The Script Programming Language

Author: Jeff Stephenson
Date: 4 April 1988

Table of Contents

Introduction

The Script adventure game language is an object-oriented language with a Lisp-like syntax. It is compiled by the sc compiler into the pseudo-code which is used by the interpreter, sci.

We will begin our discussion of the language with its basic Lisp-like characteristics, then go on to the object-oriented parts of the language.

As is Lisp, Script is based on parenthesized expressions which return values. An expression is of the form

```
(procedure [parameter parameter ...]).
```

The parameters to a procedure may themselves be expressions to be evaluated, and may be nested until you lose track of the parentheses.

Unlike Lisp, the procedure itself may NOT be the result of an evaluation. An example of an expression is

```
(+ (- y 2) (/ x 3))
```

which would be written in infix notation as

```
(y - 2) + (x / 3).
```

All expressions are guaranteed to be evaluated from left to right. Thus,

```
(= x 4)
(= y (/ (+= x 4) (/= x 2)))
```

will result in y = 2 and x = 4.

Comments in Script begin with a semi-colon, ';', and continue to the end of the line.

Files

Source files for the script compiler have the extension .sc, header (include) files have the extension .sh. Source files may have any filename -- banner.sc and castle.sc are two examples. The output file from the compilation will have the name script.nnn where nnn is determined from the script# command (covered below) which is present in the file.

There are six files besides the source file and any user-defined header files which are involved in a compilation.

classdef
: This file contains the information about the structure of the classes which have been defined in the application. It is read automatically by the compiler and is rewritten by the compiler after a successful compilation in order to keep it up to date. The user need not be concerned with it.

selector
: This contains definitions of selectors which are used in object-oriented programming. It is automatically included in a compile and, like classdef, is rewritten after a successful compile. Any symbol in a properties or methods statement or in the selector position in a send to an object is assumed to be a selector and is assigned a selector number in included in selector.

kernel.sh
: This contains the definitions for interfacing with the kernel (the machine language interpreter). It is maintained by the kernel programmers and is automatically included in system.sh.

system.sh
: This contains the definitions for interfacing with the various system classes. It is initially provided by the kernel programmers. If you wish to tweak the system scripts yourself, you will also be responsible for maintaining your copy of system.sh. It should be included in all compiles.

vocab.000
: This is the compiled output of vocab.txt, generated by the vocabulary compiler vc. It is automatically included in a compile.

classtbl
: This is an output file of the compiler which is used by the kernel to determine which script a given class is defined in. You needn't do anything to it other than not delete it.

There are two sc commands for dealing with source code organization:

script#:

The script# command sets the script number of the output file:

```
(script# 4)
```

sets the output file name to script.004, regardless of the actual name of the source file.

include:

This includes a header file in the current source file at the current position.

```
     (include "/sc/foo.sh")
or
     (include /sc/foo.sh)
```

include the file /sc/foo.sh. Include files may be nested as deeply as desired.

When including a file, the compiler first looks for the file in the current directory. If it fails to find it there, it then looks for it in the directories specified in the environment variable SINCLUDE. This variable is just like the DOS PATH variable -- the directories to search are separated by semi-colons. Thus, if you want the compiler to look for include files in f:/games/sci/system and c:/include if it doesn't find them in the current directory, you put the line

```
     set sinclude=f:/games/sci/system;c:/include
```

in your autoexec.bat file.

Definitions

define:

The define statement allows you to define a symbol which will stand for a string of text:

```
(define symbol lots of text)
```

will replace symbol, wherever it is encountered as a token, with lots of text and then continue scanning at the beginning of the replacement text. Thus, if we write

```
(define symbol some text)
(define some even more)
```

then

```
(symbol)
```

will become

```
(some text)
```

which then becomes

```
(even more text)
```

enum:

A construct for easing the definition of various states of a state-variable is enum. Say you want to walk an actor from the door of a room across the floor, up the stairs, and through another door. You have a state-variable called actor-pos which will take on a number of values, which could be defined with defines:

```
(local    actor-pos
     (define at-front-door    0)
     (define in-room          1)
     (define on-stairs        2)
     (define top-of-stairs    3)
     (define upper-door       4)
)
```

or you could get the same result with enum:

```
(local    actor-pos
     (enum
          at-front-door
          in-room
          on-stairs
          top-of-stairs
          upper-door
     )
)
```

Enum defaults its first symbol to 0. If you want a different starting value, put it right after the word enum:

```
(enum 7
     at-front-door
     in-room
     on-stairs
     top-of-stairs
     upper-door
)
```

sets at-front-door to 7, in-room to 8, etc.

synonyms:

The synonyms statement defines synonyms of words. All words must have been defined in the vocabulary file (see separate Vocabulary documentation). The statement

```
(synonyms
     (main-word  synonym1 synonym2 ...)
     ...
)
```

defines the words synonym1, synonym2, etc. to be synonyms of main-word. In input being interpreted by the script in which the synonym statement is defined, user input of synonym1 will be interpreted as if the user had typed main-word.

Data Types and Variables

Numbers:

All numbers in Script are 16 bit integers, giving a range of -32768 to +32767. Numbers may be written as decimal (1024), hex ($400), or binary (%10000000000).

Variables:

Variables hold numbers. Variables can be either global, local, or temporary, depending on when they are created and destroyed:

Global variables are created when the program starts and destroyed when it ends, and are thus accessible to all scripts at all times.

Local variables are created when a logic script is loaded and destroyed when it is purged. They are thus only available when the logic script is loaded and will not retain a value through a purge-reload cycle. You will find that, as your programming takes on a more object-oriented flavor, you will use fewer and fewer local variables.

Temporary variables are created when a procedure or method is entered and destroyed when it is left. They are thus only available to the declaring procedure and do not retain a value between calls to the procedure.

In order to throw the 'link' out of the traditional 'edit-compile-link-test' cycle of software development, YOU, rather than the linker, must define the address (i.e. variable number) of global variables. This is done with the global definition:

```
(global
     var-name var-number
     var-name var-number
     ...
)
```

This defines var-name to be global variable number var-number.

Local variables, not being accessible outside of the scripts in which they are declared and thus not requiring linking, can have their addresses set by the Script compiler. There are two ways of defining locals:

```
(local
     var-name
     var-name
     ...
)
```

defines a single variables with the names var-name.

```
(local [array-name n])
```

defines an array of n elements with the name array-name (the brackets in this do NOT mean 'optional' -- they are required).

Multiple local variable definitions may be combined in one statement:

```
(local
     var1
     [array1 10]
     [array2 5]
     var2
          .
          .
          .
)
```

Temporary variables will be discussed in the section on user-defined procedures.

Define and enum statements may be included within both global and local variable definitions.

Arrays:

To access element n of the array anArray, write

```
[anArray n]
```

Despite the syntactic difference between local variable declarations and local array declarations, there really is no distinction between variables and arrays -- any variable may be indexed as an array. Thus, if we have the local variable declarations

```
(local
     var1
     var2
     var3
     var4
)
```

we can set the value of var1 to that of var4 by any of the following statements:

```
(= var1 var4)
(= var1 [var2 2])
(= var1 [var3 1])
(= [var2 -1] [var1 3])
```

The first method is obviously the preferred method for clarity, but this array property of all variables allows access to variable numbers of parameters in a user-defined procedure (see section on user-defined procedures).

This property of variables is also the basis of the method by which you declare global arrays -- you simply leave an array-sized gap in the global variable numbering sequence. To declare var2 as a global array of 10 elements, write

```
(global
```

```
          var1      23
          var2      24
                         ;10 element array
          var3      34
     )
```

and access var2 as an array:

```
     [var2 7]
```

Pointers:

Some kernel calls require pointers to variables, rather than the value of a variable. A pointer to a variable is created by preceding a variable reference with the '@' sign. Pointers may be created to array elements as well as to simple variables:

```
          @ego           ;pointer to the variable ego
          @[foo 3]       ;pointer to fourth element of array foo
```

Since there is currently no way in sc to dereference a pointer, this is only useful for passing pointers to kernel calls.

Text:

Text strings are strings of characters enclosed in double quotes, and may be used anywhere you like:

```
     (Print "This is immediate text.")
```

prints the text string,

```
     (= textToPrint "This text is referenced through a variable.")
```

sets the variable to a pointer to the text string, and

```
     (instance foo of Bar
          (properties
               name:"fooBar"
          )
     )
```

sets the name property of foo to be a pointer to the text string.

When sc goes to squirrel a text string away, it first checks to see if it has seen the string before. If so, it just uses the previous text, rather than duplicating the text. For long text strings which are used in several places, however, the likelihood that you will manage to type the text identically in each case is small. In this case you can simply put the text in a define statement

```
     (define lotsOftext  "This is a long text string. I am using a
                          define statement to avoid having to type
                          it repeatedly.")
```

This introduces another aspect of text strings: If text is too long fit on a single line, you may enter it on several lines. Multiple white-space (spaces, tabs, and newlines) gets converted to a single space, so the text above ends up with just one space between the words on each line. If you want multiple spaces, enter them as underbars, '_'. These are converted to spaces in the string, but are not compacted.

To include a '_' in text, type '_', where '\' is the escape character. Explicit newlines are entered just as in C: '\n'. A CR/LF pair is entered as '\r' (the '\r' should be used in place of '\n' in all strings destined for a file). Characters which are not on the keyboard, but are defined in a font (such as the Sierra symbol in the menubar) can be included in the string by preceding the two-digit hex value of the character with the '\'. Thus, "This is the Sierra symbol: \01" would put the value 1 at the end of the string, and this character in the font is the Sierra symbol.

The maximum length of a text string is 2000 bytes.

Word-strings:

Word-strings are used to represent templates for user input in Said statements. A word-string is a string enclosed in single quotes which contains meta-characters describing the content of a sentence. The meta-characters and their meanings are described in the separate Vocabulary documentation.

```
(if (Said 'give/pirate/gold coins<#' @howMany)
     (Print "Get lost creep.")
)
```

As with text strings, identical strings are stored only once.

Characters:

Characters are single ASCII characters, and are denoted by preceding the character with the reverse single quote ("tick") character:

```
`A   represents uppercase A and
`?   represents the question mark
```

Several character sequences represent special key combinations:

```
`^a  represents ctrl-A
`@b  represents alt-B
`#4  represents the F4 key
```

Literal selectors:

Sometimes, as in the code

```
(cast eachElementDo: #showSelf:)
```

you want to send the value of selector rather than use the selector as the start of another message to an object (these terms will be described in Object Oriented Programming in Script). Preceding the selector with a '#' produces the literal value of the selector rather than using it as a message.

Primitive Procedures

Arithmetic primitives:

In the following, e1, e2, ... are arbitrary expressions.

(+ e1 e2 [e3...])
     Evaluates to e1 + e2 [+ e3 ...]

(* e1 e2 [e3...])
     Evaluates to e1 * e2 [* e3 ...]

(- e1 e2)
     Evaluates to e1 - e2

(/ e1 e2)
     Evaluates to e1 / e2

(mod e1 e2)
     Evaluates to the remainder of e1 when divided by e2.

(<< e1 e2)
     Evaluates to e1 << e2 where the << operation shifts its left hand side left by the number of bits specified by its right hand side. (As in C).

(>> e1 e2)
     Evaluates to e1 >> e2 as in << except a right shift.

(^ e1 e2 [e3 ...])
     Evaluates to e1 ^ e2 [^ e3 ^ ...] where '^' is the bitwise exclusive-or operator.

(& e1 e2 [e3 ...])
     Evaluates to e1 & e2 [& e3 & ...] where '&' is the bitwise and operator.

(| e1 e2 [e3])
     Evaluates to e1 | e2 [| e3 | ...] where '|' is the bitwise or operator.

(! e1)
     Evaluates to TRUE if e1 == 0, else FALSE.

(~ e1)
     Evaluates to the bit-wise not of e1, i.e. all 1 bits are changed to 0 and all 0 bits are changed to 1.

Boolean primitives:

These procedures are always guaranteed to evaluate their parameters left to right and to terminate the moment the truth value of the expression is determined. If the truth value of the boolean is determined before an expression is reached, the expression is never evaluated.

(> e1 e2 [e3...])
    Evaluates to TRUE if e1 > e2 [> e3 ...], else FALSE.

(>= e1 e2 [e3...])
    Evaluates to TRUE if e1 >= e2 [>= e3 ...], else FALSE.

(< e1 e2 [e3...])
    Evaluates to TRUE if e1 < e2 [< e3 ...], else FALSE.

(<= e1 e2 [e3...])
    Evaluates to TRUE if e1 <= e2 [<= e3 ...], else FALSE.

(== e1 e2 [e3...])
    Evaluates to TRUE if e1 == e2 [== e3 ...], else FALSE.

(!= e1 e2 [e3...])
    Evaluates to TRUE if e1 != e1 [!= e3 ...], else FALSE.

(and e1 e2 [e3...])
    Evaluates to TRUE if all the expressions are non-zero, else FALSE.

(or e1 e2 [e3...])
    Evaluates to TRUE if any of the expressions are non-zero, else FALSE.

(not e)
    Evaluates to TRUE if the expression is zero, else FALSE.

Assignment primitives:

All assignment procedures store a value in a variable and return that value as the result of the assignment. In the following, v is a variable and e an expression.

```
(= v e)
     v = e

(+= v e)
     v = v + e

(-= v e)
     v = v - e

(*= v e)
     v = v * e

(/= v e)
     v = v / e

(|= v e)
     v = v | e

(&= v e)
     v = v & e

(^= v e)
     v = v ^ e

(>>= v e)
     v = v >> e

(<<= v e)
     v = v << e

(++ v)
     v = v + 1

(-- v)
     v = v - 1
```

Control Flow

In the following, code1, ..., codeN are arbitrary sequences of expressions. There are no BEGIN ... END blocks as in Pascal or progn forms as in Lisp.

The value of a control flow expression is the value of the last expression in the control body which was evaluated. Thus, if we execute the following code:

```
(= x 3)
(= y 2)
(= y (if (> x y)
         (- x y)
     else
         (+ x y)
     )
)
```

y will have the value 1.

Return:

```
(return [expression])
```

The return statement returns control to the procedure which called the currently executing procedure. If the optional expression is present, that value is returned as the value of the current procedure. There is an implicit return at the end of all procedures, and the value returned in that case is the value of the last expression evaluated. A return from the main procedure of script 0 returns to the operating system.

Conditionals:

(if expression code1 [else code2])

If expression is not FALSE, execute code1, else execute code2. (The else clause is optional).

(cond (e1 code1) (e2 code2) ... [(else codeN)])

Evaluate e1. If it is not FALSE, execute code1 and exit the cond clause. If it is FALSE, evaluate e2 and continue. If all of the expressions are FALSE and the optional else clause is present, execute codeN.

(switch expression (exp1 code1) (exp2 code2) ... [(else codeN)])

Evaluate expression. If it is equal to exp1, execute code1 and exit the switch. If it is equal to exp2, execute code2 and exit. If it doesn't equal any of the expressions and the optional else clause is present, execute codeN.

Iteration:

(for (initialization) condition (re-initialization) code)

Evaluate the expressions comprising initialization. Then evaluate condition. If the result is FALSE, exit the loop. Otherwise, execute code, then the expressions comprising re-initialization, and loop back to condition.

(while condition code)

Evaluate condition. If not FALSE, execute code and loop back to evaluate condition again. Exit the loop when condition is FALSE. (Note that this means that the value of a while condition is always FALSE.) This is equivalent to

```
(for () condition () code)
```

(repeat code)

Continually execute the code until some condition in the code (a break) causes the loop to be exited. This is equivalent to

```
(while TRUE code)
```

or

```
(for () TRUE () code)
```

Supporting constructs for iteration:

(break [n])

Break out of n levels of loops. If n is not specified break out of the innermost loop.

(breakif expression [n])

If expression is not FALSE, break out of n levels of loops. If n is not specified, break out of the innermost loop.

(continue [n])

Loop back to the beginning of the nth level loop. If n is not specified, loop to the beginning of the innermost loop.

(contif expression [n])

If expression is not FALSE, loop back to the beginning of the nth level loop. If n is not specified, loop to the beginning of the innermost loop.

Procedures

Procedures are created with the procedure construct:

```
(procedure (proc-name [p1 p2 ...] [&tmp t1 t2...])
     code
)
```

This defines the procedure with the name proc-name. This procedure takes parameters p1, p2, ... and allocates temporary variables (which disappear on exit from the procedure) t1, t2, ... Note that the procedure may take no parameters and have no automatic variables. In this case, the definition would be

```
(procedure (proc-name)
     code
)
```

You can define temporary arrays in the same way as you would local arrays:

```
(procedure (proc-name &tmp [array n])
     code
)
```

Code in these examples is any list of valid expressions.

All procedures have at least one parameter, the compiler defined variable argc (argument count), which gives the number of parameters passed to the procedure.

For example, you might define the procedure square, to square a number, as follows:

```
(procedure (square n)
     (* n n)
)
```

or the procedure max to find the maximum of an arbitrary number of numbers passed to the procedure:

```
(procedure
     (max
          p         ;parameters (will be accessed as an array)
          &tmp
          biggest   ;temporary variable containing maximum
          i         ;index into parameter array
     )

     (for ((= i 0) (= biggest 0))
          (< i argc)     ;compare to number of parameters passed.
          ((++ i))       ;note that this is a LIST of expressions,
                         ;not a single expression.

          (if (> [p i] biggest)
               (= biggest [p i])
```

```
            )

        )

        (return biggest)
    )
```

The call

```
    (max 3 -4 -9 0 -2 7 12 4 3 5)
```

will return the value 12.

In order to use a procedure before it has been defined in a source file (for example making a call to max before the actual definition of max), the compiler must be told that the procedure's name corresponds to a procedure, not an object. This is done with another form of the procedure statement:

```
    (procedure
        procedure-name
        procedure-name
        ...
    )
```

Tells the compiler to compile code for procedure calls when it encounters procedure-name, rather than code for send messages to an object.

&rest:

Argc, as discussed above, makes it easy to write procedures (or methods, discussed in Object Oriented Programming in Script) which handle a variable number of arguments. If a procedure or method has received a variable number of arguments and wants to pass them on to another procedure or method, things get messy. The only way to do this given only argc is to build a switch statement on argc which looks like

```
    (procedure (foo arg)
        (switch argc
            (0   (mumble))
            (1   (mumble arg))
            (2   (mumble arg [arg 1]))
            (3   (mumble arg [arg 1] [arg 2]))
            ...
        )
    )
```

This not only involves a lot of typing and generates a lot of code, it limits the number of arguments which can be passed to the next function (you can only type a finite number of clauses in the switch statement).

&rest exists to solve this problem -- it stands for the rest of the parameters not specified in the procedure or method definition. The above procedure could then be written simply as

```
    (procedure (foo)
```

```
          (mumble &rest)
     )
```

which is not only easier to type and read but also produces smaller, faster object code.

A variant of &rest allows you to specify all parameters starting at any point in a parameter list of a procedure or method definition:

```
     (procedure (foo arg1 arg2 arg3 arg4)
          (mumble (&rest arg3))
     )
```

passes all arguments starting with arg3 to procedure mumble.

Extern:

Calling a procedure in another script is another matter. Since there is no link phase in the development cycle, one procedure cannot know the address of a procedure in a different script. The extern statement allows a script to know where the external procedure is:

```
     (extern
          procedure-name script-number entry-number
          ...
     )
```

This says that the procedure referred to by the symbol procedure-name in this script is to be found in script number script-number at entry number entry-number in the script's dispatch table. kernel.sh and base.sh both use the extern statement to let all other scripts know where their public procedures are.

The dispatch table for a script is defined by the public statement:

Public:

All procedures within a script which are to be accessed from outside the script must be entered in the dispatch table for the script with the public statement:

```
     (public
          procedure-name entry-number
          ...
     )
```

puts the procedure procedure-name in the dispatch table at entry number entry-number. The entries need not be in numeric order, nor do the numbers need to be continuous (though if they're not continuous the table will be larger than it needs to be).

Using SC

The sc compiler is invoked with the command

```
sc file_spec [file_spec] [options]
```

Any number of file specifications may be entered on the command line, and a file specification may include wild-card names.

Options are:

-l
: Generate an assembly language code listing for the file. This is useful when using the built-in debugger of sci, which lists only the assembly language code, not the source. When compiling filename.sc, the list file is named filename.sl

-n
: Turns off 'auto-naming' of objects. As described in Script Classes for Adventure Games, each object has a name, or 'print-string' property, which is how to represent the object textually. Unless the property is explicitly set, the compiler will generate the value for this property automatically, using the object's symbol string for the name. The object names, however, take up space in the heap. While they are useful (almost vital) for debugging, if you're running out of heap in a room, it might help to compile with the -n option to leave the names out.

-oout-dir
: Set the directory for the output file (script.nnn) to out-dir.

-v
: Turns on verbose mode, which prints the number of bytes occupied by various parts of the output file (code, objects, text, etc.).

-z
: Turn off optimization. Not a particularly useful option except for those of us who must maintain the compiler.

Index