Es: A shell with higher-order functions

Paul Haahr – Adobe Systems Incorporated
Byron Rakitzis – Network Appliance Corporation

ABSTRACT

In the fall of 1990, one of us (Rakitzis) re-implemented the Plan 9 command interpreter, *rc*, for use as a UNIX shell. Experience with that shell led us to wonder whether a more general approach to the design of shells was possible, and this paper describes the result of that experimentation. We applied concepts from modern functional programming languages, such as Scheme and ML, to shells, which typically are more concerned with UNIX features than language design. Our shell is both simple and highly programmable. By exposing many of the internals and adopting constructs from functional programming languages, we have created a shell which supports new paradigms for programmers.

Although most users think of the shell as an interactive command interpreter, it is really a programming language in which each statement runs a command. Because it must satisfy both the interactive and programming aspects of command execution, it is a strange language, shaped as much by history as by design.
— Brian Kernighan & Rob Pike [1]

Introduction

A shell is both a programming language and the core of an interactive environment. The ancestor of most current shells is the 7th Edition Bourne shell[2], which is characterized by simple semantics, a minimal set of interactive features, and syntax that is all too reminiscent of Algol. One recent shell, *rc*[3], substituted a cleaner syntax but kept most of the Bourne shell's attributes. However, most recent developments in shells (e.g., *csh*, *ksh*, *zsh*) have focused on improving the interactive environment without changing the structure of the underlying language – shells have proven to be resistant to innovation in programming languages.

While *rc* was an experiment in adding modern syntax to Bourne shell semantics, *es* is an exploration of new semantics combined with *rc*-influenced syntax: *es* has lexically scoped variables, first-class functions, and an exception mechanism, which are concepts borrowed from modern programming languages such as Scheme and ML[4, 5].

In *es*, almost all standard shell constructs (e.g., pipes and redirection) are translated into a uniform representation: function calls. The primitive functions which implement those constructs can be manipulated the same way as all other functions: invoked, replaced, or passed as arguments to other functions. The ability to replace primitive functions in *es* is key to its extensibility; for example, a user can override the definition of pipes to cause remote execution, or the path-searching machinery to implement a path look-up cache.

At a superficial level, *es* looks like most UNIX shells. The syntax for pipes, redirection, background jobs, etc., is unchanged from the Bourne shell. *Es*'s programming constructs are new, but reminiscent of *rc* and Tcl[6].

Es is freely redistributable, and is available by anonymous ftp from *ftp.white.toronto.edu*.

Using *es*

Commands

For simple commands, *es* resembles other shells. For example, newline usually acts as a command terminator. These are familiar commands which all work in *es*:

```
cd /tmp
rm Ex*
ps aux | grep '^byron' |
  awk '{print $2}' | xargs kill -9
```

For simple uses, *es* bears a close resemblance to *rc*. For this reason, the reader is referred to the paper on *rc* for a discussion of quoting rules, redirection, and so on. (The examples shown here, however, will try to aim for a lowest common denominator of shell syntax, so that an understanding of *rc* is not a prerequisite for understanding this paper.)

Functions

Es can be programmed through the use of shell functions. Here is a simple function to print the date in *yy-mm-dd* format:

```
fn d {
   date +%y-%m-%d
}
```

Functions can also be called with arguments. *Es* allows parameters to be specified to functions by placing them between the function name and the open-brace. This function takes a command `cmd` and arguments `args` and applies the command to each argument in turn:

```
fn apply cmd args {
   for (i = $args)
      $cmd $i
}
```

For example:[1]

```
es> apply echo testing 1.. 2.. 3..
testing
1..
2..
3..
```

Note that `apply` was called with more than two arguments; *es* assigns arguments to parameters one-to-one, and any leftovers are assigned to the last parameter. For example:

```
es> fn rev3 a b c {
   echo $c $b $a
}
es> rev3 1 2 3 4 5
3 4 5 2 1
```

If there are fewer arguments than parameters, *es* leaves the leftover parameters null:

```
es> rev3 1
1
```

So far we have only seen simple strings passed as arguments. However, *es* functions can also take program fragments (enclosed in braces) as arguments. For example, the `apply` function defined above can be used with program fragments typed directly on the command line:

```
es> apply @ i {cd $i; rm -f *} \
                        /tmp /usr/tmp
```

This command contains a lot to understand, so let us break it up slowly.

In any other shell, this command would usually be split up into two separate commands:

```
es> fn cd-rm i {
   cd $i
   rm -f *
}
es> apply cd-rm /tmp /usr/tmp
```

Therefore, the construct

```
@ i {cd $i; rm -f *}
```

is just a way of inlining a function on the command-line. This is called a *lambda*.[2] It takes the form

`@` *parameters* `{` *commands* `}`

In effect, a lambda is a procedure ‘‘waiting to happen.’’ For example, it is possible to type:

```
es> @ i {cd $i; rm -f *} /tmp
```

directly at the shell, and this runs the inlined function directly on the argument `/tmp`.

There is one more thing to notice: the inline function that was supplied to `apply` had a parameter named `i`, and the `apply` function itself used a reference to a variable called `i`. Note that the two uses did not conflict: that is because *es* function parameters are *lexically scoped*, much as variables are in C and Scheme.

Variables

The similarity between shell functions and lambdas is not accidental. In fact, function definitions are rewritten as assignments of lambdas to shell variables. Thus these two *es* commands are entirely equivalent:

```
fn echon args {echo -n $args}
fn-echon = @ args {echo -n $args}
```

In order not to conflict with regular variables, function variables have the prefix `fn-` prepended to their names. This mechanism is also used at execution time; when a name like `apply` is seen by *es*, it first looks in its symbol table for a variable by the name `fn-apply`. Of course, it is always possible to execute the contents of any variable by dereferencing it explicitly with a dollar sign:

```
es> silly-command = {echo hi}
es> $silly-command
hi
```

The previous examples also show that variables can be set to contain program fragments as well as simple strings. In fact, the two can be intermixed:

```
es> mixed = {ls} hello, {wc} world
es> echo $mixed(2) $mixed(4)
hello, world
es> $mixed(1) | $mixed(3)
      61      61     478
```

Variables can hold a list of commands, or even a list of lambdas. This makes variables into versatile tools. For example, a variable could be used as a function dispatch table.

[1] In our examples, we use ‘‘`es>` ’’ as *es*’s prompt. The default prompt, which may be overridden, is ‘‘`;` ’’ which is interpreted by *es* as a null command followed by a command separator. Thus, whole lines, including prompts, can be cut and pasted back to the shell for re-execution. In examples, an italic fixed width font indicates user input.

[2] The keyword `@` introduces the lambda. Since `@` is not a special character in *es* it must be surrounded by white space. `@` is a poor substitute for the letter λ, but it was one of the few characters left on a standard keyboard which did not already have a special meaning.

Binding

In the section on functions, we mentioned that function parameters are lexically scoped. It is also possible to use lexically-scoped variables directly. For example, in order to avoid interfering with a global instance of `i`, the following scoping syntax can be used:

```
let (var = value) {
    commands which use $var
}
```

Lexical binding is useful in shell functions, where it becomes important to have shell functions that do not clobber each others' variables.

Es code fragments, whether used as arguments to commands or stored in variables, capture the values of enclosing lexically scoped values. For example,

```
es> let (h=hello; w=world) {
    hi = { echo $h, $w }
}
es> $hi
hello, world
```

One use of lexical binding is in redefining functions. A new definition can store the previous definition in a lexically scoped variable, so that it is only available to the new function. This feature can be used to define a function for tracing calls to other functions:

```
fn trace functions {
    for (func = $functions)
        let (old = $(fn-$func))
            fn $func args {
              echo calling $func $args
              $old $args
            }
}
```

The `trace` function redefines all the functions which are named on its command line with a function that prints the function name and arguments and then calls the previous definition, which is captured in the lexically bound variable `old`. Consider a recursive function `echo-nl` which prints its arguments, one per line:

```
es> fn echo-nl head tail {
    if {!~ $#head 0} {
        echo $head
        echo-nl $tail
    }
}
es> echo-nl a b c
a
b
c
```

Applying `trace` to this function yields:

```
es> trace echo-nl
es> echo-nl a b c
calling echo-nl a b c
a
calling echo-nl b c
b
calling echo-nl c
c
calling echo-nl
```

The reader should note that

`!`*cmd*

is *es*'s ''not'' command, which inverts the sense of the return value of *cmd*, and

~ *subject pattern*

matches *subject* against *pattern* and returns true if the subject is the same as the pattern. (In fact, the matching is a bit more sophisticated, for the pattern may include wildcards.)

Shells like the Bourne shell and *rc* support a form of local assignment known as *dynamic binding*. The shell syntax for this is typically:

var=*value command*

That notation conflicts with *es*'s syntax for assignment (where zero or more words are assigned to a variable), so dynamic binding has the syntax:

```
local (var = value) {
    commands which use $var
}
```

The difference between the two forms of binding can be seen in an example:

```
es> x = foo
es> let (x = bar) {
    echo $x
    fn lexical { echo $x }
}
bar
es> lexical
bar
es> local (x = baz) {
    echo $x
    fn dynamic { echo $x }
}
baz
es> dynamic
foo
```

Settor Variables

In addition to the prefix (`fn-`) for function execution described earlier, *es* uses another prefix to search for *settor variables*. A settor variable `set-`*foo* is a variable which gets evaluated every time the variable *foo* changes value. A good example of settor variable use is the `watch` function:

```
fn watch vars {
   for (var = $vars) {
      set-$var = @ {
         echo old $var '=' $$var
         echo new $var '=' $*
         return $*
      }
   }
}
```

`Watch` establishes a settor function for each of its parameters; this settor prints the old and new values of the variable to be set, like this:

```
es> watch x
es> x=foo bar
old x =
new x = foo bar
es> x=fubar
old x = foo bar
new x = fubar
```

Return Values

UNIX programs exit with a single number between 0 and 255 reported as their statuses. *Es* supplants the notion of an exit status with ''rich'' return values. An *es* function can return not only a number, but any object: a string, a program fragment, a lambda, or a list which mixes such values.

The return value of a command is accessed by prepending the command with <>:

```
es> fn hello-world {
   return 'hello, world'
}
es> echo <>{hello-world}
hello, world
```

This example shows rich return values being used to implement hierarchical lists:

```
fn cons a d {
   return @ f { $f $a $d }
}
fn car p { $p @ a d { return $a } }
fn cdr p { $p @ a d { return $d } }
```

The first function, `cons`, returns a function which takes as its argument another function to run on the parameters `a` and `d`. `car` and `cdr` each invoke the kind of function returned by `cons`, supplying as the argument a function which returns the first or second parameter, respectively. For example:

```
es> echo <>{car <>{cdr <>{
   cons 1 <>{cons 2 <>{cons 3 nil}}
}}}
2
```

Exceptions

In addition to traditional control flow constructs – loops, conditionals, subroutines – *es* has an exception mechanism which is used for implementing non-structured control flow. The built-in function `throw` raises an exception, which typically consists of a string which names the exception and other arguments which are specific to the named exception type. For example, the exception `error` is caught by the default interpreter loop, which treats the remaining arguments as an error message. Thus:

```
es> fn in dir cmd {
   if {~ $#dir 0} {
     throw error 'usage: in dir cmd'
   }
   fork     # run in a subshell
   cd $dir
   $cmd
}
es> in
usage: in dir cmd
es> in /tmp ls
webster.socket     yacc.312
```

By providing a routine which catches `error` exceptions, a programmer can intercept internal shell errors before the message gets printed.

Exceptions are also used to implement the `break` and `return` control flow constructs, and to provide a way for user code to interact with UNIX signals. While six error types are known to the interpreter and have special meanings, any set of arguments can be passed to `throw`.

Exceptions are trapped with the built-in `catch`, which typically takes the form

```
catch @ e args { handler } { body }
```

`Catch` first executes *body*; if no exception is raised, `catch` simply returns, passing along *body*'s return value. On the other hand, if anything invoked by *body* throws an exception, *handler* is run, with `e` bound to the exception that caused the problem. For example, the last two lines of `in` above can be replaced with:

```
catch @ e msg {
   if {~ $e error} {
      echo >[1=2] in $dir: $msg
   } {
      throw $e $msg
   }
} {
   cd $dir
   $cmd
}
```

to better identify for a user where an error came from:

```
es> in /temp ls
in /temp: chdir /temp:
        No such file or directory
```

Spoofing

Es's versatile functions and variables are only half of the story; the other part is that *es*'s shell syntax is just a front for calls on built-in functions. For example:

```
ls > /tmp/foo
```

is internally rewritten as

```
%create 1 /tmp/foo {ls}
```

before it is evaluated. `%create` is the built-in function which opens `/tmp/foo` on file-descriptor 1 and runs `ls`.

The value of this rewriting is that the `%create` function (and that of just about any other shell service) can be *spoofed*, that is, overridden by the user: when a new `%create` function is defined, the default action of redirection is overridden.

Furthermore, `%create` is not really the built-in file redirection service. It is a hook to the *primitive* `$&create`, which itself cannot be overridden. That means that it is always possible to access the underlying shell service, even when its hook has been reassigned.

Keeping this in mind, here is a spoof of the redirection operator that we have been discussing. This spoof is simple: if the file to be created exists (determined by running `test -f`), then the command is not run, similar to the C-shell's ``noclobber'' option:

```
fn %create fd file cmd {
   if {test -f $file} {
      throw error $file exists
   } {
      $&create $fd $file $cmd
   }
}
```

In fact, most redefinitions do not refer to the `$&`-forms explicitly, but capture references to them with lexical scoping. Thus, the above redefinition would usually appear as

```
let (create = $fn-%create)
   fn %create fd file cmd {
      if {test -f $file} {
         throw error $file exists
      } {
         $create $fd $file $cmd
      }
   }
```

The latter form is preferable because it allows multiple redefinitions of a function; the former version would always throw away any previous redefinitions.

Overriding traditional shell built-ins is another common example of spoofing. For example, a `cd` operation which also places the current directory in the title-bar of the window (via the hypothetical command `title`) can be written as:

```
es> let (pipe = $fn-%pipe) {
   fn %pipe first out in rest {
      if {~ $#out 0} {
         time $first
      } {
         $pipe {time $first} $out $in {%pipe $rest}
      }
   }
}
es> cat paper9 | tr -cs a-zA-Z0-9 '\012' | sort | uniq -c | sort -nr | sed 6q
 213 the
 150 a
 120 to
 115 of
 109 is
  96 and
   2r   0.3u   0.2s   cat paper9
   2r   0.3u   0.2s   tr -cs a-zA-Z0-9 \012
   2r   0.5u   0.2s   sort
   2r   0.4u   0.2s   uniq -c
   3r   0.2u   0.1s   sed 6q
   3r   0.6u   0.2s   sort -nr
```

Figure 1: Timing pipeline elements

```
let (cd = $fn-%cd)
    fn cd {
        $cd $*
        title `{pwd}
    }
```

Spoofing can also be used for tasks which other shells cannot do; one example is timing each element of a pipeline by spoofing `%pipe`, along the lines of the pipeline profiler suggested by Jon Bentley[7]; see Figure 1.

Many shells provide some mechanism for caching the full pathnames of executables which are looked up in a user's `$PATH`. *Es* does not provide this functionality in the shell, but it can easily be added by any user who wants it. The function `%pathsearch` (see Figure 2) is invoked to look-up non-absolute file names which are used as commands.

One other piece of *es* which can be replaced is the interpreter loop. In fact, the default interpreter is written in *es* itself; see Figure 3.

A few details from this example need further explanation. The exception `retry` is intercepted by `catch` when an exception handler is running, and causes the body of the `catch` routine to be re-run.

```
let (search = $fn-%pathsearch) {
    fn %pathsearch prog {
        let (file = <>{$search $prog}) {
            if {~ $#file 1 && ~ $file /*} {
                path-cache = $path-cache $prog
                fn-$prog = $file
            }
            return $file
        }
    }
}
fn recache {
    for (i = $path-cache)
        fn-$i =
    path-cache =
}
```

Figure 2: Path caching

```
fn %interactive-loop {
    let (result = 0) {
        catch @ e msg {
            if {~ $e eof} {
                return $result
            } {~ $e error} {
                echo >[1=2] $msg
            } {
                echo >[1=2] uncaught exception: $e $msg
            }
            throw retry
        } {
            while {} {
                %prompt
                let (cmd = <>{%parse $prompt}) {
                    result = <>{$cmd}
                }
            }
        }
    }
}
```

Figure 3: Default interactive loop

`%parse` prints its first argument to standard error, reads a command (potentially more than one line long) from the current source of command input, and throws the `eof` exception when the input source is exhausted. The hook `%prompt` is provided for the user to redefine, and by default does nothing.

Other spoofing functions which either have been suggested or are in active use include: a version of `cd` which asks the user whether to create a directory if it does not already exist; versions of redirection and program execution which try spelling correction if files are not found; a `%pipe` to run pipeline elements on (different) remote machines to obtain parallel execution; automatic loading of shell functions; and replacing the function which is used for tilde expansion to support alternate definitions of home directories. Moreover, for debugging purposes, one can use `trace` on hook functions.

Implementation

Es is implemented in about 8000 lines of C. Although we estimate that about 1000 lines are devoted to portability issues between different versions of UNIX, there are also a number of work-arounds that *es* must use in order to blend with UNIX. The `path` variable is a good example.

The *es* convention for path searching involves looking through the list elements of a variable called `path`. This has the advantage that all the usual list operations can be applied equally to `path` as any other variable. However, UNIX programs expect the path to be a colon-separated list stored in `PATH`. Hence *es* must maintain a copy of each variable, with a change in one reflected as a change in the other.

Initialization

Much of *es*'s initialization is actually done by an *es* script, called `initial.es`, which is converted by a shell script to a C character string at compile time and stored internally. The script illustrates how the default actions for *es*'s parser is set up, as well as features such as the `path`/`PATH` aliasing mentioned above.

Much of the script consists of lines like:

```
fn-%and        = $&and
fn-%append     = $&append
fn-%background = $&background
```

which bind the shell services such as short-circuit-and, backgrounding, etc., to the `%`-prefixed hook variables.

There are also a set of assignments which bind the built-in shell functions to their hook variables:

```
fn-.       = $&dot
fn-break   = $&break
fn-catch   = $&catch
```

The difference with these is that they are given names invoked directly by the user; ``.'' is the Bourne-compatible command for ``sourcing'' a file.

Finally, some settor functions are defined to work around UNIX path searching (and other) conventions. For example,

```
set-path = @ {
    local (set-PATH = )
        PATH = <>{%flatten : $*}
    return $*
}
set-PATH = @ {
    local (set-path = )
        path = <>{%fsplit : $*}
    return $*
}
```

A note on implementation: these functions temporarily assign their opposite-case settor cousin to null before making the assignment to the opposite-case variable. This avoids infinite recursion between the two settor functions.

The Environment

UNIX shells typically maintain a table of variable definitions which is passed on to child processes when they are created. This table is loosely referred to as the environment or the environment variables. Although traditionally the environment has been used to pass values of variables only, the duality of functions and variables in *es* has made it possible to pass down function definitions to subshells. (While *rc* also offered this functionality, it was more of a kludge arising from the restriction that there was not a separate space for ``environment functions.'')

Having functions in the environment brings them into the same conceptual framework as variables – they follow identical rules for creation, deletion, presence in the environment, and so on. Additionally, functions in the environment are an optimization for file I/O and parsing time. Since nearly all shell state can now be encoded in the environment, it becomes superfluous for a new instance of *es*, such as one started by *xterm*(1), to run a configuration file. Hence shell startup becomes very quick.

As a consequence of this support for the environment, a fair amount of *es* must be devoted to ``unparsing'' function definitions so that they may be passed as environment strings. This is complicated a bit more because the lexical environment of a function definition must be preserved at unparsing. This is best illustrated by an example:

```
es> let (a=b) fn foo {echo $a}
```

which lexically binds `b` to the variable `a` for the scope of this function definition. Therefore, the external representation of this function must make this information explicit. It is encoded as:

```
es> whatis foo
%closure(a=b)@ * {echo $a}
```

(Note that for cultural compatibility with other shells, functions with no named parameters use ``*'' for binding arguments.)

Interactions With Unix

Unlike most traditional shells, which have feature sets dictated by the UNIX system call interface, *es* contains features which do not interact well with UNIX itself. For example, rich return values make sense from shell functions (which are run inside the shell itself) but cannot be returned from shell scripts or other external programs, because the *exit/wait* interface only supports passing small integers. This has forced us to build some things into the shell which otherwise could be external.

The exception mechanism has similar problems. When an exception is raised from a shell function, it propagates as expected; if raised from a subshell, it cannot be propagated as one would like it to be: instead, a message is printed on exit from the subshell and a false exit status is returned. We consider this unfortunate, but there seemed no reasonable way to tie exception propagation to any existing UNIX mechanism. In particular, the signal machinery is unsuited to the task. In fact, signals complicate the control flow in the shell enough, and cause enough special cases throughout the shell, so as to be more of a nuisance than a benefit.

One other unfortunate consequence of our shoehorning *es* onto UNIX systems is the interaction between lexically scoped variables, the environment, and subshells. Two functions, for example, may have been defined in the same lexical scope. If one of them modifies a lexically scoped variable, that change will affect the variable as seen by the other function. On the other hand, if the functions are run in a subshell, the connection between their lexical scopes is lost as a consequence of them being exported in separate environment strings. This does not turn out to be a significant problem, but it does not seem intuitive to a programmer with a background in functional languages.

One restriction on *es* that arose because it had to work in a traditional UNIX environment is that lists are not hierarchical; that is, lists may not contain lists as elements. In order to be able to pass lists to external programs with the same semantics as passing them to shell functions, we had to restrict lists to the same structure as *exec*-style argument vectors. Therefore all lists are flattened, as in *rc* and *csh*.

Garbage Collection

Since *es* incorporates a true lambda calculus, it includes the ability to create true recursive structures, that is, objects which include pointers to themselves, either directly or indirectly. While this feature can be useful for programmers, it has the unfortunate consequence of making memory management in *es* more complex than that found in other shells. Simple memory reclamation strategies such as arena style allocation [8] or reference counting are unfortunately inadequate; a full garbage collection system is required to plug all memory leaks.

Based on our experience with *rc*'s memory use, we decided that a copying garbage collector would be appropriate for *es*. The observations leading to this conclusion were: (1) between two separate commands little memory is preserved (it roughly corresponds to the storage for environment variables); (2) command execution can consume large amounts of memory for a short time, especially when loops are involved; and, (3) however much memory is used, the working set of the shell will typically be much smaller than the physical memory available. Thus, we picked a strategy where we traded relatively fast collection times for being somewhat wasteful in the amount of memory used in exchange. While a generational garbage collector might have made sense for the same reasons that we picked a copying collector, we decided to avoid the added complexity implied by switching to the generational model.

During normal execution of the shell, memory is acquired by incrementing a pointer through a pre-allocated block. When this block is exhausted, all live pointers from outside of garbage collector memory, the *rootset*, are examined, and any structure that they point to is copied to a new block. When the rootset has been scanned, all the freshly copied data is scanned similarly, and the process is repeated until all reachable data has been copied to the new block. At this point, the memory request which triggered the collection should be able to succeed. If not, a larger block is allocated and the collection is redone.

During some parts of the shell's execution – notably while the *yacc* parser driver is running – it is not possible to identify all of the rootset, so garbage collection is disabled. If an allocation request is made during this time for which there is not enough memory available in the arena, a new chunk of memory is grabbed so that allocation can continue.

Garbage collectors have developed a reputation for being hard to debug. The collection routines themselves typically are not the source of the difficulty. Even more sophisticated algorithms than the one found in *es* are usually only a few hundred lines of code. Rather, the most common form of GC bug is failing to identify all elements of the rootset, since this is a rather open-ended problem which has implications for almost every routine. To find this form of bug, we used a modified version of the garbage collector which has two key features: (1) a collection is initiated at every allocation when the collector is not disabled, and (2) after a collection finishes, access to all the memory from the old region is disabled.[3] Thus, any reference to a pointer in garbage collector space which could be invalidated by a collection immediately causes a memory protection fault. We strongly recommend this technique to anyone implementing a copying garbage collector.

There are two performance implications of the garbage collector; the first is that, occasionally, while the shell is running, all action must stop while the collector is invoked. This takes roughly 4% of the running time of the shell. More serious is that at the time of any potential allocation, either the collector must be disabled, or all pointers to structures in garbage collector memory must be identified, effectively requiring them to be in memory at known addresses, which defeats the registerization optimizations required for good performance from modern architectures. It is hard to quantify the performance consequences of this restriction.

The garbage collector consists of about 250 lines of code for the collector itself (plus another 300 lines of debugging code), along with numerous declarations that identify variables as being part of the rootset and small (typically 5 line) procedures to allocate, copy, and scan all the structure types allocated from collector space.

Future Work

There are several places in *es* where one would expect to be able to redefine the built-in behavior and no such hook exists. The most notable of these is the wildcard expansion, which behaves identically to that in traditional shells. We hope to expose some of the remaining pieces of *es* in future versions.

One of the least satisfying pieces of *es* is its parser. We have talked of the distinction between the core language and the full language; in fact, the translation of *syntactic sugar* (i.e., the convenient UNIX shell syntax presented to the user) to core language features is done in the same *yacc*-generated parser as the recognition of the core language. Unfortunately, this ties the full language in to the core very tightly, and offers little room for a user to extend the syntax of the shell.

We can imagine a system where the parser only recognizes the core language, and a set of exposed transformation rules would map the extended syntax which makes *es* feel like a shell, down to the core language. The *extend-syntax* [9] system for Scheme provides a good example of how to design such a mechanism, but it, like most other macro systems designed for Lisp-like languages, does not mesh well with the free-form syntax that has evolved for UNIX shells.

[3]This disabling depends on operating system support.

The current implementation of *es* has the undesirable property that all function calls cause the C stack to nest. In particular, tail calls consume stack space, something they could be optimized not to do. Therefore, properly tail recursive functions, such as `echo-nl` above, which a Scheme or ML programmer would expect to be equivalent to looping, have hidden costs. This is an implementation deficiency which we hope to remedy in the near future.

Es, in addition to being a good language for shell programming, is a good candidate for a use as an embeddable ‘‘scripting’’ language, along the lines of Tcl. *Es*, in fact, borrows much from Tcl – most notably the idea of passing around blocks of code as unparsed strings – and, since the requirements on the two languages are similar, it is not surprising that the syntaxes are so similar. *Es* has two advantages over most embedded languages: (1) the same code can be used by the shell or other programs, and many functions could be identical; and (2) it supports a wide variety of programming constructs, such as closures and exceptions. We are currently working on a ‘‘library’’ version of *es* which could be used stand-alone as a shell or linked in other programs, with or without shell features such as wildcard expansion or pipes.

Conclusions

There are two central ideas behind *es*. The first is that a system can be made more programmable by exposing its internals to manipulation by the user. By allowing spoofing of heretofore unmodifiable shell features, *es* gives its users great flexibility in tailoring their programming environment, in ways that earlier shells would have supported only with modification of shell source itself.

Second, *es* was designed to support a model of programming where code fragments could be treated as just one more form of data. This feature is often approximated in other shells by passing commands around as strings, but this approach requires resorting to baroque quoting rules, especially if the nesting of commands is several layers deep. In *es*, once a construct is surrounded by braces, it can be stored or passed to a program with no fear of mangling.

Es contains little that is completely new. It is a synthesis of the attributes we admire most from two shells – the venerable Bourne shell and Tom Duff’s *rc* – and several programming languages, notably Scheme and Tcl. Where possible we tried to retain the simplicity of *es*’s predecessors, and in several cases, such as control flow constructs, we believe that we have simplified and generalized what was found in earlier shells.

We do not believe that *es* is the ultimate shell. It has a cumbersome and non-extensible syntax, the support for traditional shell notations forced some unfortunate design decisions, and some of *es*'s features, such as exceptions and rich return values, do not interact as well with UNIX as we would like them to. Nonetheless, we think that *es* is successful as both a shell and a programming language, and would miss its features and extensibility if we were forced to revert to other shells.

Acknowledgements

We'd like to thank the many people who helped both with the development of *es* and the writing of this paper. Dave Hitz supplied essential advice on where to focus our efforts. Chris Siebenmann maintained the *es* mailing list and ftp distribution of the source. Donn Cave, Peter Ho, Noel Hunt, John Mackin, Bruce Perens, Steven Rezsutek, Rich Salz, Scott Schwartz, Alan Watson, and all other contributors to the list provided many suggestions, which along with a ferocious willingness to experiment with a not-ready-for-prime-time shell, were vital to *es*'s development. Finally, Susan Karp and Beth Mitcham read many drafts of this paper and put up with us while *es* was under development.

References

1. Brian W. Kernighan and Rob Pike, *The UNIX Programming Environment,* Prentice-Hall, 1984.
2. S. R. Bourne, ''The UNIX Shell,'' *Bell Sys. Tech. J.*, vol. 57, no. 6, pp. 1971-1990, 1978.
3. Tom Duff, ''Rc – A Shell for Plan 9 and Unix Systems,'' in *UKUUG Conference Proceedings*, pp. 21-33, Summer 1990.
4. William Clinger and Jonathan Rees (editors), *The Revised^4 Report on the Algorithmic Language Scheme*, 1991.
5. Robin Milner, Mads Tofte, and Robert Harper, *The Definition of Standard ML,* MIT Press, 1990.
6. John Ousterhout, ''Tcl: An Embeddable Command Language,'' in *Usenix Conference Proceedings*, pp. 133-146, Winter 1990.
7. Jon L. Bentley, *More Programming Pearls,* Addison-Welsey, 1988.
8. David R. Hanson, ''Fast allocation and deallocation of memory based on object lifetimes,'' *Software—Practice and Experience*, vol. 20, no. 1, pp. 5-12, January, 1990.
9. R. Kent Dybvig, *The Scheme Programming Language,* Prentice-Hall, 1987.

Author Information

Paul Haahr is a computer scientist at Adobe Systems Incorporated where he works on font rendering technology. His interests include programming languages, window systems, and computer architecture. Paul received an A.B. in computer science from Princeton University in 1990. He can be reached by electronic mail at *haahr@adobe.com* or by surface mail at Adobe Systems Incorporated, 1585 Charleston Road, Mountain View, CA 94039.

Byron Rakitzis is a system programmer at Network Appliance Corporation, where he works on the design and implementation of their network file server. In his spare time he works on shells and window systems. His free-software contributions include a UNIX version of *rc*, the Plan 9 shell, and *pico*, a version of Gerard Holzmann's picture editor *popi* with code generators for SPARC and MIPS. He received an A.B. in Physics from Princeton University in 1990. He has two cats, Pooh-Bah and Goldilocks, who try to rule his home life. Byron can be reached at *byron@netapp.com* or at Network Appliance Corporation, 2901 Tasman Drive, Suite 208 Santa Clara, CA 95054.