.Match(OBN, Pat.Alt(PBlank, PCode), Pat.#Skip, 0, 0, 0, CodeRFunc);
> ```
>
> It might seem appealing to use separate SFunc for `PBlank` and `PCode` instead of the single RFunc; but that will not work!! The Reps in `PWS` and `PC` are `#Cons` (by default) and `#Max`. So they will repeat until their containing Seq is satisfied, at which point the Seq triggers off its SFunc. But the ever greedy `#Max` Rep will then look at the next line... and the Seq will be satisfied again! When it comes to `#Max`, only RFunc give the proper result. Here, this requirement spreads from Rep to its Seq container because of `#Cons`.

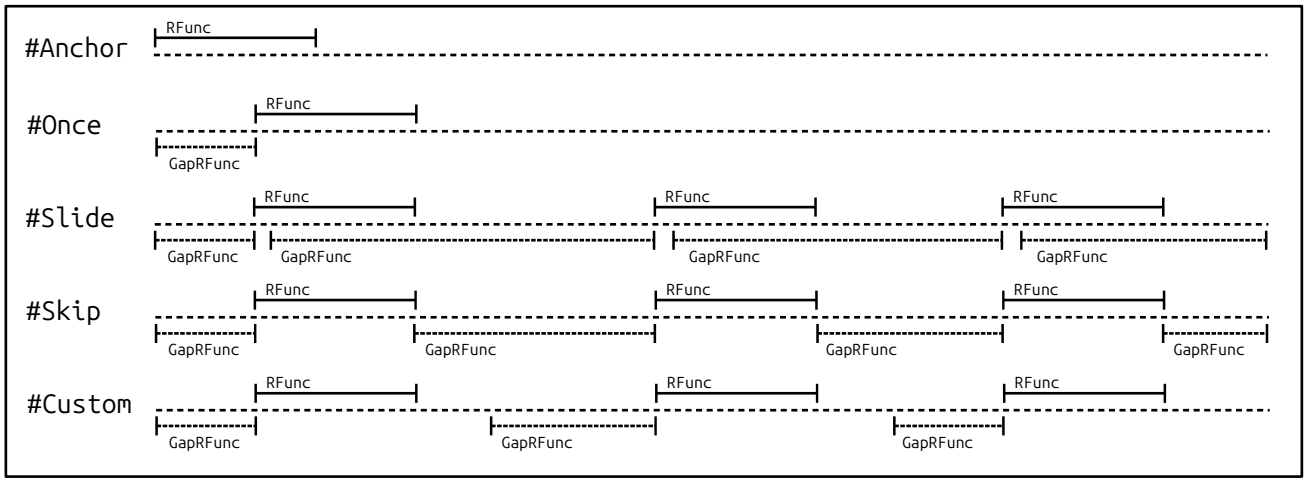
There are two important notes in the above scenario: (the main code before the *whitespace* digression)

1) `DoCopy` *must be* an RFunc, it cannot be an SFunc for `PCopy`. SFunc are called in a tight sub-loop that matches individual patterns/strings while GapRFunc callbacks are called from the *top-level* Match loop. Therefore, SFunc are invoked *before* Match recognizes success and has a chance to invoke the GapRFunc for the *preceding* data. The problem here is not that SFunc would be obviated, but that it would be called *before* GapRFunc, thus scrambling the output.

2) The above example could be implemented, albeit inefficiently, with just an RFunc (and no GapRFunc). But the final `Alt` would need one or two more entries. At a minimum the `Alt` would need a final `Any()` to match regular C code, that is not currently matched by other alternatives, or perhaps a `PWord` and `PNotWord` to serve the same purpose. Either way, GapRFunc in conjunction with a `#Skip` Match will be simpler and faster.

The GapRFunc is called on areas where `Match` scans the data buffer *before* its Pat/Str arg bites and the RFunc is invoked. In the case of `#Anchor`, *nothing* is ever scanned, so the GapRFunc is *not* called. Similarly, `#Once` will call the GapRFunc only on data *preceding* the match. This could be *nothing* if the Pat matches instantly or it could be the entire buffer if Pat does not match at all! The remaining match modes will similarly call the GapRFunc only on gap area, which in the case of `#Slide` may greatly overlap matched areas.

> The script itself could replicate GapRFunc functionality by maintaining state variables in the RFunc, and perhaps calling it explicitly after the final Match. But the main purpose of a scripting language is to make life easier, so GapRFunc is provided as a convenience.



Another example looks for *palindromes* in a source file. While a highly unlikely task, it will illustrate an important point. (And it did find many instances of *pop* and *level* in scScript sources!)

```
Function GetPal()()
{
    Var S = Str.CCase(Pat.GetStr(), Str.#Low);
    Var L = Str.CLen(S);

    if ((L > 1) && (S == Sys.Reverse(S)))
        Write(S, " "), PalCount += 1;

    return 1;
}
```

`GetPal` is an SFunc to verify if the matched word is a palindrome. If so, it will write it out and increment the global `PalCount` variable. Note the use of `Str.CCase` to change the extracted string to lowercase. This is important for the test stage, as "`Baab`" will not equal (`==`) "`baaB`" when reversed.

Instead of the naive definition of a word, this example will use the proper C definition: a group of alphanumeric characters starting with a letter or underscore! This excludes pure numbers or words of the form `2MyVar` etc.

The outline starts with PAlph and PWord:

```
PAlph =  Pat.Char(Pat.#Alpha, "_");
PWord =  Pat.Seq(Pat.Char(Pat.#Letter, "_"),
                 Pat.Rep(0, 0, Pat.#Self + Pat.#Max, PAlph),
                 Pat.#S_Func, GetPal);

Pat.Match(BN, PWord, Pat.#Skip);
```

But this will be sub-optimal! Sooner or later, Match will come across a hex number such as `0xACF2300B`, scan over the leading `0`, and consider the rest a word!! While this will never be a palindrome, it is obviously the wrong thing to do.

The solution is to expand the definition with an *elimination* pattern:

```
PAlph =  Pat.Char(Pat.#Alpha, "_");
PElim =  Pat.Rep(1, 0, Pat.#Self + Pat.#Max, PAlph, Pat.#S_Func, NotWord);
PWord =  Pat.Seq(Pat.Char(Pat.#Letter, "_"),
                 Pat.Rep(0, 0, Pat.#Self + Pat.#Max, PAlph),
                 Pat.#S_Func, GetPal);
PPalin = Pat.Alt(PWord, PElim);

Pat.Match (BN, PPalin, Pat.#Skip);
```

Using an Alt with an explicit elimination pattern (`PElim`) will prevent the `Match` scanner from partially eating a *word-like* alphanumeric string. `PElim` will fully digest and eliminate anything that could cause such confusion.

The final example (yes, *final*!!), while surprisingly simple, was actually used to test scEmacs. A routine was needed to create Windows-style files, with `<CR><LF>` terminating every line, from Linux-style files with just `<LF>` line ends. Importantly, it should be smart enough *not* to add redundant chars if the file already had `<CR><LF>` on some lines.

```
Var BN = Ed.ReadFile("./SampleSC/Sample_BigC.txt", Ed.#Show);
Var OutBN = Ed.GetBuf("./SampleSC/Sample_TestCR.txt");          // Output!

Var LinePat = Pat.Seq(Pat.#Res, "\n", Pat.#Neg, Pat.Seq(Pat.Go(-2), "\r"));
Var LineCount = 0;

Function GapRFunc(PDict, Start, End)()
{
   Ed.Xfer(OutBN, BN, Start, End);
   Return 1;
}

Function RFunc()()
{
   Ed.Write(OutBN, "\r\n");
   LineCount += 1;
   If (LineCount % 64 == 0) Write(".");
   Return 1;
}

Pat.Match(BN, LinePat, Pat.#Skip, 0, 0, 0, RFunc, GapRFunc);
```

The solution uses a one-line Pat along with 2 callback functions. The script reads the input file into the BN buffer and creates a fresh output buffer, OutBN, to receive the data. This is the normal mode for automated file changes, reading and writing to the same buffer is *usually and most emphatically* a recipe for disaster. Once the buffer is written, the user can view it and save the file with scEmacs. Alternatively, the code could include the following, after the Pat.Match:

```
Ed.ShowBuf(OutBN);
Ed.SaveFile(OutBN);
```

Note: Code in SampleSC/Pat4.sc *DOES NOT* save the buffer, if it did you would have to delete the file each time you tried it!

In the above, the obviously named GapRFunc copies each *scanned* input line verbatim to OutBN on the way to matching its EOL <LF>. The RFunc then simply writes the <CR><LF> at the end of each copied line. It also outputs a dot (to *Output*) for every 64 lines, to show *signs of life* when processing large files. Note the use of Ed.Xfer instead of the previously used Pat.GetStr (or similar) function. Building strings takes a significant amount of memory management and processing, but *transferring* bytes from one buffer to another can be quite rapid, especially if the destination buffer is hidden and scEmacs does not have to do constant window updates/scrolls.

The transfer of data is instigated by a #Skip match of LinePat, which just looks for <LF> characters. But LinePat only matches if there is *not* already a <CR> before the <LF>. This is detected with a #Neg sub-Seq that jumps back *2 spots* and looks for a <CR>. If this sub-Seq fails, LinePat will happily match, the accumulated span that was *gapped* over by Match will be copied to OutBN by GapRFunc, and RFunc will add a <CR><LF> after it. This will reset the Match, which will #Skip over and continue processing lines from the very next position.

When the <LF> (or anything else) is matched, the core Match routine moves to the next character of input, this will be +0 by definition. In the case of LinePat, the immediately preceding character, the <LF> that just matched, will be at Go(-1). So the character before that, where the <CR> would be, uses a Go(-2).

An astute reader would ask "but what of the very beginning, suppose there is an <LF> at position 0?" Interestingly, that just works! Go(-2) will obviously fail as it cannot go *before* the beginning of the buffer. But LinePat does not care why its #Neg sub-Seq fails, as long as it does. Whether there is no <CR> or no Go(-2), the sub-Seq fails and LinePat fires.

Note: scEmacs cannot draw a <CR> as it has no visible glyph, nor a *carriage* to return! Instead, it just denotes the <CR> (and any other missing glyphs) with a small box. While scEmacs can write and save the OutBN buffer with all the <CR> chars, it normally just filters them out when reading a file in for the first time.

It is an easy matter to delve even deeper into pattern matching. But the above (lengthy) exposition and the many examples have covered major points, hopefully forming a good base to explore from. For those wanting to get deeper into the implementation of the matcher, the section on low-level debugging will introduce its low-level dynamics, and of course there is always the source code!!

Common Errors

Pattern matching, as you have witnessed, is inherently complex, so it should be no surprise that there are a few common errors:

- Using RFunc without adding `#Res` to the main Pats, **
- Making Rep `#Self` without switching to `#Max` (it is normally `#Min`),
- Confusing `ByteCount` with `CharCount` and `#Rel` with `#Abs`.
- Using a `#Cons` Pat in a `#Cons` Rep (will slow things down drastically).

** I had a bewildering error when `Goto` was used in a test file to jump (over other code) directly to a `Match` call I wanted to study. Alas, the overzealous `Goto` I wrote also jumped over the RFunc definition/assignment, so the `Match` function was being fed `Int-0` instead of the expected RFunc closure! `Match` was perfectly happy and ran normally, no errors or warnings, but no RFunc was ever called. Turns out the compiler *saw* the RFunc `Function` and declared its variable, but the function innards was not defined at runtime because of the `Goto`, so it kept its initial value of `Int-0`.

The first mistake is often quite a head scratcher, as the match records no results and does not call the RFunc callback. While many such problems can be seemingly fixed with *smarter* defaults, the problems they introduce, when the smart defaults are wrong, would be even harder to catch! Perhaps a better solution would be to have an optional *check* function to warn if the pattern match is suspect.

The At Pat, although a convenient predicate for matching, can also cause a few problems, especially if the particular Match is called with explicit `StartPos` and `FirstPos` values. For example, `Pat.At(2)` will default to `#Rel`, so counts `2` characters from `FirstPos` given to the Match function. But a preceeding `Pat.Go(2)` will have advanced `2` characters from *`StartPos`*, perhaps a very different location. Similarly, `Pat.At(2, Pat.#Abs)` will count from the very beginning of the buffer, `Pos 0`, regardless of `FirstPos` or `StartPos`.

In summary, `#Rel` (default) for Go, counts characters from the *current* Match position, which will be `StartPos` in a fresh Match. But `#Rel` for At counts chars from `FirstPos`/`LastPos`. Similarly, `#Abs` for Go measures from `FirstPos`/`LastPos` given to the Match, but `#Abs` for At measures from the very beginning/end of the Buffer. So `#Abs` in Go generally corresponds to `#Rel` in At! (One can also argue for a future `#Byte` count option in Go and At Pats.)

A final common error is altering the Match buffer *during* the match. It is quite tempting to call `Ed.Del` or `Ed.Write` (even `Ed.Xfer`) from an SFunc or RFunc. But the low-level machinery will not expect or understand such buffer changes! If nothing else, this will alter the buffer length and the current Match position, typically leading to an infinite loop or (more often) a crash. An innocuous Pat like At(-1) cannot function if the buffer length is changed! The only workable approach is to use the Match buffer as the source template and copy/xfer as necessary to a *new* buffer, skipping all that should be deleted.

Low-Level Debugging

Pats are inherently recursive (tree) structures, as each Pat may include one or more other Pat/Str. Matching therefore requires a *depth-firs*t traversal of the Pat tree to compare terminal nodes (strings and chars) to data in the given buffer. In that sense, there is always an *order* to Pat elements, there are preceding Pats and subsequent Pats to compare. `#Cons` adds an extra dimension to this recursion, as a given match is placed on hold, while subsequent Pats are visited and verified.

```
...
Match Succeeded... Steps:590 Recursions:2

Begin Matching... From:75
    Pat:Alt From:75
    | Pat:Seq From:75
    | Pat:Seq From:75
    | | Pat:Rep From:77
    | | | Pat:Alt From:77
    | | | | Pat:Char From:77
    | | | Pat:Alt From:78
    | | | | Pat:Char From:78
    | | | | Pat:Rep From:78
    ...
    | | | | | Pat:Char From:108
    | | | | | Pat:Char From:109
    | | | | | Pat:Char From:110
    | | | | | Pat:Char From:111

Match Succeeded... Steps:228 Recursions:2

Begin Matching... From:112
    Pat:Alt From:112
    | Pat:Seq From:112
    | Pat:Seq From:112
    | | Pat:Rep From:114
    | | | Pat:Alt From:114
    ...
```

```
...
Match Succeeded... Steps:590 Recursions:2

Begin Matching... From:75
    Pat:Alt "Cnt" From:75
    | Pat:Seq "DStr" From:75
    | Pat:Seq "LCom" From:75
    | | Pat:Rep "ComW" From:77
    | | | Pat:Alt From:77
    | | | | Pat:Char From:77
    | | | Pat:Alt From:78
    | | | | Pat:Char From:78
    | | | | Pat:Rep "Word" From:78
    ...
    | | | | | Pat:Char From:108
    | | | | | Pat:Char From:109
    | | | | | Pat:Char From:110
    | | | | | Pat:Char From:111

Match Succeeded... Steps:228 Recursions:2

Begin Matching... From:112
    Pat:Alt "Cnt" From:112
    | Pat:Seq "DStr" From:112
    | Pat:Seq "LCom" From:112
    | | Pat:Rep "ComW" From:114
    | | | Pat:Alt From:114
    ...
```

All this recursion in *two* dimensions can be quite expensive, both in memory usage and processing time. scScript uses a more efficient and manageable *iterative* process instead, where automata theory guarantees the conversion at the cost of an explicit control stack. So scScript uses an `MStack` of *match steps* to control its iterative process. There is also an `RStack` to store *results* for RFunc invocation without backtracking. But the combination is quite complex and all but incomprehensible without specialized printouts.

```
...
1>MStack(Res:1)... Steps:590  Levels:2
1>Starting Match at 75...

00001 L01: ["Cnt"*0]->75
00002 L01: ["Cnt"*1]->75
00003 L01: ["Cnt"*2]->75
00004 L01: ["Cnt"*3]["DStr"*0]->75
00005 L01: ["Cnt"*3]["DStr"*1]->75 ***** FAIL
00006 L01: ["Cnt"*3]->75
00007 L01: ["Cnt"*4]["LCom"*0]->75
00008 L01: ["Cnt"*4]["LCom"*1]->77
00009 L01: ["Cnt"*4]["LCom"*1]["ComW":0]->77
00010  L02: ["Cnt"*4]["LCom"*1]["ComW"*0][C:1->1]->77
00011  L02: ["Cnt"*4]["LCom":1]["ComW"*0][C:1->0]["LCom"*2|1]->77
00012  L02: ["Cnt"*4]["LCom":1]["ComW"*0][C:1->0]["LCom"*3|1]->77 ***** FAIL
00013 L01: ["Cnt"*4]["LCom"*1]["ComW"*0]->77
00014 L01: ["Cnt"*4]["LCom"*1]["ComW"*1][Alt*0]->77
00015 L01: ["Cnt"*4]["LCom"*1]["ComW"*1][Alt*1][Char*32]->78
00016 L01: ["Cnt"*4]["LCom"*1]["ComW"*1][Alt*1]->78
00017 L01: ["Cnt"*4]["LCom"*1]["ComW"*1]->78
00018  L02: ["Cnt"*4]["LCom"*1]["ComW"*1][C:1->1]->78
00019  L02: ["Cnt"*4]["LCom":1]["ComW"*1][C:1->0]["LCom"*2|1]->78
00020  L02: ["Cnt"*4]["LCom":1]["ComW"*1][C:1->0]["LCom"*3|1]->78 ***** FAIL
...
```

scScript provides compile-time `#define` macros to turn on/off pattern matching information displays. The `SC_DEBUG_DISP_MATCH` macro, if set to `1`, will print out a trace of the matching steps to stdout

(either the debugger or the terminal, *not* *Output*). This is normally turned off as such output will *drastically* slow down operations. The match display will be of the format shown on the top left. But it will be hard to read if there are many sub-parts with the same Pat type. So `#S_Name` (*attachment*) directives are provided to name individual Pats, making debugging display references, as shown on the top right, easy to disambiguate.

The bottom display is produced when `SC_DEBUG_DISP_MSTACK` is set to `1`, it shows `MStack` contents at every step. The exact details are available in scScript source files, but each entry corresponds to a Pat. The number shows the *stage* being evaluated and the final `->` indicates the target position in the buffer. In the example above, `"Cnt"` is an Alt pattern, with its entries numbered `1` through `5`. It starts by looking at Pos `75` of the buffer. Stage `0` is generally the initial setup phase for every Pat. Stage `3` of `"Cnt"`, therefore refers to checking its 3rd entry, the `"DStr"` Seq pattern. This, in turn quickly fails when its very first entry does not match in step `00005`. But apparently the 4th alternative in `"Cnt"` matches in step `00009` and launches a `#Cons` check in step `00010`.

`[C:1->1]` on the `MStack` indicates a `#Cons` check has been initiated. The first `1` indicates the *cons level*, more useful when there are `17` levels and the display wraps around many times. The second `1` indicates which `MStack` record to check next (`0`-based with `-1` meaning it is done). Using the iterative `MStack`-controlled algorithm, `#Cons` checks proceed *backwards* by re-visiting (and advancing) previous steps. So the next step is to re-visit and advance entry `1`, therefore to check the 2nd stage of `"LCom"`.

There are quite a few symbols in these notations and they each convey their own subtle information about the Matching algorithm. For example, step `00011` has two `"LCom"` entries on the `MStack`, the first *now* uses a *colon* while the second has the more common *asterix*. This has to do with how the `MStack` is back-traversed to generate `RStack` entries when the match succeeds. The `MStack` entry with the asterix will set the `RStack` results, while the one(s) with the colon should be ignored on the reverse traversal (as they have been superseded) for this purpose. This is an optimization in the matching algorithm, there is no need to repeat a match that has already been done as part of `#Cons` deliberations!

Importantly, each match may take *very* many steps and nested `#Cons` checks can quickly become quite inefficient and confusing. It is not hard to write a match that pushes `50` or more entries on the `MStack`. To aid legibility (and avoid wrap-arounds), the system will write out the full `MStack` only on every 5th step, displaying just the last row for the other 4.

> Script writers should see these to appreciate the folly of multiple nested #Cons!

The time it takes to match and the number of steps involved (shown as a 5-digit counter above) is generally proportional to the size of the data buffer being matched. This is entirely expected and makes good sense as going through `100K` of data *should* take longer than `10K`. The complexity of the Pat will obviously have a similar impact, as each individual match, whether hit or miss, would require more steps. One would also expect to have more items on the `MStack` at a given time (horizontal span on the display above) for more complex patterns. Nevertheless, memory requirements should remain generally fixed for a given pattern, regardless of the data size it scans over.

Unfortunately nested `#Cons` have an exponential impact on memory requirements (average `MStack` depth) as well as processing time. Worse, nested `#Cons` could yield an `MStack` peak depth proportional to the data size! This is a problem, as the system will fail on larger data sets.

The following (highly) *contrived* example illustrates the point: (Alt will always treat its *last* entry as `#Self`, the source code had to be changed for this example to remove this optimization.)

```
AnyC   = Pat.Alt(Pat.#Cons, |>\"<|, Pat.Any());
EatStr = Pat.Rep(0, 0, AnyC);
DStr   = Pat.Seq(|>"<|, EatStr, |>"<|);

Pat.Match(BN, DStr, Pat.#Once);
```

For this example, assume BN contains a total of only 8 data characters:

`<space>`"Test"`<newline>`

The above match would result in the following (abbreviated) `MStack` trace:

```
1>Starting Match at 0...

00001 L01: [Seq*0]->0
00002 L01: [Seq*1]->0 ***** FAIL

1>MStack(Res:0)... Steps:2  Levels:1
1>Starting Match at 1...

00001 L01: [Seq*0]->1
00002 L01: [Seq*1]->2
00003 L01: [Seq*1][Rep:0]->2
00004  L02: [Seq*1][Rep*0][C:1->0]->2

00005  L02: [Seq:1][Rep*0][C:1->-1][Seq*2|0]->2
00006  L02: [Seq:1][Rep*0][C:1->-1][Seq*3|0]->2 ***** FAIL
00007 L01: [Seq*1][Rep*0]->2
00008 L01: [Seq*1][Rep*1][Alt*0]->2
00009 L01: [Seq*1][Rep*1][Alt*1]->2

00010 L01: [Seq*1][Rep*1][Alt*2][Any*1]->3
00011 L01: [Seq*1][Rep*1][Alt*2]->3
00012  L02: [Seq*1][Rep*1][Alt:2][C:1->1]->3
...
00040    L04: [Seq*1][Rep:1][Alt:2][C:1->0][Rep:2|1][Alt:2][C:2->0][Rep:3|1][Alt:2][C:3->0]
              [Rep*4|1][Alt*2][Any*1]->6
00041    L04: [+5..][Alt:2][C:2->0][Rep:3|1][Alt:2][C:3->0][Rep*4|1][Alt*2]->6
00042     L05: [+6..][C:2->0][Rep:3|1][Alt:2][C:3->0][Rep*4|1][Alt:2][C:4->10]->6
00043     L05: [+7..][Rep:3|1][Alt:2][C:3->0][Rep:4|1][Alt:2][C:4->0][Rep*4|1]->6
00044      L06: [+8..][Alt:2][C:3->0][Rep:4|1][Alt:2][C:4->0][Rep*4|1][C:5->0]->6

00045      L06: [Seq:1][Rep:1][Alt:2][C:1->0][Rep:2|1][Alt:2][C:2->0][Rep:3|1][Alt:2][C:3->0]
                [Rep:4|1][Alt:2][C:4->0][Rep*4|1][C:5->-1][Seq*2|0]->6
00046      L06: [+9..][C:3->0][Rep:4|1][Alt:2][C:4->0][Rep*4|1][C:5->-1][Seq*3|0]->7

1>MStack(Res:1)... Steps:46  Levels:6
```

Where `MStack` usage peaks at 16 entries, all to match 4 characters (Interestingly, the *normal* `#Self` version still requires 42 steps, but only 4 `MStack` entries). Doubling the data to 8 characters requires an acceptable 86 steps instead of the previous 46. But the `MStack` would peak at 28 entries! Matching 16 characters will need 188 steps and 52 `MStack` entries while 32 characters requires 326 steps and 100 entries. By extension, a modest 16K of data would need 48K `MStack` entries, each 64 bytes! (3 MB of stack memory seems huge to us older folks who started with 4k of RAM on 8-bit processors!)

> Alas, there are no obvious shortcuts or heuristics to reduce the burden of combined `#Cons`. Each `#Cons` level provides a fork in the search tree of matches and one cannot be eliminated without wiping out possible solutions.

Future Directions

The current Match algorithm has a very hard time going *backwards* in the text buffer. If a particular Pat starts at the end of the buffer and jumps (perhaps N chars/lines) backwards, it will eventually go past the beginning of the buffer (or permissible area) and fail. But Match is not smart enough to just stop, instead it advances `+1` position from the *previous* start and tries again, and again.... just as it would for a forward match going through a *gap* area.

While not quite as dramatic, mindlessly advancing `+1` position can also be inefficient for forward matches. If the particular word does not match, it makes no sense to advance `+1` char *into* the word and try again. It would be much better to jump directly to the *next* word instead. The same logic may apply to lines or paragraphs, depending on the particular pattern definitions.

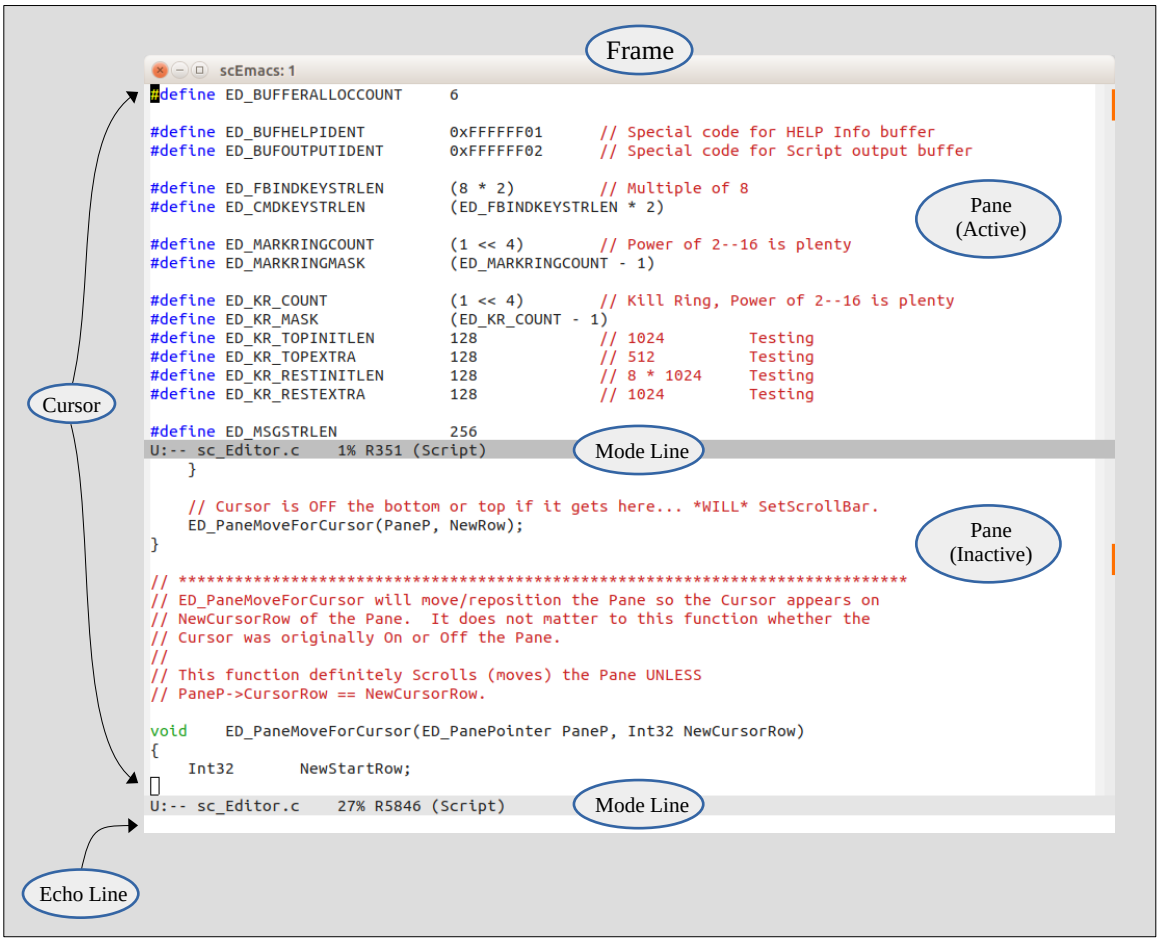
There are a few possible solutions, but the most appealing *general* one is to also provide a `FailMFunc`. The regular `MFunc` (if provided) is normally called *only* when the Pat matches. It tells the Match apparatus whether to continue and where to go next. A similar `FailMFunc` could perform that function when the Pat does *not* match. So instead of mindlessly advancing `+1` in search of the next match, it could *advance* backwards or even jump to the next word or line. This mechanism can enable proper backwards traversals, but also make forward matching more efficient. Given the more advanced functionality, it might be best to limit its use to a new `RMatch` function, one that takes an extra `FailMFunc` arg right after `MFunc`.

(An expedient step would be to supply a `#SkipWord` option for the current Match!)

scEmacs™

Overview

scEmacs reads text files into memory *Buffers* where they can be displayed and edited. It can have many such buffers at a given time and they can be displayed and manipulated independently. Buffers are displayed in *Panes* on windows called *Frames* (to avoid confusion with OS terms). There can be many frames (windows) at the same time, each showing *one or more* panes, and each pane will display the contents of its *single* buffer. However, multiple panes can display the very same buffer, as illustrated by the two below.



At any given time, there can be only a single *active* pane and it will reside in the frontmost frame window. Every pane always shows its cursor, as it is impossible to move it off the visible area. But only the *one* active pane has a solid (blinking) cursor, all other panes (on all other frames) show an empty cursor outline instead. Every pane also has a *Mode* line that names its buffer, shows the display mode (*Script* in this case), as well as other information. Only the active pane (top pane above) gets a highlighted (dark gray) mode line. Each frame window also has an *Echo* line on the bottom to display feedback and enter commands.

The whole purpose of scEmacs is to allow fast and efficient manipulation of text directly from the keyboard, *without* having to use the mouse. *Control* and *Meta* (*alt* on modern keyboards) key combos (used like *shift* keys) are employed to issue commands. For example, `Control-f` (`C-f`) is a command to move the cursor forward one character. Similarly, `Meta-f` (`M-f`) will move it forward one *word*.

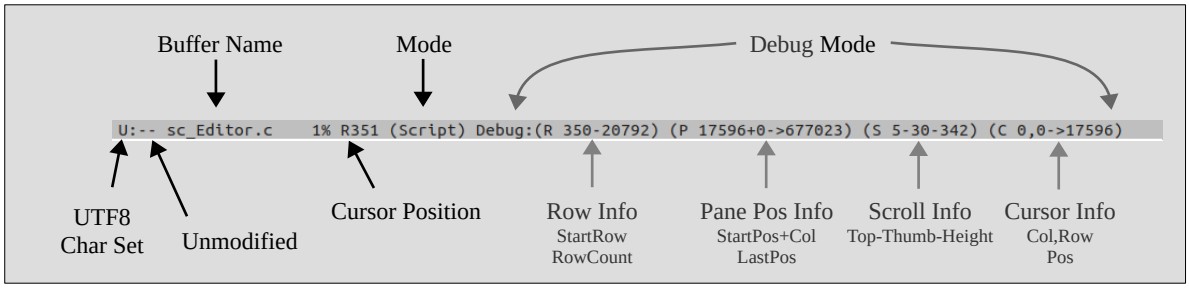
These keyboard commands are really *bindings* for internal functions that can also be invoked by name. While some internal functions are not bound to any key combinations, *all* must have a name! So `M-f` is bound to `forward-word` and can also be called by typing `M-x forward-word`. Actually `M-x` itself just calls `exec-named-cmd` which places the cursor in the bottom echo line for entry of command names. This echo line also provides auto-completion. Simply hit `Tab` after typing a few letters and scEmacs tries to automatically resolve the function name (or as much of it as it can).

> There are many function names that start with `f` so `M-x f` followed by `Tab` yields no progress while `M-x fo` followed by `Tab` will complete to `M-x forward-`. Subsequently typing `w` followed by another `Tab` will yield `M-x forward-word`!

scEmacs Specifications

- Supports UTF-8 characters in left-to-right format. Double-wide and compounded glyphs are not supported. A single `LineFeed` (`LF` or `\n`) character marks the end-of-line and files using CR or CR+LF conventions are *filtered* when read.
- Uses *signed* 32-bit indices, therefore limits file/buffer lengths to 2 gigabytes, subject to memory availability. This limitation pertains to *byte* count, not glyph count as UTF-8 characters can be up to 4 bytes long.
- Supports Text and Script modes, the latter is hand-tailored for scScript, but works for general display/edit of source code. Script mode provides color hilites and smart tabs to help formatting.
- Can display in color, but this information cannot be saved as `.txt` does not support meta-data. Source code displayed in Script mode is colorized *on the fly*. scEmacs has a simple syntax colorizer, but it is easy to extend and alter.
- Provides multiple levels of `Undo` but does not `Auto-save`.
- Provides incremental search and query replace operations.
- Lacks a mini-buffer facility, but allows for limited editing commands (delete, yank, etc.) on the echo line as well as *auto-completion* and query response interactions.
- Supports a *Kill Ring* and *Mark List*. Also supports mouse-based selection, scrolling, and even Unix style XSel copy/paste (middle button/wheel).
- Provides a complex *pop-up* window/menu facility for enumeration and selection of commands, buffers, marks, and kill ring text fragments.

- Provides specialized commands to drive scScript operation as well as i/o interaction.
- In Debug mode (compiled with `-DDEBUG` in `Makefile`), will display additional realtime information in the mode line and provide more debugging and test functions.



Starting Up

When launched, scEmacs brings up a single frame window with a single pane, displaying `*Temp-1*` buffer. This buffer can be immediately deleted, but then scEmacs will create `*Temp-2*` as it requires at least one buffer for operation.

- To create a new file, simply type to enter text, then use `C-x C-w` to `write-file`. scEmacs will provide the current directory information on the bottom echo line and prompt for the file name. At this point edit the directory specifications (using `Tab` for auto-completion while traversing the directory tree) and enter the desired file name.

```
U:** *Temp_1*    All R1 (Text) Debug:(R 0-0) (P 0+0->8) (S 0-683-684) (C 8,0->8)
Write File: /home/shawn/Desktop/Project/Sources/scEmacs-2.0/
```

- To edit an existing file, use `C-x C-f` for `find-file`. scEmacs will prompt in a similar way, but this time to read an existing file into the buffer where it can be viewed and edited.
- To save the file after some edits, use `C-x C-s` to `save-file`.
- Finally, use `C-x C-c` to `quit-and-exit` when done. scEmacs will warn and ask for exit confirmation if there are any *modified* buffers. One can also exit by closing all open frame windows, but this will simply exit scEmacs without checking buffers.

Commands

The most important command is `C-g` (`abort`) which flushes any mistyped or partial commands and exits any undesirable modes or situations. It works similar to `^C` on most systems.

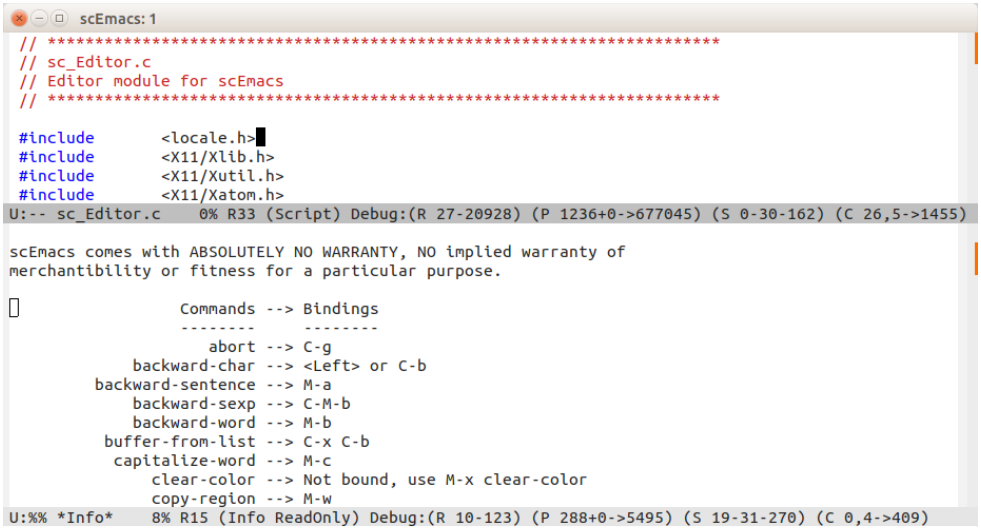
To reverse out any edits, use `C-/` (`undo`). If one `undo` does not go back far enough, repeat!

There are many commands to `kill` text, they all place the text in the *killring* for later retrieval. Use `C-y` to `yank` the killed contents (and paste them) at the current cursor location. For a previous kill, do a `C-y` first, then use one or more `M-y` (`yank-pop`) to go farther back in the killring. This unconventional method of *`cutting`* and *`pasting`* goes back to the historic origins of RealEmacs!

Most commands accept a *numeric* argument to repeat their operation, many even reverse direction if given a negative number. To enter a numeric count type `C-u` followed by the number (or type `M-number`) before entering the command. A `C-u` with *no* subsequent digits defaults to a count of 4, therefore `C-u C-f` will move the cursor forward 4 places just as `C-u 4 C-f` or `M-4 C-f` would.

`C-space` (`set-mark`) *marks* the cursor location. Subsequent movements of the cursor will *select* and highlight text between the mark and the cursor. Once selected, you can delete/kill the entire *region* with a single command! In fact typing *any* character will replace the selected text with that single letter. `C-g` will exit the selection mode and remove the highlighting. There are also commands to display previous marks and move the cursor to them, without selecting anything.

The command `C-M-x` (`exec-from-list`) will pop-up a menu/window listing all available commands. It permits scrolling and selection (using the mouse or Up/Down keys) of any command, which it will immediately execute. Hitting any key (even just `Control` or `Meta`) will get rid of the pop-up. While technically a selection menu, this is also a help mechanism, enumerating all commands and binding in one place. This, and all similar pop-ups, can be dragged from their title area (like any window) to re-position and will remember the new location.



Finally, `C-h` (`help`) will *split* the frame to show a listing all scEmacs commands in alphabetical order. `C-x o` (`other-pane`) will jump to the other pane and `C-x 0` (`kill-pane`) will close it. Note the mode line on the ReadOnly `*Info*` pane. This buffer cannot be modified, and if killed, will be regenerated on the fly. But being a buffer, it can be written (`C-x C-w`) to a file, at which point it stops being a readonly info buffer, so subsequent calls for `help` will re-generate a new version!

Operations:	These commands are essential to the operation of scEmacs. The most commonly used being `M-x` to execute a named operation. Many, but not all, scEmacs commands will accept a numerical arg. These can even be negative numbers and some commands will change polarity or direction based on these.	
`abort`	`C-g`	Stop current task/mode, get back to normal operations.
`quit-and-exit`	`C-x C-c`	End scEmacs session, will require confirmation if buffers are modified.
`exec-from-list`	`C-M-x`	Show command pop-up window.
`exec-named-cmd`	`C-x`	Prompts (in echo line) for command name to execute.
`help`	`C-h`	Generates and displays *Info* buffer. Will display the *definitive* guide to all implemented commands!
`Numerical Arg`	`C-u` or `M-digits`	Enter a number for the next command. Not a named command, just syntax!
`debug`	`DEBUG only`	Jump into active debugger. Debugger can set breakpoints then resume execution.
`test-loc-pos`	`DEBUG only`	Test low level positioning routines. Will take too long if buffer has more than 10-15K bytes. Results are written to debugger window (stdout).

Cursor & Display:	scEmacs is fundamentally a cursor-based editor as real *Emacs* predated mice and graphics. All editing operations, whether adding or deleting text, generally occur at the cursor. While it can be re-positioned with the mouse, scEmacs provides many commands for efficiently moving the cursor using just the keyboard. Horizontal units of movement are: (forward/back or end/beginning) • Character • Word • Sentence (works well for `Text`, less so in `Script` mode.) • SExp (a *balanced* expression of (), [], or {} possible nesting other SExp! Best in `Script` mode, little use in `Text`.) Vertical units of movement are: (next/previous) • Row (One horizontal sequence of characters, from edge of visible Pane.) • Line (Begins at the top of buffer or an LF, goes to end of buffer or another LF.) A long line displayed on a narrow pane may wrap around to form multiple rows. • Page (As many rows and lines that the pane can display at one time.) While a line is based on characters in the buffer, rows and pages are entirely dependent on the display size of the current pane, which can be trivially re-sized.	
`forward-char`	`<right>, C-f`	Move cursor forward one character.
`backward-char`	`<left>, C-b`	Move cursor backward one character.
`forward-word`	`M-f`	Move cursor forward one word.
`backward-word`	`M-b`	Move cursor backward one word.
`forward-sentence`	`M-e`	Move cursor to end of sentence.
`backward-sentence`	`M-a`	Move cursor to beginning of sentence.
`forward-sexp`	`C-M-f`	Move cursor forward one SExp. This cmd will stall and beep if it cannot balance the parens. Just like RealEmacs, it will not differentiate among different paren types. Unlike RealEmacs, it does not process quotes or skip comments.
`backward-sexp`	`C-M-b`	Move cursor backward one SExp.
`go-up`	`<up>, C-p`	Move cursor up one row. A *line* starts at the beginning of a buffer or after a `Linefeed` character and continues to the end of the buffer or another `Linefeed`. A long line displayed on a narrow Pane may wrap around to form multiple *rows*. Each row is a single horizontal sequence of characters, beginning and ending at the pane margin.
`go-scroll-up`	`C-<up>`	Move cursor up one row, scroll the pane down 1 row.
`go-down`	`<down>, C-n`	Move cursor down one row.
`go-scroll-down`	`C-<down>`	Move cursor down one row, scroll the pane up 1 row.
`go-line-end`	`<end>, C-e`	Move cursor to the end of the line.
`go-line-beg`	`<home>, C-a`	Move cursor to the beginning of the line.
`go-end`	`C-<end>, M->`	Move cursor to the end of the buffer.
`go-beg`	`C-<home>, M-<`	Move cursor to the beginning of the buffer.
`goto-char`	`M-g c`	Move cursor to the *numbered* (1-based) character and set a mark. If no count arg, prompts for a number on echo line. If cursor is on (or just after) a number, will show it as the default.

goto-line	M-g g, M-g l, M-g M-g	Move cursor to the *numbered* (1-based) line and set a mark. If no count, prompts for a number on echo line. If cursor is on (or just after) a number, will show it as the default.
next-page	PageDown, C-v	Move cursor to the next page. With number N, scrolls the pane N rows.
prev-page	PageUp, M-v	Move cursor to the previous page. With number N, scrolls the pane N rows.
center-page	C-l	Repositions displayed page. First application will scroll the display vertically to center cursor on the pane. If repeated, will scroll to place cursor at the top. The third iteration will scroll to place cursor at the bottom. Subsequent applications will repeat this cycle. Positive N arg will scroll to place cursor N rows below top and negative will scroll to place it N rows above bottom.
cursor-info	C-x =	Display cursor information in echo line. Will specify the char under the cursor as well as its buffer *Pos* and visible pane *Col*,*Row*.

Mark & Selection:	Many operations will operate on a selection range, a highlighted region of text between a *mark* and the cursor. The traditional practice is to use C-space to set a mark, then move the cursor ahead or behind it, selecting a region. But it is also common to select by dragging the mouse. (C-g (abort) will immediately cancel the selection and remove the highlighting.) Once selected, the entire region can be deleted, instantly replaced with typed chars, or killed and placed in the killring. (Undo is always available to remedy accidental deletion.)	
set-mark	C-@, C-space	Place a mark at the cursor potion. scEmacs enters *selection* mini-mode, so subsequent cursor movements will select the intervening text region.
mark-from-list	C-M-z	Displays pop-up showing all current marks. Select one to move cursor to.
exchange-point-mark	C-x C-x	Swap position of cursor with last mark.
select-all	C-x h	Select entire contents of buffer. Will mark the beginning and position cursor at end.
select-word	double-click	Selects word (also whitespace) under mouse. Positions mark at the beginning and cursor at the end.
select-line	triple-click	Selects entire line under mouse. Positions mark at the beginning and cursor at the end.

Editing:	The most common ways to change a buffer is to type text in, delete text, or *kill* some text and perhaps *yank* it out in a new location (instead of the more modern *cut* / *paste*). The unique killring mechanism acts as a circular stack of *killed* text fragments, ready for pasting. `C-y` (`yank`) will paste the latest text fragment at the cursor, but immediately subsequent `M-y` (`yank-pop`) commands will swap it for preceding ones from the killring. scEmacs has only a single global killring, shared between all buffers. So it is trivial to `kill` or `copy` in one buffer and `yank` it into another. Unlike `kill`, `delete` commands do not store in the killring, so *cannot* be used to `yank` text. But `Undo` will happily replace text removed by either, albeit in its original position and without altering the killring. There are also options to change word case (upper, lower, capitalize) and transpose characters.	
`delete-forward`	`<delete>, C-d`	Delete character under cursor.
`delete-backward`	`<backspace>`	Delete character before cursor.
`delete-word-forward`	`C-Delete>, M-d`	Delete from cursor to end of word.
`delete-word-backward`	`C-<backspace>, M-<backspace>, M-<delete>`	Delete from cursor back to beginning of the word.
`kill-region`	`C-w`	Remove selected region and place in killring. (Modern *Cut*!)
`Copy-region`	`M-w`	Copy selected text to killring, does *not* alter buffer. (Modern *Copy*!)
`kill-line`	`C-k`	Kill from cursor to end of *line* and place in killring. Count N = 0 arg, Kill to beginning of line. Count N > 0 arg, Kill next N Lines. Count N < 0 arg, Kill prev N Lines.
`Kill-sentence`	`M-k`	Kill from cursor to end of sentence, and place in killring.
`kill-sexp`	`C-M-k`	Kill from cursor to end of SExp, and place in killring.
`yank`	`C-y`	Place the top killring entry in the buffer at the cursor. (Modern *Paste*!)
`yank-pop`	`M-y`	Replace yanked item with preceding one in killring. `M-y` can only be issued, and repeated, *immediately* after `C-y`.
`yank-from-list`	`C-M-y`	Displays pop-up of all killring entries. Select an entry to place in buffer at cursor.
`capitalize-word`	`M-c`	Capitalize word starting at cursor. (Knows Latin accented chars.)
`upcase-word`	`M-u`	Uppercase word starting at cursor. (Knows Latin accented chars.)
`downcase-word`	`M-l`	Lowercase word starting at cursor. (Knows Latin accented chars.)
`transpose-char`	`C-t`	Swap the character under the cursor with the one before it. Will swap the previous 2 characters if cursor is at end of line. Arg of N will move previous char (and cursor) N places forward. (Negative sign of N is ignored.)
`expand-ligatures`	`None`	Expands ligatures (such as Æ) in entire buffer or just selected text.
`stuff-tabs`	`DEBUG only`	Stuff multiple lines of `Tabs` into Buffer.
`stuff-text`	`DEBUG only`	Stuff multiple lines of generated text into Buffer.

Formatting:	scEmacs provides a number of commands for adding or deleting whitespace, consisting of space, tab, or linefeed (LF) characters. In *Text* mode, an insert-tab command simply adds a tab character, displayed as 1-8 blanks, advancing the column to a *tabstop* multiple of 8. Tabs are smarter in *Script* mode as they conform to *programming* indentation where tabstops are multiples of 4. Such an insert-tab command will first add enough spaces to advance to a multiple of 4. Then will go back to *tabify* the preceding spaces, turning as many spaces into tab characters as possible. If an insert-tab is entered in the leading whitespace of a line, it will add enough space to align the first text column with the *preceding* line (going up N lines to find a non-empty one if necessary) then tabify those spaces. Similarly, the insert-return command will add a linefeed to create a new line in *Text* mode. But in *Script* mode, it will also perform an insert-tab to align the new line with its preceding (perhaps N lines up if previous lines are blank). The indent-rigidly command performs en masse indentation of multiple lines. It is commonly used to re-indent selected sections of code that are being nested further in or out. Left and right arrows will move each line in the selected region one space, or one tabstop (multiples of 4 in *Script* and 8 in *Text* mode) if the the shift key is held down. Beyond individual lines, formatting text paragraphs is also important. scEmacs can flow or justify paragraphs to the desired fill column. This mechanism is even smart enough to allow for non-alphanumeric line leads, such as comment markers or other decorations.		
insert-tab	Tab	Inserts a Tab character at cursor. Given arg N, will place N spaces (not Tabs), ignores negative sign. *Script* mode: will match general programming indentation and will convert spaces in white area to Tab characters (*Tabify*). • Will space to match previous line if in leading whitespace. • Otherwise, will insert 4 spaces each time.
insert-return	<enter> <return>	Inserts a linefeed character at cursor. Given arg N, will place N linefeeds, ignores negative sign. *Script* mode: last linefeed will also get Tabs/Spaces to match prev line.
open-new-line	C-o	Given arg N, will insert N Return characters at cursor, ignores neg sign. Cursor position will not change.
join-lines	M-^	Merges lines by deleting whitespace (Space, Tab, Return) around cursor. Ignores numerical arg.
delete-horiz-space	M-\	Deletes Spaces and Tabs around cursor. If given a numerical arg, only deletes before the cursor.
indent-rigidly	C-x Tab	Indents every line in selected region, will *Tabify* as possible. If called with arg N, will move them N spaces (pos or neg) and exit. Otherwise, will enter *mini-mode* (like query-replace): • <Left> or <Right> will shift all lines 1 space as directed. • Shift+<Left> or <Right> will shift 8 spaces (4 if *Script*).
set-fill-column	C-x f	Sets internal FillColumn for fill-paragraph command. Will prompt for value if no numerical arg given.
fill-paragraph	M-q	Flows paragraph to FillColumn limit. Will *fully* justify text if given a numeric arg. Ignores selected region. Allows for whitespace and non-alphanumeric line leader in format.
test-justify	DEBUG only	Writes elaborate test of full justification to debug window (stdout).

Searching:	scEmacs provides mini-modes for incremental search and query replace.	
`search-forward`	`C-s`	Gets into forward *incremental-search* mini-mode, from cursor location: • Initial `C-s` prompts for search string on Echo Line. • Initial `C-s C-s` will load same search string as last time. • `C-w` (N times) after `C-s` will load next N words as search string. • Subsequent `C-s`, will move to the next match. • Subsequent `C-r` will switch to reverse search. • Can continue to type/delete search string on echo line. • Will wrap around buffer if continue to search.
`search-backward`	`C-r`	Gets into *reverse* incremental-search mini-mode, from cursor location. Same as `C-s`, only going backwards.
`query-replace`	`M-%`	Prompts user for *query-replace* mini-mode. (Can only go forward). Provides default pattern and replacement from last use. Finds the next match and prompts user for action: • `Y`, `y`, or `Space` to replace and continue. • `.` (period) to replace and exit. • `N`, `n`, `<delete>`, or `<backspace>` to skip and continue. • `!` to replace all remaining matches, then exit. • `I` or `i` to replace all remaining matches, show them, then exit. • `<Return>` to stop and exit.

Search, and by extension query-replace, is case insensitive *unless* one or more uppercase characters are included in the template string. So *MyName* will only match *MyName*, but *myname* will happily match *MyName*, *mYnAmE*, etc. In addition to base ASCII, scScript also understands *Basic Latin*, *Latin-1 Supplement*, *Latin Extended-A*, *Latin Extended-B*, and *Latin Extended Additional* characters in the Unicode definition (`0x0000` to `0x024F` and `0x1E00` to `0x1EFF` code points). scEmacs therefore understands that *É* is the uppercase form of *é* and acts accordingly.

Similarly, search (and query replace) will happily match base ASCII *root* characters against their accented brethren. So *entree* will match against *entree* or *entrée*. But if given the accented version as template, search (and query replace) will only find *identically* accented text in the buffer. In that sense, search is normally *accent* insensitive, unless one or more accented characters are included in the template string.

scEmacs has an *expand-ligatures* command, but search and query replace are currently not smart enough to match a ligature against its expansion, or vice versa. For example, matching *a+e* against *Æ* or *ß* against *ss* will fail.

Undo:	Every scEmacs buffer stores the sequence of individual changes made to it and can sequentially unwind and undo these changes. There will be memory limitations, but the system is efficient enough to provide ample undo capacity in most cases. Importantly, scEmacs will: • Delete to `undo` text addition. • Add to `undo` text deletion. • Often group char-by-char operations together for efficiency. The system will *not* restore cursor position after an undo, nor undo color changes.	
`undo`	`C-/, C-_, C-x u`	Reverse the last change made to the buffer!
`disable-undo`	None	Turns off `Undo` for buffer and releases its memory. Restore with `reset-undo`.
`reset undo`	None	Releases old Undo memory for buffer (if any), then creates anew.
`print-undo-memory`	DEBUG only	Prints image of Undo memory blocks to debug window (stdout).

Files & Buffers:	A text editor like scEmacs has to read disk files into memory buffers, display, edit, and otherwise process them, then write them back to disk files. scEmacs can handle multiple buffers and multiple views (panes) of such buffers at a given time. Only a few (`C-x`) commands are needed for all this.	
`find-file`	`C-x C-f`	Reads specified file into a new buffer and displays in current pane. Will prompt with directory (`pwd`) and allow entry/edit of filename.
`write-file`	`C-x C-w`	Writes and associates/names current buffer to specified file. Will prompt with directory (`pwd`) and allow entry/edit of filename. Will ask to create any missing directories in the specified path.
`save-file`	`C-x C-s`	Saves current buffer to its associated file. Calls `write-file` if buffer has no associated file.
`save-some-files`	`C-s s`	Goes through all modified buffers and saves their files.
`insert-file`	`C-x I`	Reads the file and inserts its contents into current buffer at cursor.
`pwd`	`None`	Displays the current directory in the echo line.
`switch-to-buffer`	`C-x b`	Prompts for the buffer name and switches current pane to it.
`buffer-from-list`	`C-x C-b`	Displays pop-up of all buffers, will switch current pane to selection.
`kill-buffer`	`C-x k`	Removes and deletes current buffer. All Panes displaying buffer will close or switch to another.
`read-only-mode`	`C-x C-q`	Toggles read-only status of current buffer.
`unmodify`	`M-~`	Resets buffer flags to *unmodified*. (`--` in mode line instead of `**`.)

Panes & Frames:	scEmacs supports multiple window *frames*, each showing one or more *panes*. Each pane displays and edits a single *buffer*. Only a single pane is ever active, and it resides on the frontmost frame window. `C-x 5` (*five* for *frame*) commands control frames.	
`split-pane`	`C-x 2`	Create a new pane by splitting the current one into two. Original pane stays at the top and remains active.
`other-pane`	`C-x o`	Activates the next pane on the frame instead of the current one.
`kill-pane`	`C-x 0`	Remove the current pane from the frame.
`kill-other-panes`	`C-x 1`	Removes all *other* panes on current frame.
`new-frame`	`C-x 5 2`	Creates a new frame with a single pane.
`other-frame`	`C-x 5 o`	Makes the next frame window active.
`kill-frame`	`C-x 5 0`	Closes the current frame, but only if there are others.
`kill-other-frames`	`C-x 5 1`	Closes all other frames, if there are any.

Color:	scEmacs can display text in a limited (but easily expanded) color palette. Importantly, *Script* mode will color source code on the fly using a simple (fast) parser. These colors are for display only as text files have no reliable way of storing color or other meta data. Each buffer can have its own color memory which stores color transitions. So a single black-to-red transition mid paragraph would mean all subsequent characters will be drawn in red. Importantly, the same buffer will be displayed with the same colors on all panes. Available colors are: black, gray, red, crimson, orange, brown, green, blue, purple, or yellow. (scEmacs does *not* include color changes in the undo buffers.)	
`set-color`	`none`	Sets the text color for the selected text region. If no region, sets color from cursor to the next color change, if any. Prompts in the Echo Line to specify color.
`clear-all-color`	`none`	Removes *all* color specifications and resets color memory.
`print-color-memory`	`DEBUG only`	Prints out image of color memory to debug window (stdout).

Script:	scEmacs is the main data and control interface to scScript. In a typical scenario, script code is written in scEmacs, fed to scScript for compilation, and then executed. The output of the script goes to the *O*utput** buffer, and is displayed on an scEmacs pane in realtime. Scripts can be read from an scEmacs buffer or directly from file: • `script-do` reads the entire *buffer* and execute the code. • `script-compile` reads the entire *buffer*, compiles it, and write a `.bsc` binary file. • `script-run` reads the .bsc (or just .sc) from the named *file* and executes it. Both `script-do` and `script-run` (from .sc file) will compile the sources in place. But `script-compile` will write out the binary `.bsc` file, so it can be read by `script-run` and executed immediately without invoking the compiler again. Scripts may also be run *interactively* with `script-try`. This will only compile/execute the *selected* region, which should be syntactically complete and stand-alone top-level *chunk*. Importantly, such interactive execution will retain script *state* between runs, allowing chunks to use previously defined variables and functions. When all is done, `script-reset` will wipe state information, ready for another clean start. (Most script commands will automatically `reset` before executing.) Normally, the script will write to its default *output* buffer which is automatically displayed at execution time. `script-show-output` and `script-wipe-output` will display and clean this buffer on command. As a normal buffer, *output* can also be written/saved to a file. The user may even designate any existing buffer as *the output* using `script-set-output`. While running, a script may execute a S*ys.Pause* function to return control to scEmacs. This is normally done to allow user housekeeping on files and buffers. When completed, the `script-resume` command will return control back to the script while `script-exit` will cancel and terminate the *paused* script.	
`script-mode`	none	Sets current buffer to *Script* mode, auto-scans and colorizes. If already in Script mode, *removes color* and returns to Text mode.
`text-mode`	none	Sets current buffer to *Text* mode, leaves color information intact.
`script-do`	none	Read script source code in buffer, compile, and execute.
`script-compile`	none	Read script source code in buffer, compile, and write .bsc binary file.
`script-run`	none	Prompts in echo line to specify .sc or .scb file to be directly read, compiled (.sc only), and executed. No scEmacs buffers are involved!
`script-try`	none	Read, compile, and execute selected text region (*chunk*) in *interactive* mode. Script state will be maintained to *try* the next chunk.
`script-reset`	none	Clear out *interactive* script state.
`script-set-output`	none	Makes current buffer the Output for Script *write* instructions.
`script-show-output`	none	Make script *Output* buffer visible if not so: • Selects another frame if it shows *Output* or • Adds an *Output* pane to the current frame.
`script-wipe-output`	none	Cleans out the *Output* buffer and shows it.
`script-new-output`	none	Creates new *Output* *if* existing one is associated with a disk file. Otherwise, does a `script-wipe-output`. ****
`script-exit`	none	Exits (termiantes) *paused* script.
`script-resume`	none	Resumes *paused* script.
`script-vm-init`	none	Initialize scScript memory. Done automatically when scEmacs starts.

`script-vm-kill`	`none`	Clean and delete script memory. Must initialize before running another script.
`script-vm-stats`	`none`	Display (in `*Output*`) information about script memory.
`script-test-dict`	`DEBUG only`	Stress-test for DICT mechanism in scScript, displays on *Output*.
`script-test-string`	`DEBUG only`	Stress-test for STR mechanism in scScript, displays on *Output*.

**** The *Output* buffer is just another scEmacs buffer. `C-x C-w` will *associate* *Output* with a disk file, so scScript output can be saved to file. But it is impossible to *unassociate* a buffer from a disk file. Instead, `script-new-output` will create a new pristine (*unassociated*) *Output* buffer. Importantly, one can associate (`C-x C-w`) the new *Output* buffer with the *same* disk file again, in the process getting multiple buffers pointing to the same disk file. Sadly these buffers will happily over-write each other's disk files! scEmacs does its best to empower the user, but cannot keep one's metatarsals safe from target practice.

`Script-wipe-output` will empty out the *Output* buffer *even if* associated with a disk file. So `script-new-output` should be used to get a *new* buffer if the intention is to keep the old output around. The latter command may display the new *Output* buffer on the Pane, in place of the old one, but the old buffer will remain accessible in scEmacs (using `C-x C-b` or `C-x buffer-from-list` to show a list of buffers).