NAME

zmsh − zmailer shell

SYNOPSIS

zmsh

[−**CIJLOPRSYisaefhntuvx**] [−**c** *command*] [*script ...*]

DESCRIPTION

The *zmsh*(1zm) is an implementation of the Bourne shell suitable for use with the ZMailer *router*(8zm) as its configuration file language interpreter. It contains extensions that allow structured data (in the form of lists) to be manipulated.

The shell supports three basic kinds of functions: Unix commands, user-defined functions, and builtin commands. The later comes in two variations: normal functions which take string arguments and return a status code (much as an embedded Unix command would work), and list-manipulation functions which understand list arguments and can return list arguments. The defined functions can take any form of argument and return any form of value (a status code, a string, or a list).

Shell operations (pipes, backquote evaluation and substitution) will work between combinations of builtin functions, defined functions, and Unix commands.

The shell precompiles its input to a (possibly optimized) byte-code form, which is then interpreted as required. This means that the original form of the input is not kept around in-core for future reference. If the input is an included file, the shell will try to save the byte-code form in a **.fc** file associated with the input file. For example, if input is from **file.cf**, the shell will try to create **fc/file.fc** and then **file.fc**. These files will in turn be searched for and loaded (after a consistency check) whenever a **.cf** file is included.

The effects of input and output redirections are predicted prior to the execution of a command and its I/O setup.

INCOMPATIBILITIES

zmsh is based on the System V release 3 shell as described in a SunOS 4.0 manual page. It conforms to the behaviour of that shell, with the following differences:

ˆ is not accepted as an alternative pipe character.

A symbol may have both a function definition and a value.

The shell is 8 bit transparent.

All occurrences in the *sh*(1) manual page of "if any character in the word is quoted " should be read as "if the word or the first character of it is quoted ".

The −k option is not supported.

Inside backquotes, for efficient deadlock avoidance, Unix commands are executed in parallel. If you need serial execution, use ‘**(***c1* **;** *c2***)**‘ instead of ‘*c1* **;** *c2*‘.

Builtin commands that want to write more than one pipe buffer full should detach themselves into a grandchild and run to completion independently.

The **hash** builtin is not fully supported. The fancy printing options are not implemented, but program locations are hashed.

If **PS1** or **PS2** are defined functions, then their function definition will be assumed to print an appropriate prompt and the function will be called instead of printing out the value of the **PS1**/**PS2** shell variables.

In **IFS**, *newline* is always ignored (it is always a separator) when reading commands. This seems to match normal shell behaviour but not obviously so from the manual page.

The **SHELL** environment variable is not special, since there is no restricted mode.

SIGSEGV is not special.

The message printed for **${FOO?**...} starts with *progname***:** instead of **FOO:**.

The functions **login**, **newgrp**, **pwd**, and **readonly**, are not built in.

Functions can have a comma-separated list of named parameters. If the argument list is exhausted on a function call, the remainder of the parameters are set to the empty string. If the parameter list is exhausted, the **@** variable is set to the remainder of the arguments. This behaviour is backward compatible with normal shell functions.

The **local** builtin statement declares local variables within a function scope.

Builtin commands in pipelines are run within the shell, so variable settings in that context will affect the shell process.

The **type** function does not print a shell function definition in text form.

There is a **builtin** function to force its arguments to be evaluated as builtin function call.

The termination string in here documents must be static. Something like **cat << ‘echo EOF‘** will not work.

EXTENSIONS

The following additions and extensions have been made relative to the base shell:

The Korn shell backquote mechanism is supported. This means that **‘foo bar‘** is equivalent to **$(foo bar)** in all contexts, although the later form is preferred for clarity.

If you are used to an old Bourne shell, the following are the unusual builtin functions in this shell:

test (or [), **getopts**, **times**, **type**, **builtin**, **sleep**.

The **type** and **builtin** functions are lifted from the Ninth Edition Bourne shell. The **test** and **sleep** functions are in the shell because the mailer will use them very frequently.

The **ssift**/**tsift** statements are special-purpose constructs for the mailer. They act like a **case** statement, except the labels are regular expressions, are separated from the label body by whitespace instead of being terminated by a closing parenthesis, and exiting a **ssift**/**tsift** label body will just cause a fall-through to the next **ssift / tsift** label. This is similar to the production-rule semantics of Sendmail.

The **tsift** structure has (for compability reasons) alias: **sift**. The difference in between **ssift** and **tsift**(**sift**) is on what is being matched: The **tsift** matches **TOKENS** in RFC-822 sense, while **ssift** at first expands the token sequence into a string, and then does the regexpr matching.

The **local** statement can appear anywhere in a scope (a {...} grouping) and declares variables that are created in the current scope and destroyed on exit from it. Such variables are initialized to refer to a null string.

Functions may be defined with named parameters, which are scoped variables and destroyed on return from the function.

LIST SEMANTICS

The semantics of lists have been defined in various contexts:

List:

(a b (c (d e) f (g)))

A list may contain strings and other lists as elements. It is printed and usually entered using traditional Lisp syntax.

Variable assignment:

recipient=(what ever)
nil=()

A shell variable value may be either a string or a list. A list value can be entered directly.

Element counts:

$#variable
$(length $(expression))

Every list-valued variable has a length accessed using the **$#** prefix, as in *csh (1)* syntax. The **length** builtin function can also be used to count the number of toplevel elements of a list. Empty lists have length 0. Strings have no length.

Command line:

router $(list smtp neat.cs rayan) (*...***)**

Command-line arguments to builtin or defined functions may be lists. The first argument cannot be entered directly as a list, but later arguments may be. The restriction is due to a syntax clash with function definitions.

Loops:

for i in yes (a b c) no; do; *...***; done**

A list in a loop list is treated as a single element. The loop variable will be bound to it.

Associative (property) lists:

$(get variable symbol)

The value of **variable** should be a list of even length, with alternating attributes and values. The given symbol is matched with an attribute, and the following value is returned.

Associative (property) assignment:

setf $(get variable symbol) value

The **setf** function is used to change the value that would be returned by an expression, typically this is done by pointer manipulation.

Some new builtin functions have been defined to operate on lists:

car (or **first**), returns the first element of the list which is its argument.

cdr (or **rest**), returns the list after the first element of the list which is its argument.

elements
is used to explode a list, for use in a loop or to concatenate the list elements together. For example:

hostlist=(neat.cs smoke.cs)

```
for hosts in $(elements hostlist)
do
        ...
done
```

get is a property list lookup function. Property lists are lists of alternating keywords (properties or attributes) and values. For example:

```
jane=(hair brown eyes blue)
get $jane eyes
```

grind is used for lists instead of **echo**, use it to print a list value.

last returns the last element of the list which is its argument.

length
returns the count of elements of the list which is its argument.

list returns the list containing its arguments as elements.

setf takes a retrieval command and a new value as arguments. At present it works with (combinations of) **car**, **cdr**, **get** and **last**. For example:

```
setf $(get $jane eyes) azur
```

Lists are ignored in any context where they aren't expected. For example, a list value as an argument to **echo** would behave just like a null argument. Builtin functions either know about lists (which subsumes normal strings), or about string values. Defined functions are flexible, whatever you pass them will show up in their argument list.

On a command line, the first argument can never be written as a list (since the parser won't be able to tell that from a function definition). However, the following arguments may be written as lists. This leads to constructs like:

```
aliasexpand $(list local - rayan) plist
```

The one exception to this is in the **return** statement, which may be given a list (or string) as its argument. If so, the return value from the defined function becomes list-valued, and the status code is set to 0. If the argument to the **return** statement is numeric, it is assumed to be the desired status code.

Be forewarned of strange effects if you print to stdout in a list-valued function that is called within a backquote (i.e. where the return value, not the status, is desired) and you expect a list back.

OPTIONS

The following debugging options are specific to the internal function of *zmsh*:

−C print code generation output onto stdout. If this option is doubled, the non-optimized code is printed out instead.

−I print runtime interpreter activity onto /dev/tty; argv:s of executes, assignments, variables, ...

−J print runtime interpreter activity onto /dev/tty; just argv:s of executes.

−L print lexer output onto stdout.

−O optimize the compiled script. If this option is doubled, the optimized code is also printed out.

−P print parser output (S/SL trace output) onto stdout.

−R print I/O actions onto /dev/tty.

−S print scanner output (token assembly) onto stdout.

−Y open /dev/tty for internal debugging use.

These are the normal shell options that are supported by *zmsh*:

−c run the given argument as a shell command script.

−i this shell is interactive, meaning prompts are printed when ready for more input, **SIGTERM** is ignored, and the shell doesn't exit easily. This flag is automatically set if stdin and stderr are both attached to a tty.

−s read commands from stdin. If there are non-option arguments to the shell, the first of these will be interpreted as a shell script to open on stdin, and the rest as arguments to the script.

−a automatically export new or changed shell variables.

−e exit on non-zero status return of any command.

−f disables filename generation (a.k.a. "globbing").

−h hash and cache the location of Unix commands. This option is set by default.

−n read commands but do not execute them.

−t exit after running one command.

−u unset variables produce an error on substitution.

−v print shell input as it is read.

−x print commands as they are executed.

BUGS

Nested here documents don't work with optimization off, and terminator string isn't stacked.

Hitting interrupt during I/O stuff is a bad thing.

The file descriptor prediction doesn't always work, especially in (nested) multi-pipe command lines.

SEE ALSO

sh(1), *router*(8zm), *zmailer.conf* (1zm).

AUTHOR

This program authored and copyright by:
Rayan Zachariassen <no address>
Some "small" tweaks by:
Matti Aarnio <mea@nic.funet.fi>