ES: Abridged Reference

This is a tidied-up, rationalized, cleaned-up and severely abridged description of ES. It was extracted from the man page in order to provide essential information on the language in a more precise and systematic manner, but with less detail. One still needs the man page for the details

Overall Structure

A *script* consists of commands.

A *command* is a sequence of words terminated by newline, semicolon (`;`), or ampersand (`&`). A *word* is either a string or a program fragment, and is terminated by white space or any of the *special characters* `$ & ' \ # ^ < = > ; ` { | }`. A *program fragment* is a command or a sequence of commands enclosed in `{` and `}`.

Multiple commands within a fragment are separated by `;` or newlines. Thus grouped, the output of all of them may be redirected to the same file.

Strings can also be enclosed in a pair of single quotes (`'`). Characters between single quotes, including special and control characters, are left uninterpreted. A `'` in a `'`-enclosed string is quoted by `''`. The string `''` is the empty string.

The backslash (\) quotes the immediately following character if it is special except for newline. The backslash-newline is treated as a space and thus continues a line. `\a`, `\b`, `\e` (escape), `\f`, `\n`, `\r`, `\t`, `\x`*dd* and `\`*ddd* are treated like in C.

The first word in a command designates what has to be executed: an ES function or the name of a host system file, containing a program. The remaining words are passed as arguments. The first word of a command can also be a fragment, in which case it is executed ignoring the remaining words, except when that fragment is a 'lambda': lambdas take the remaining words as arguments (see *Functions*).

A command ending with `&` is executed in the background.

The number sign (`#`) begins a comment ending at (but not including) the next newline.

Values and Expressions

Many commands operate on list data. A *list* is a sequence of words (thus, syntactically, commands are lists). A single word is a list of length 1, so all values are treated as lists. If necessary, parentheses are used to enclose list contents. The empty list is represented by `()` and has length 0. Lists are flat, non-hierarchical: 'nesting' a list into a list expands its contents rather than actually nesting it. E. g., `((a b) () c)` is the same as `a b c`.

A program fragment, because it is a word, is a value that may be passed as an argument, stored in a variable, written to a file or a pipe etc.

The 'concatenation' (`^`) applies to two or more lists and is actually a Cartesian product: `x^(1 2 3)` produces `x1 x2 x3`, and `(x y)^(1 2 3)` produces `x1 x2 x3 y1 y2 y3`.

If a word or a keyword is immediately followed by another word, keyword, `$` or `` ` `` without intervening spaces, then `^` is automatically inserted between them.

The `$^` operator creates a single-element list from the items of its argument, separating them with spaces.

When a value is interpreted as a truth value, and each of its items, as a list, equals `0` or `''` (that includes the empty list), the value is considered 'true', otherwise it is 'false'.

The `` ` `` operator, as in `` `{`*command*`} `` or `` `*command* ``, returns a list formed by splitting up the standard output of *command* into separate items, using whitespace as delimiters. (The second form is applicable when the command is given by a single string.) The `` `` `` operator does a similar job, but makes it possible to specify explicitly the set of characters to be used as delimiters.

The `<=` operator, as in `<={`*command*`}`, gives the return value of a command, in particular – the exit status of an external program.

The return value of an assignment operation is the assigned value.

The operators `&&` (and), `||` (or), and `!` (not) are used for conditional execution of commands. They take as arguments return values which they interpret as truth values. *command*$_1$ `&&` *command*$_2$ executes *command*$_2$ only when *command*$_1$ is true. Similarly, *command*$_1$ `||` *command*$_2$ executes *command*$_2$ only when *command*$_1$ is false. `!` inverts the truth value of a command.

Variables

Assigning has one of the forms

varname `=` *list*

or

(*list-of-varnames*) `=` *list*,

where in the latter form the variables take values from *list*, respecting the order of the two lists. If there are more variables than values, the excess variables remain undefined. Otherwise, the last variable collects the remaining *list* value(s).

A variable name is either a sequence of non-special characters that does not represent a number, or a quoted string. A variable name, as well as a list of variable names, can be produced by an expression. In such a case, a variable name may have to be enclosed in `(` and `)`.

An undefined variable has `()` as a value, and explicitly assigning `()` is regarded as undefining. The latter can also be done by 'assigning nothing': `a =`.

The `$` is a *dereferencing* operator. It can be applied to a single variable name or to a list of such names, and more than once in a succession. When applied to a list, `$` produces a list from all the applications of `$` to the individual variables in the original list. E. g., `(x y z) = a b c d e; echo $(x z)` prints `a c d e`.

Dereferencing a variable whose value is a fragment leads to executing the fragment.

The `$#` operator applies to a variable name and returns the length of the corresponding value as a list.

Variables may be indexed with the notation `$`*var*`(`*I*`)`, where *I* is a list of integers or ranges, possibly produced by an expression. Subscripts are based at 1. Those referring to nonexistent elements expand to `()`. Subscripts and ranges can appear in any order and need not be unique.

A *range* has the form *m* `...` *n*, where either *m* or *n* can be omitted, defaulting to start and end of the list.

The name `*` refers to the list of arguments to a function or script. `$`*n*, where *n* is an integer, is a shorthand for `$*(`*n*`)`. `* = $*(2 ...)` is similar to `shift` in SH.

By default, all user-defined variables are exported into the environment.

Locality

Variable assignments may be made local to a command or to a set of commands with the construct

`local (`*var* `=` *value*`;` *var* `=` *value* ...`)`
  command

where *command* may be a fragment. The construct

`let (`*var* `=` *value*`;` *var* `=` *value* ...`)`
  command

introduces a lexical scope.

A reference textually from within a command to a variable defined in a `let` of which that command is the body refers to the `let`-bound variable even when dereferencing takes place in an execution context outside of the `let`, i. e. a `let` can export closures. In contrast, a reference from within the body of a `local` only refers to the `local`-bound variable when dereferencing takes place within the execution of that `local`; outside of it, the same name refers to a variable defined elsewhere. For that reason, `local` binding is also called 'dynamic'.

`local`-bound variables are exported into the environment, and changing one will invoke the settor function (see the *Functions* section) with the corresponding name. Lexically bound variables are not exported into the environment, and do not cause the invocation of settor functions.

Pattern Matching

`~` *subject pattern pattern* ...

matches a value against patterns, including wildcards (`?`, `*`, `[ ]`, lists), returning 0 (true) if the subject matches any of the patterns, and 1 (false) otherwise. The subject itself can be a list; then for a success it suffices any item of the list to match.

If the subject contains metacharacters, it is expanded against filenames. E. g., `~ * ?` returns true if any file in the current directory has a single-character name.

`~~` *subject pattern pattern* ...

returns the parts of the subject that match patterns, e. g.

`~~ (foo.c foo.x bar.h) *.[ch]`

is the list `(foo c bar h)`.

Sequencing Control

Command execution sequencing is controlled through built-in commands.

`if` *test-command then-command*
  test-command then-command
  ...
  else-command

evaluates the test commands in order, and runs the first *then-command* for which the test is true. If none of the tests is true, the *else-command* is run, if present.

`while` *test command*
evaluates *test* and, if it is true, runs *command* and repeats.

`forever` *command*
runs *command* repeatedly until it raises an exception or ES exits.

`for (`v_1`=`$list_1$`; `v_2`=`$list_2$`; ...)` *command*
runs *command* while lexically binding the variables, in parallel, to items of the corresponding lists, until all lists get exhausted. The `for` command is special in that it has special syntax rather than being implemented as a primitive (see *Primitives*) and having a hook attached to it (see *Hooks*).

`break` *value*
exits the current loop, returning *value*.

`eval` *list*
concatenates the elements of *list* with spaces and feeds the result to the interpreter for rescanning and executing as a command.

`.` *options file args*
reads *file* as input to ES and executes its contents. During execution, the `0` variable is set to the name of the file.

`exec` *command*
replaces ES with *command*, or, if it contains only redirections, applies them to the current shell.

`exit` *status*
causes ES to exit with *status*. Default is 0.

`fork` *command*
runs *command* in a subshell. State-changing operations within the subshell (`cd`, variable assignments) have no effect on the parrent.

`wait` *pid*
waits for *pid*, or for any child process, to exit.

There are some built-in commands for test formation. The operators `&&`, `||`, and `!` (see *Values and Expressions*), the pattern matching command (`~`, see *Pattern Matching*), and external commands such as `test` are also relevant.

`true`
returns 0.

`false`
returns 1.

`access` *options paths*
tests if the named paths are accessible according to the options presented. The default test is whether a file exists.

Exception Handling

Exceptions are represented by lists. The first word of an exception is treated as the type of exception being raised.

`throw` *exception arguments*
raises *exception*, passing *arguments* to the handler.

`catch` *handler command*
runs *command*, and if it raises an exception, runs *handler* passing the exception as an argument.

`retry`
is an exception. Raised from a handler, causes the command of the `catch` to be run again.

`unwind-protect` *command cleanup*
runs *command* and, after it completes normally or raises an exception, runs *cleanup*.

`eof`
is an end-of-file exception.

`error`
is a general-type exception.

`signal` *name*
is an exception raised when the shell itself receives a signal, and the signal is listed in the `signals` variable. *name* is the name of the signal that was raised.

`break` (see *Sequencing Control*) and `return` (see *Functions*) are also exceptions.

Functions

Anonymous functions (*lambdas*) are specified by a special form of a program fragment:
`@` *parameters* `{`*commands*`}`.
The parameters are variable names, to which values are assigned when the lambda is called as a command, similar to the way variable names within a list get bound to a list of values (including parameter and argument lists of different lengths). Lambdas are different from the other fragments in that when a lambda is the first word in a command, its execution does not ignore the rest of the words but takes them as arguments.

The syntax form
`fn` *name parameters* `{`*commands*`}`
is used to introduce a function and is a shortcut for
`fn-`*name* `= @` *parameters* `{`*commands*`}`,
i. e. it creates a variable named `fn-`*name* whose value is a lambda. If a name for which

a `fn-` variable exists is the first word of a command, the value of the `fn-` variable is substituted for that word, and the parameters are bound to the following words. The variable `0` is bound to the name of the function.

The parameters of a lambda (or a function) are lexically bound.

Functions may be removed with the syntax `fn` *name*, which is a shortcut for the assignment `fn-`*name* `=`.

Within a function, `return` *value* causes the current function to exit, returning *value*.

A variable whose name is of the form `set-`*var* is treated as a function (a '*settor function*') which is invoked whenever assigning to *var* takes place. *var* must not be a lexically bound variable. The parameters are bound to the value involved in the assignment, and the actual value being assigned is the result of the invocation. While the settor function is running, the variable `0` is bound to the name of the variable being assigned to.

`result` is ES's identity function: it returns a list of whatever arguments it is given. In practice, `result` is useful wherever a command is needed to produce a certain value, such as in

```
{if {...} {result ...}
    {...} {result ...}
                  ...}
```

Introspection

`var` *variables*
prints the definitions of *variables* in a form suitable for use as ES input.

`vars` *options names*
prints the program objects with the listed names in a form suitable for use as ES input. *options* select classes of objects: variables, (ordinary) functions, settor functions, exported or private, built-in. Exported variables is the default.

`whatis` *programs*
for each named program, prints the pathname, primitive, lambda, or code fragment which would be run if the program appeared as the first word of a command.

I/O

`echo` *value*
prints *value* to the standard output.

`%read`
reads from standard input and returns a string or, in the case of end-of-file, `()`.

Similar to other shells, the standard input, output, and error channels, as well as other files known by active descriptors can be redirected from or to a file or a pipe.

'*Here-documents*' and '*here-strings*', as known e. g. from BASH, are also supported.

Special Variables

Some variables (`0`, `pid`, `home`, `path`, `history`, `prompt`, `signals` etc.) have special meaning to ES, and redefining them changes interpreter's behaviour. Only dynamically bound (top-level or `local`-bound) variables are interpreted in this way. Lexically bound variables are irrelevant.

Hooks

The hook functions implement some internal ES operations, which in some cases it is convenient to invoke explicitly. Besides, their values can be changed, thus altering ES's behaviour.

The names of all hook functions begin with `%`. (The names of some built-in utility functions also begin with `%`.)

Primitives

A primitive is like a function but is referenced by prefixing its name with `$&`. Calling a primitive is a sure way to refer to a built-in behaviour of ES, even in the presence of redefinitions. Some primitives implement built-in functions with the same names, others implement hooks. Still others have no direct connection to BIFs or hooks. The list of primitives can be obtained by calling the primitive `primitives`.