Hoodospel Documentation

Release alpha

Nikolay Pavlov

June 20, 2015

Contents

Contents:

CHAPTER 1

Language syntax

1.1 Top-level syntax

Hoodospel source code is a sequence of commands separated by newlines. Each command consists of a *command name* followed by a sequence of *argument tokens* followed by a *command prefix* with *argument tokens* which are used to specify more then one command argument. Both first and prefixed arguments are generally optional.

Between command name, first arguments, prefixes and prefix arguments there may be whitespaces (tabs and spaces). They may also precede command.

```
command ::= command-name ( argument+ )? ( prefix argument* )*
```

Both command prefixes and command name are non-empty sequences of latin capital letters and underscores starting with a capital letter.

```
prefix  ::= [A-Z] [A-Z_]*
command-name ::= prefix
```

Hoodospel also supports comments. Comment may be started at any place where some token is expected. Comments are identified by preceding hash character.

```
comment ::= "#" .*
```

There are eight kind of tokens which may form an argument: *variables*, *numbers*, *single-quoted*, *double-quoted*, *plain* and *figure braces* strings, *functions* and *parenthesis expression*.

```
argument ::= variable | number | string | function | parenthesis_expr
string ::=   single-quoted-string
           | double-quoted-string
           | plain-string
           | figure-braces-string
```

- Variable is a sigil followed by a non-empty sequence of latin letters, digits and underscores. Two sigils are supported: `$` indicates environment variable, `&` indicates hoodospel variable.

```
    variable ::= hoodospel-variable | env-variable
    hoodospel-variable ::= "&" varname
    environment-variable ::= "$" varname
    varname ::= [a-zA-Z0-9_]+
```

- Numbers start with either a digit or a sign: _ for negative numbers and + for positive numbers. There must be at least one digit in number.

```
number ::= ("_" | "+")? [0-9]+
```

- Single-quoted strings are sequences of characters starting and ending with a single quote. To escape a single quote you should double it. No other escapes are possible.

```
single-quoted-string ::= "'" ([^'] | "''")* "'"
```

- Double-quoted strings are sequences of characters starting and ending with a double quote. The following escape sequences are accepted: `\xXX` (but not `\x00`), `\uXXXX` (but not `\u0000`), `\UXXXXXXXX` (but not `\U00000000`), `\\`, `\"`, `\r`, `\n`, `\t`.

  Meaning of the escape sequences:

Sequence	Meaning
`\xXX`	Byte 0xXX.
`\uXXXX`	Unicode character U+XXXX
`\UXXXXXXXX`	Unicode character U+XXXXXXXX
`\\`	Backslash
`\"`	Double quote
`\r`	Carriage return (0x0D)
`\n`	Newline (0x0A)
`\t`	Tab (0x09)

```
double-quoted-string ::= "\"" ([^"\\] | escape-sequence)* "\""
escape-sequence ::=   "\\x" ( hex-digit x 2 )
                    | "\\u" ( hex-digit x 4 )
                    | "\\U" ( hex-digit x 8 )
                    | "\\" [\\"rnt]
```

- Plain strings start with either a unicode character, a lowercase latin letter, a back or forward slash, a dot or a dash. Following characters are considered a part of plain string as long as they are not whitespace characters, parenthesis, brackets or figure braces.

```
plain-string ::= [a-z/\\.\-] [^\[\]{}() \t]*
```

- There is a special kind of plain strings: figure braces strings that contain only figure braces.

```
figure-braces-string ::= "{"+ | "}"+
```

- Functions are just like *prefixes*, but unlike them functions start with a colon:

```
function ::= ":" [A-Z] [A-Z_]*
```

- There are also parenthesis expressions:

```
parenthesis_expr ::= "(" argument* ")"
```

1.2 Command arguments

Different commands accept different arguments. There are kinds of arguments:

- Lval arguments designate arguments which may be assigned to. Rlval arguments designate existing variables which may be assigned to. Both always contain a single *variable token*.
- Empty arguments are for command prefixes. They designate that prefix does not accept any arguments: only the presence of the prefix matters.

- Expression arguments are the only ones that may contain more then one token. In fact they may contain any number of *argument tokens*. Note that parenthesis in *parenthesis expressions* must be balanced.
- Pattern is an *expression* which must result in a string value treated like described in *pattern syntax* section.

  There is no difference between expressions and patterns from the parser point of view.
- Message is an *expression* which must result in a string value followed by other values treated like described in *messages* section.

  There is no difference between expressions and messages from the parser point of view.
- Version argument is a single token: a single-quoted string looking like `'M'`, `'M.m'` or `'M.m.p'` (where `M` stands for major version number, `m` stands for minor version number and `p` stands for patch level).

1.3 Expression evaluation

Expressions are written in a reverse polish notation. They are processed as following: evaluator processes tokens one by one.

- *Parenthesis tokens* are mostly ignored (but checked for being balanced).
- *Various string tokens* push single string value to the stack.
- *Number tokens* push integer value to the stack.
- *Variable tokens* put variable value onto the stack.
- *Function tokens* pop some values from the stack, process them using given function and push the result onto the stack.

Some functions referenced by *function tokens* take fixed number of arguments, in this case this predefined number of arguments is popped from the stack. But there are also functions with variable number of arguments (only up to ten arguments are supported). In this case top value in the stack defines number of arguments that will be popped from the stack. Supported numbers: any non-negative integer, `any` string that will take all values on the stack be function arguments and `}`, `}}`, `}}}` and so on string which will make evaluator process the stack until corresponding `{`, `{{`, `{{{` and so on respectively is found. E.g. the following constructs are the same:

```
PRINT MESSAGE (    abc def ghi / 4   JOIN )
PRINT MESSAGE (    abc def ghi / all JOIN )
PRINT MESSAGE (  { abc def ghi / }   JOIN )
PRINT MESSAGE ( {{ abc def ghi / }}  JOIN )
```

All will print `abc/def/ghi`.

1.4 Pattern syntax

Hoodospel uses ERE-like patterns. The following metacharacters are supported:

Single atoms:

`.` Matches any character except for newline.

`[...]`, `[^...]` Collections: matches any (`[...]` form) or none (`[^...]` form) of the characters from the collection.

`(...)` Capturing groups. You may specify up to ten of them.

`^` Start of the line. Zero-width.

$ End of the line. Zero-width.

\... Escape sequence. Escape followed by any of the metacharacters matches this metacharacter literally. Other supported escapes:

Escape	Meaning
\xXX	Byte 0xXX, except for x00: it is not supported.
\e	Escape.
\n	Newline character.
\r	Carriage return character.
\t	Tab character.
\b	Backslash character.

Note: anything else is undefined

Quantifiers:

{N}, {N,}, {N,M} Matches from N to M occurences of preceding atom. First form matches exactly N (M=N), second form matches N or more (M=∞).

***** Matches zero or more occurences of preceding atom.

+ Matches one or more occurence of preceding atom.

? Matches zero or one occurences of preceding atom.

Other:

re1|re2 Branch: matches either re1 or re2.

1.5 Messages

Messages are strings in a printf-like format. That is regular text interleaved with `%{flags}{conversion}` atoms.

Supported flags (they must be given in order below):

+ For numbers: prepend + sign to positive numbers.

For strings: ignored.

− Left-align the converted value. Default is right alignment. Only useful if *field width* was specified.

Convert the value to alternate form. Only meaningful for *x or X* (makes it prepend `0x` to the result), *o* (makes it prepend additional zero unless first resulting character was already zero), *e or E*, *f*, *g or G* (makes it print decimal point even if no digits follow it).

0 Pad value with zeroes instead of spaces.

N or * Specifies field width. `N` is a sequence of decimal digits not starting with `0`. If `*` is specified then width is taken from the next argument.

.N or .* Specifies precision. For *d, i or u*, *x or X* and *o* this specifies minimal number of digits printed, for *e or E* and *f* this specifies the number of digits to appear after the radix character, for *g or G* this specifies the maximum number of significant digits and for *s* this specifies the maximum number of characters.

Supported conversions:

u, i, d Integer argument is converted to decimal notation (signed in case of `%i` and `%d`). Behavior is undefined when trying to use `%u` for negative integers.

o Integer is converted to octal notation. Behavior is undefined when trying to convert negative integers.

x or X Integer is converted to hexadecimal notation. If `X` is used then hexadecimal digits `A` till `F` are capitalized otherwise they are printed in lower case. Behavior is undefined when trying to convert negative integers.

e or E Number is converted to `[-]A.Be±C` scientific notation. If `E` is used then capital letter `E` is used for the exponent, otherwise `e` is used.

f Number is converted to `[-]A.B` decimal notation.

g or G Number is converted to either *scientific notation* or *decimal notation* depending on its value. `G` uses `E` for scientific notation.

s String conversion: embeds given string.

CHAPTER 2

Hoodospel commands

2.1 Block commands

IF *expr* [*OPERATOR* *expr*]
    commands
ELSE_IF *expr* [*OPERATOR* *expr*]
    commands
ELSE
    commands
END_IF

Conditional execution block. If `IF` is used without any suffixes (without *OPERATOR* part) then its argument is considered true as long as it is not empty (for strings) and not zero (for numbers). In any case all of the *expressions* must leave only one value in the stack.

Supported operators:

Operator	Arguments	Is true if first ... the second
IS	string, string	is identical to
IS_NOT	string, string	is different from
MATCHES	string, pattern	matches
NOT_MATCHES	string, pattern	does not match
EQ	number, number	is equal to
NE	number, number	is not equal to
LE	number, number	is lesser then or equal to
GE	number, number	is greater then or equal to
LT	number, number	is lesser then
GT	number, number	is greater then

`ELSE_IF` and `ELSE` sections are optional, there also may be no commands after each of the block headers (`IF`, `ELSE_IF`, `ELSE`). Commands after block header are executed if it is the first block header in a sequence whose condition is true. Commands after `ELSE` will be executed if there are no block headers with condition that is true.

Commands after each block header may also start their own subblocks.

2.2 Non-block commands

PRINT *LEVEL* *message*

Print message with the given level. Given expression may leave more then one value in the stack, in this case the whole stack will be passed to *printf* meaning that first value will be a format string and the following values are being inserted in this string according to it.

cmd:PRINT:message_level option controls which messages are output and which are not.

Supported levels (in order of significance): `DEBUG_INFO`, `MESSAGE`, `WARNING`, `ERROR`.

ABORT *message*

Abort execution. This will abort with a error ID `aborted` and message constructed from *expr* like in *PRINT command*.

VERSION *version*

Check whether current hoodospel version matches given one. It is considered matching if current major version number is identical to requested one and the following numbers are less then or equal to current. If some number is missing it is considered to be zero. E.g.

Current	Requested	Resolution
`1.0.0`	`'0.0.0'`	Fail (major version numbers differ)
`1.0`	`'1.2'`	Fail (zero is lesser then two)
`1.0.2`	`'1.0.3'`	Fail (two is lesser then three)
`1.0`	`'1.0'`	Success
`1.1`	`'1.0'`	Success (one is greater then zero)
`1.0.0`	`'1.0'`	Success (missing number is zero)
`1.0`	`'1.0.0'`	Success (missing number is zero)

RUN_SHELL *expr* [OUTPUT_TO *var*] [INPUT_STRING *expr*] [EXPECTING_EXIT_CODE *expr*] [IGNORE_EXIT_CODE]

Run shell command. Expression given as a first argument is expected to leave more then one string in stack: `RUN_SHELL ( echo abc )` will run command `echo` with argument `abc`, but `RUN_SHELL "echo abc"` will run command `echo abc` with no arguments (and most likely fail).

This command will fail if launched command exits with code different from zero. `EXPECTING_EXIT_CODE` and `IGNORE_EXIT_CODE` prefixes override this behavior: first will make hoodospel expect fail if exit code is different from the one from the prefix argument, first will make hoodospel not fail regardless of exit code.

If `OUTPUT_TO` string was specified then launched command output will be assigned to given variable.

If `INPUT_STRING` string was specified then launched command will receive given string in the stdin.

CHANGE_DIRECTORY_TO *expr*

Change current directory to the given one.

SET *var* TO *expr*

Set given variable value to given value. Expression must leave only one value in the stack. Can also be used to set environment variables, but in this case expression must leave only one *string* value.

DELETE *TYPE* *expr*

Delete given filesystem object. *TYPE* may be `FILE`, `DIRECTORY` and `EMPTY_DIRECTORY`. Expression must leave exactly one string value in the stack.

COPY *TYPE* *expr* (TO *expr* | TO_DIRECTORY *expr* | HERE)
MOVE *TYPE* *expr* (TO *expr* | TO_DIRECTORY *expr* | HERE)

Copy or move given filesystem object to given location. *TYPE* may be either `FILE` or `DIRECTORY`, other prefixes specify target location:

Prefix	Description
TO	`COPY FILE a TO b` copies file contents to file `b`
TO_DIRECTORY	`COPY FILE a TO_DIRECTORY d` copies file contents to `d/a`
HERE	`COPY FILE d/a HERE` copies file contents to file `a`

In all cases expressions must leave exactly one string value in the stack.

CREATE_DIRECTORY *expr* [RECURSIVE]

Create directory with given name. If `RECURSIVE` prefix is given then parent directories are also created if necessary.

SUBSTITUTE *pattern* WITH *expr* IN *var* [IGNORE_CASE] [REPLACE_ALL]

Substitute given pattern with given replacement string. Operates on a given variable, result is recorded back into it. `IGNORE_CASE` flag makes regex engine ignore case, `REPLACE_ALL` makes hoodospel replace all occurences of a pattern (it replaces only the first by default).

WRITE *expr* TO *expr* [TEMP_SUFFIX *expr*]

Write given string to given file. When TEMP_SUFFIX expression is given then in place of writing directly to a given file it will write to *`TO`*`.`*`TEMP_SUFFIX`* file and then rename file that was written to to *`TO`*.

READ *expr* TO *var*

Write given file contents to given variable.

QUESTION *message* RESULT_TO *var* *TYPE* [RESULT_FROM *expr*] [DEFAULT *expr*]

Ask user a question. The result is recorded to the given variable. Question is processed in the following order (assuming key is the first value in message stack):

1. Check out whether there is answer file in the current directory: `.hoodospel.ans`. If there is one then it should have format "*key* `\t` *string*". If there is one and it contains key then RESULT_TO variable is populated with the given answer. This answer is processed according to TYPE.
2. Check out whether RESULT_FROM expression is not empty. If it is not it is proccessed according to given TYPE and used to populate RESULT_TO variable.
3. Check out whether cmd:QUESTION:use_default option is true. If it is then DEFAULT is used to populate the variable. DEFAULT is processed according to TYPE as well.
4. Last, if DEFAULT was not specified and other variants failed user is asked to answer the question. User answer is processed according to TYPE.

Possible types:

Type	Description
BOOLEAN	Transforms "yes", "y", "true" and "1" strings to 1 and "no", "n", "false" and "0" strings to 0.
STRING	Takes string unmodified.

If cmd:QUESTION:write_answers option is true then this command also writes answer to `.hoodospel.ans` file.

CHAPTER 3

Hoodospel functions

***type arg* :EXISTS** If entity of the given type exists pushes one to the stack. Otherwise pushes zero. Possible types:

Type	Description
file	File. VimL implementation checks file for being readable.
directory	Directory.
command	Executable. I.e. `command false :EXISTS` checks whether there is `false` command somewhere in `$PATH`.

:CURRENT_DIRECTORY Pushes full path to the current directory to the stack.

:SEPARATOR Pushes directory separator to the stack (i.e. `/` on *nix systems and `\` on windows).

***str... separator numargs* :JOIN** Joins given strings using given separator and pushes result to the stack. Function with *variable number of arguments*: `abc def / 3 :JOIN` will push `abc/def` to the stack, just like `{ abc def / } :JOIN` will.

:PLATFORM Pushes name of the platform hoodospel is running on. Possible outputs: `qnx`, `vms`, `os2`, `amiga`, `beos`, `mac`, `windows`, `unix`, `other`.

:OS_NAME Pushes less specific name of the platform hoodospel is running on. Possible outputs: `posix`, `nt`, `os2`, `other`.

***str* :SHELL_SPLIT** Pops one value from the stack, splits it on unescaped spaces, unescapes (replaces all `\.` with `.`) and pops resulting values back onto the stack.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search