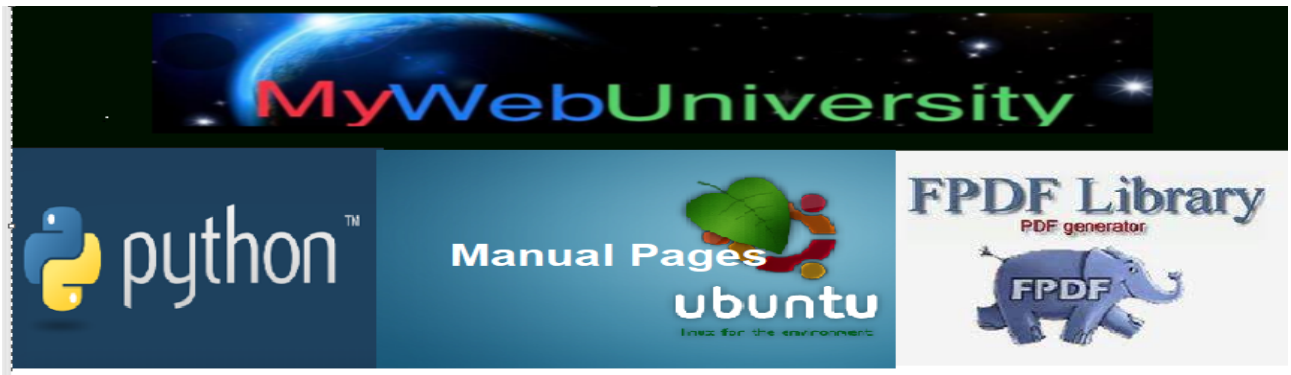

Full credit is given to the above companies including the OS that this PDF file was generated!

Linux Ubuntu 26.04 Manual Pages on command 'fish-language.1'

$ man fish-language.1

FISH-LANGUAGE(1) fish-shell FISH-LANGUAGE(1)

This document is a comprehensive overview of fish's scripting language.

For interactive features see Interactive use <>.

SYNTAX OVERVIEW

Shells like fish are used by giving them commands. A command is executed by writing the name of the command followed by any arguments. For example:

```
echo hello world
```

echo <> command writes its arguments to the screen. In this example the output is hello world.

Everything in fish is done with commands. There are commands for repeating other commands, commands for assigning variables, commands for treating a group of commands as a single com? mand, etc. All of these commands follow the same basic syntax.

Every program on your computer can be used as a command in fish. If the program file is lo? cated in one of the PATH directories, you can just type the name of the program to use it. Otherwise the whole filename, including the directory (like /home/me/code/checkers/checkers or ../checkers) is required.

Here is a list of some useful commands:

? cd <>: Change the current directory

? ls: List files and directories

? man: Display a manual page - try man ls to get help on your "ls" command, or man mv to get information about "mv".

? mv: Move (rename) files

? cp: Copy files

? open <>: Open files with the default application associated with each filetype

? less: Display the contents of files

Commands and arguments are separated by the space character ' '. Every command ends with ei?
ther a newline (by pressing the return key) or a semicolon ;. Multiple commands can be writ?
ten on the same line by separating them with semicolons.

A switch is a very common special type of argument. Switches almost always start with one or more hyphens - and alter the way a command operates. For example, the ls command usually lists the names of all files and directories in the current working directory. By using the -l switch, the behavior of ls is changed to not only display the filename, but also the size, permissions, owner, and modification time of each file.

Switches differ between commands and are usually documented on a command's manual page. There are some switches, however, that are common to most commands. For example, --help will usually display a help text, --version will usually display the command version, and -i will often turn on interactive prompting before taking action. Try man your-command-here to get information on your command's switches.

So the basic idea of fish is the same as with other unix shells: It gets a commandline, runs expansions, and the result is then run as a command.

TERMINOLOGY

Here we define some of the terms used on this page and throughout the rest of the fish docu?
mentation:

? Argument: A parameter given to a command. In echo foo, the "foo" is an argument.

? Builtin: A command that is implemented by the shell. Builtins are so closely tied to the operation of the shell that it is impossible to implement them as external commands. In echo foo, the "echo" is a builtin.

? Command: A program that the shell can run, or more specifically an external program that the shell runs in another process. External commands are provided on your system, as exe? cutable files. In echo foo the "echo" is a builtin command, in command echo foo the "echo" is an external command, provided by a file like /bin/echo.

? Function: A block of commands that can be called as if they were a single command. By us? ing functions, it is possible to string together multiple simple commands into one more advanced command.

? Job: A running pipeline or command.

? Pipeline: A set of commands strung together so that the output of one command is the input

of the next command. echo foo | grep foo is a pipeline.

? Redirection: An operation that changes one of the input or output streams associated with a job.

? Switch or Option: A special kind of argument that alters the behavior of a command. A switch almost always begins with one or two hyphens. In echo -n foo the "-n" is an option.

QUOTES

Sometimes you want to give a command an argument that contains characters special to fish, like spaces or $ or *. To do that, you can use quotes:

rm "my file.txt"

to remove a file called my file.txt instead of trying to remove two files, my and file.txt.

Fish understands two kinds of quotes: Single (') and double ("), and both work slightly dif? ferently.

Between single quotes, fish performs no expansions. Between double quotes, fish only per? forms variable expansion and command substitution in the $(command). No other kind of expan? sion (including brace expansion or parameter expansion) is performed, and escape sequences (for example, \n) are ignored. Within quotes, whitespace is not used to separate arguments, allowing quoted arguments to contain spaces.

The only meaningful escape sequences in single quotes are \', which escapes a single quote and \\, which escapes the backslash symbol. The only meaningful escapes in double quotes are \", which escapes a double quote, \$, which escapes a dollar character, \ followed by a new? line, which deletes the backslash and the newline, and \\, which escapes the backslash sym? bol.

Single quotes have no special meaning within double quotes and vice versa.

More examples:

grep 'enabled)$' foo.txt

searches for lines ending in enabled) in foo.txt (the $ is special to grep: it matches the end of the line).

apt install "postgres-*"

installs all packages with a name starting with "postgres-", instead of looking through the current directory for files named "postgres-something".

ESCAPING CHARACTERS

Some characters cannot be written directly on the command line. For these characters, so-called escape sequences are provided. These are:

? \a represents the alert character.

? \e represents the escape character.

? \f represents the form feed character.

? \n represents a newline character.

? \r represents the carriage return character.

? \t represents the tab character.

? \v represents the vertical tab character.

? \xHH or \XHH, where HH is a hexadecimal number, represents a byte of data with the speci?
fied value. For example, \x9 is the tab character. If you are using a multibyte encoding,
this can be used to enter invalid strings. Typically fish is run with the ASCII or UTF-8
encoding, so anything up to \X7f is an ASCII character.

? \ooo, where ooo is an octal number, represents the ASCII character with the specified
value. For example, \011 is the tab character. The highest allowed value is \177.

? \uXXXX, where XXXX is a hexadecimal number, represents the 16-bit Unicode character with
the specified value. For example, \u9 is the tab character.

? \UXXXXXXXX, where XXXXXXXX is a hexadecimal number, represents the 32-bit Unicode charac?
ter with the specified value. For example, \U9 is the tab character. The highest allowed
value is U10FFFF.

? \cX, where X is a letter of the alphabet, represents the control sequence generated by
pressing the control key and the specified letter. For example, \ci is the tab character

Some characters have special meaning to the shell. For example, an apostrophe ' disables ex?
pansion (see Quotes). To tell the shell to treat these characters literally, escape them
with a backslash. For example, the command:

```
echo \'hello world\'
```

outputs 'hello world' (including the apostrophes), while the command:

```
echo 'hello world'
```

outputs hello world (without the apostrophes). In the former case the shell treats the apos?
trophes as literal ' characters, while in the latter case it treats them as special expan?
sion modifiers.

The special characters and their escape sequences are:

? \ (backslash space) escapes the space character. This keeps the shell from splitting ar?
guments on the escaped space.

? \$ escapes the dollar character.

? \\ escapes the backslash character.

? * escapes the star character.

? \? escapes the question mark character (this is not necessary if the qmark-noglob feature flag is enabled).

? \~ escapes the tilde character.

? \# escapes the hash character.

? \(escapes the left parenthesis character.

? \) escapes the right parenthesis character.

? \{ escapes the left curly bracket character.

? \} escapes the right curly bracket character.

? \[escapes the left bracket character.

? \] escapes the right bracket character.

? \< escapes the less than character.

? \> escapes the more than character.

? \& escapes the ampersand character.

? \| escapes the vertical bar character.

? \; escapes the semicolon character.

? \" escapes the quote character.

? \' escapes the apostrophe character.

As a special case, \ immediately followed by a literal new line is a "continuation" and tells fish to ignore the line break and resume input at the start of the next line (without introducing any whitespace or terminating a token).

INPUT/OUTPUT REDIRECTION

Most programs use three input/output (I/O) streams:

? Standard input (stdin) for reading. Defaults to reading from the keyboard.

? Standard output (stdout) for writing output. Defaults to writing to the screen.

? Standard error (stderr) for writing errors and warnings. Defaults to writing to the screen.

Each stream has a number called the file descriptor (FD): 0 for stdin, 1 for stdout, and 2 for stderr.

The destination of a stream can be changed using something called redirection. For example, echo hello > output.txt, redirects the standard output of the echo command to a text file.

? To read standard input from a file, use <SOURCE_FILE.

? To read standard input from a file or /dev/null if it can't be read, use <?SOURCE_FILE.

? To write standard output to a file, use >DESTINATION.

? To write standard error to a file, use 2>DESTINATION. [1]

? To append standard output to a file, use >>DESTINATION_FILE.

? To append standard error to a file, use 2>>DESTINATION_FILE.

? To not overwrite ("clobber") an existing file, use >?DESTINATION or 2>?DESTINATION. This is known as the "noclobber" redirection.

DESTINATION can be one of the following:

? A filename to write the output to. Often >/dev/null to silence output by writing it to the special "sinkhole" file.

? An ampersand (&) followed by the number of another file descriptor like &2 for standard error. The output will be written to the destination descriptor.

? An ampersand followed by a minus sign (&-). The file descriptor will be closed. Note: This may cause the program to fail because its writes will be unsuccessful.

As a convenience, the redirection &> can be used to direct both stdout and stderr to the same destination. For example, echo hello &> all_output.txt redirects both stdout and stderr to the file all_output.txt. This is equivalent to echo hello > all_output.txt 2>&1. You can also use &>> to append both stdout and stderr to the same destination.

Any arbitrary file descriptor can be used in a redirection by prefixing the redirection with the FD number.

? To redirect the input of descriptor N, use N<DESTINATION.

? To redirect the output of descriptor N, use N>DESTINATION.

? To append the output of descriptor N to a file, use N>>DESTINATION_FILE.

File descriptors cannot be used with a <? input redirection, only a regular < one.

For example:

```
  # Write `foo`'s standard error (file descriptor 2)
  # to a file called "output.stderr":
  foo 2> output.stderr
  # if $num doesn't contain a number,
  # this test will be false and print an error,
  # so by ignoring the error we can be sure that we're dealing
  # with a number in the "if" block:
  if test "$num" -gt 2 2>/dev/null
```

```
    # do things with $num as a number greater than 2
else
    # do things if $num is <= 2 or not a number
end
# Save `make`s output in a file:
make &>/log
# Redirections stack and can be used with blocks:
begin
    echo stdout
    echo stderr >&2 # <- this goes to stderr!
end >/dev/null # ignore stdout, so this prints "stderr"
# print all lines that include "foo" from myfile, or nothing if it doesn't exist.
string match '*foo*' <?myfile
```

It is an error to redirect a builtin, function, or block to a file descriptor above 2. How? ever this is supported for external commands.

[1] Previous versions of fish also allowed specifying this as ^DESTINATION, but that made another character special so it was deprecated and removed. See feature flags.

PIPING

Another way to redirect streams is a pipe. A pipe connects streams with each other. Usually the standard output of one command is connected with the standard input of another. This is done by separating commands with the pipe character |. For example:

```
cat foo.txt | head
```

The command cat foo.txt sends the contents of foo.txt to stdout. This output is provided as input for the head program, which prints the first 10 lines of its input.

It is possible to pipe a different output file descriptor by prepending its FD number and the output redirect symbol to the pipe. For example:

```
make fish 2>| less
```

will attempt to build fish, and any errors will be shown using the less pager. [2]

As a convenience, the pipe &| redirects both stdout and stderr to the same process. This is different from bash, which uses |&.

[2] A "pager" here is a program that takes output and "paginates" it. less doesn't just do pages, it allows arbitrary scrolling (even back!).

COMBINING PIPES AND REDIRECTIONS

It is possible to use multiple redirections and a pipe at the same time. In that case, they are read in this order:

1. First the pipe is set up.

2. Then the redirections are evaluated from left-to-right.

This is important when any redirections reference other file descriptors with the &N syntax. When you say >&2, that will redirect stdout to where stderr is pointing to at that time.

Consider this helper function:

```
# Make a function that prints something to stdout and stderr
function print
    echo out
    echo err >&2
end
```

Now let's see a few cases:

```
# Redirect both stderr and stdout to less
print 2>&1 | less
# or
print &| less
# Show the "out" on stderr, silence the "err"
print >&2 2>/dev/null
# Silence both
print >/dev/null 2>&1
```

JOB CONTROL

When you start a job in fish, fish itself will pause, and give control of the terminal to the program it just started. Sometimes, you want to continue using the commandline, and have the job run in the background. To create a background job, append an & (ampersand) to your command. This will tell fish to run the job in the background. Background jobs are very use? ful when running programs that have a graphical user interface.

Example:

```
emacs &
```

will start the emacs text editor in the background. fg <> can be used to bring it into the foreground again when needed.

Most programs allow you to suspend the program's execution and return control to fish by pressing ctrl-z (also referred to as ^Z). Once back at the fish commandline, you can start

other programs and do anything you want. If you then want you can go back to the suspended command by using the fg <> (foreground) command.

If you instead want to put a suspended job into the background, use the bg <> command.

To get a listing of all currently started jobs, use the jobs <> command. These listed jobs can be removed with the disown <> command.

At the moment, functions cannot be started in the background. Functions that are stopped and then restarted in the background using the bg <> command will not execute correctly.

If the & character is followed by a non-separating character, it is not interpreted as back? ground operator. Separating characters are whitespace and the characters ;<>&|.

FUNCTIONS

Functions are programs written in the fish syntax. They group together various commands and their arguments using a single name.

For example, here's a simple function to list directories:

```
function ll
    ls -l $argv
end
```

The first line tells fish to define a function by the name of ll, so it can be used by writ? ing ll on the commandline. The second line tells fish that the command ls -l $argv should be called when ll is invoked. $argv is a list variable, which always contains all arguments sent to the function. In the example above, these are passed on to the ls command. The end on the third line ends the definition.

Calling this as ll /tmp/ will end up running ls -l /tmp/, which will list the contents of /tmp.

This is a kind of function known as an alias.

Fish's prompt is also defined in a function, called fish_prompt <>. It is run when the prompt is about to be displayed and its output forms the prompt:

```
function fish_prompt
    # A simple prompt. Displays the current directory
    # (which fish stores in the $PWD variable)
    # and then a user symbol - a '?' for a normal user and a '#' for root.
    set -l user_char '?'
    if fish_is_root_user
        set user_char '#'
```

```
    end
    echo (set_color yellow)$PWD (set_color purple)$user_char
  end
```

To edit a function, you can use funced <>, and to save a function funcsave <>. This will store it in a function file that fish will autoload when needed.

The functions <> builtin can show a function's current definition (and type <> will also do if given a function).

For more information on functions, see the documentation for the function <> builtin.

Defining aliases

One of the most common uses for functions is to slightly alter the behavior of an already existing command. For example, one might want to redefine the ls command to display colors. The switch for turning on colors on GNU systems is --color=auto. An alias around ls might look like this:

```
  function ls
    command ls --color=auto $argv
  end
```

There are a few important things that need to be noted about aliases:

? Always take care to add the $argv variable to the list of parameters to the wrapped com? mand. This makes sure that if the user specifies any additional parameters to the func? tion, they are passed on to the underlying command.

? If the alias has the same name as the aliased command, you need to prefix the call to the program with command to tell fish that the function should not call itself, but rather a command with the same name. If you forget to do so, the function would call itself until the end of time. Usually fish is smart enough to figure this out and will refrain from do? ing so (which is hopefully in your interest).

To easily create a function of this form, you can use the alias <> command. Unlike other shells, this just makes functions - fish has no separate concept of an "alias", we just use the word for a simple wrapping function like this. alias <> immediately creates a function.

Consider using alias --save or funcsave <> to save the created function into an autoload file instead of recreating the alias each time.

For an alternative, try abbreviations <#abbreviations>. These are words that are expanded while you type, instead of being actual functions inside the shell.

Autoloading functions

Functions can be defined on the commandline or in a configuration file, but they can also be automatically loaded. This has some advantages:

? An autoloaded function becomes available automatically to all running shells.

? If the function definition is changed, all running shells will automatically reload the altered version, after a while.

? Startup time and memory usage is improved, etc.

When fish needs to load a function, it searches through any directories in the list variable $fish_function_path for a file with a name consisting of the name of the function plus the suffix .fish and loads the first it finds.

For example if you try to execute something called banana, fish will go through all directo? ries in $fish_function_path looking for a file called banana.fish and load the first one it finds.

By default $fish_function_path contains the following:

? A directory for users to keep their own functions, usually ~/.config/fish/functions (con? trolled by the XDG_CONFIG_HOME environment variable).

? A directory for functions for all users on the system, usually /etc/fish/functions (really $__fish_sysconfdir/functions).

? Directories for other software to put their own functions. These are in the directories under $__fish_user_data_dir (usually ~/.local/share/fish, controlled by the XDG_DATA_HOME environment variable) and in the XDG_DATA_DIRS environment variable, in a subdirectory called fish/vendor_functions.d. The default value for XDG_DATA_DIRS is usually /usr/share/fish/vendor_functions.d and /usr/local/share/fish/vendor_functions.d.

If you are unsure, your functions probably belong in ~/.config/fish/functions.

As we've explained, autoload files are loaded by name, so, while you can put multiple func? tions into one file, the file will only be loaded automatically once you try to execute the one that shares the name.

Autoloading also won't work for event handlers, since fish cannot know that a function is supposed to be executed when an event occurs when it hasn't yet loaded the function. See the event handlers section for more information.

If a file of the right name doesn't define the function, fish will not read other autoload files, instead it will go on to try builtins and finally commands. This allows masking a function defined later in $fish_function_path, e.g. if your administrator has put something into /etc/fish/functions that you want to skip.

If you are developing another program and want to install fish functions for it, install them to the "vendor" functions directory. As this path varies from system to system, you can use pkgconfig to discover it with the output of pkg-config --variable functionsdir fish. Your installation system should support a custom path to override the pkgconfig path, as other distributors may need to alter it easily.

COMMENTS

Anything after a # until the end of the line is a comment. That means it's purely for the reader's benefit, fish ignores it.

This is useful to explain what and why you are doing something:

```
function ls
    # The function is called ls,
    # so we have to explicitly call `command ls` to avoid calling ourselves.
    command ls --color=auto $argv
end
```

There are no multiline comments. If you want to make a comment span multiple lines, start each line with a #.

Comments can also appear after a line like so:

```
set -gx EDITOR emacs # I don't like vim.
```

CONDITIONS

Fish has some builtins that let you execute commands only if a specific criterion is met: if <>, switch <>, and <> and or <>, and also the familiar &&/|| syntax.

The if statement

The if <> statement runs a block of commands if the condition was true.

Like other shells, but unlike typical programming languages you might know, the condition here is a command. Fish runs it, and if it returns a true exit status (that's 0), the if-block is run. For example:

```
if test -e /etc/os-release
    cat /etc/os-release
end
```

This uses the test <> command to see if the file /etc/os-release exists. If it does, it runs cat, which prints it on the screen.

Unlike other shells, the condition command ends after the first job, there is no then here.

Combiners like and and or extend the condition.

A more complicated example with a command substitution:

```
if test "$(uname)" = Linux
    echo I like penguins
end
```

Because test can be used for many different tests, it is important to quote variables and command substitutions. If the $(uname) was not quoted, and uname printed nothing it would run test = Linux, which is an error.

if can also take else if clauses with additional conditions and an else <> clause that is executed when everything else was false:

```
if test "$number" -gt 10
   echo Your number was greater than 10
else if test "$number" -gt 5
   echo Your number was greater than 5
else if test "$number" -gt 1
   echo Your number was greater than 1
else
   echo Your number was smaller or equal to 1
end
```

The not <> keyword can be used to invert the status:

```
# Check if the file contains the string "fish" anywhere.
# This executes the `grep` command, which searches for a string,
# and if it finds it returns a status of 0.
# The `not` then turns 0 into 1 or anything else into 0.
# The `-q` switch stops it from printing any matches.
if not grep -q fish myanimals
    echo "You don't have fish!"
else
    echo "You have fish!"
end
```

Other things commonly used in if-conditions:

? contains <> - to see if a list contains a specific element (if contains -- /usr/bin $PATH)

? string <> - to e.g. match strings (if string match -q -- '*-' $arg)

? path <> - to check if paths of some criteria exist (if path is -rf -- ~/.config/fish/con?

fig.fish)

? type <> - to see if a command, function or builtin exists (if type -q git)

The switch statement

The switch <> command is used to execute one of possibly many blocks of commands depending on the value of a string. It can take multiple case <> blocks that are executed when the string matches. They can take wildcards. For example:

```
switch (uname)
case Linux
    echo Hi Tux!
case Darwin
    echo Hi Hexley!
case DragonFly '*BSD'
    echo Hi Beastie! # this also works for FreeBSD and NetBSD
case '*'
    echo Hi, stranger!
end
```

Unlike other shells or programming languages, there is no fallthrough - the first matching case block is executed and then control jumps out of the switch.

Combiners (and / or / && / ||)

For simple checks, you can use combiners. and <> or && run the second command if the first succeeded, while or <> or || run it if the first failed. For example:

```
# $XDG_CONFIG_HOME is a standard place to store configuration.
# If it's not set applications should use ~/.config.
set -q XDG_CONFIG_HOME; and set -l configdir $XDG_CONFIG_HOME
or set -l configdir ~/.config
```

Note that combiners are lazy - only the part that is necessary to determine the final status is run.

Compare:

```
if sleep 2; and false
    echo 'How did I get here? This should be impossible'
end
```

and:

```
if false; and sleep 2
```

```
    echo 'How did I get here? This should be impossible'
  end
```

These do essentially the same thing, but the former takes 2 seconds longer because the sleep always needs to run.

Or you can have a case where it is necessary to stop early:

```
  if command -sq foo; and foo
```

If this went on after seeing that the command "foo" doesn't exist, it would try to run foo and error because it wasn't found!

Combiners execute step-by-step, so it isn't recommended to build longer chains of them be? cause they might do something you don't want. Consider:

```
  test -e /etc/my.config
  or echo "OH NO WE NEED A CONFIG FILE"
  and return 1
```

This will execute return 1 also if the test succeeded. This is because fish runs test -e /etc/my.config, sets $status to 0, then skips the echo, keeps $status at 0, and then exe? cutes the return 1 because $status is still 0.

So if you have more complex conditions or want to run multiple things after something failed, consider using an if. Here that would be:

```
  if not test -e /etc/my.config
    echo "OH NO WE NEED A CONFIG FILE"
    return 1
  end
```

LOOPS AND BLOCKS

Like most programming language, fish also has the familiar while <> and for <> loops.

while works like a repeated if:

```
  while true
    echo Still running
    sleep 1
  end
```

will print "Still running" once a second. You can abort it with ctrl-c.

for loops work like in other shells, which is more like python's for-loops than e.g. C's:

```
  for file in *
    echo file: $file
```

```
end
```

will print each file in the current directory. The part after the in is a list of arguments, so you can use any expansions there:

```
set moreanimals bird fox
for animal in {cat,}fish dog $moreanimals
  echo I like the $animal
end
```

If you need a list of numbers, you can use the seq command to create one:

```
for i in (seq 1 5)
   echo $i
end
```

break <> is available to break out of a loop, and continue <> to jump to the next iteration.

Input and output redirections (including pipes) can also be applied to loops:

```
while read -l line
   echo line: $line
end < file
```

In addition there's a begin <> block that just groups commands together so you can redirect to a block or use a new variable scope without any repetition:

```
begin
  set -l foo bar # this variable will only be available in this block!
end
```

PARAMETER EXPANSION

When fish is given a commandline, it expands the parameters before sending them to the com? mand. There are multiple different kinds of expansions:

? Wildcards, to create filenames from patterns - *.jpg

? Variable expansion, to use the value of a variable - $HOME

? Command substitution, to use the output of another command - $(cat /path/to/file)

? Brace expansion, to write lists with common pre- or suffixes in a shorter way {/usr,}/bin

? Tilde expansion, to turn the ~ at the beginning of paths into the path to the home direc? tory ~/bin

Parameter expansion is limited to 524288 items. There is a limit to how many arguments the operating system allows for any command, and 524288 is far above it. This is a measure to stop the shell from hanging doing useless computation.

Wildcards ("Globbing")

When a parameter includes an unquoted * star (or "asterisk") or a ? question mark, fish uses it as a wildcard to match files.

? * matches any number of characters (including zero) in a file name, not including /.

? ** matches any number of characters (including zero), and also descends into subdirecto? ries. If ** is a segment by itself, that segment may match zero times, for compatibility with other shells.

? ? can match any single character except /. This is deprecated and can be disabled via the qmark-noglob feature flag, so ? will be an ordinary character.

Wildcard matches are sorted case insensitively. When sorting matches containing numbers, they are naturally sorted, so that the strings '1' '5' and '12' would be sorted like 1, 5, 12.

Hidden files (where the name begins with a dot) are not considered when wildcarding unless the wildcard string has a dot in that place.

Examples:

? a* matches any files beginning with an 'a' in the current directory.

? ** matches any files and directories in the current directory and all of its subdirecto? ries.

? ~/.* matches all hidden files (also known as "dotfiles") and directories in your home di? rectory.

For most commands, if any wildcard fails to expand, the command is not executed, $status is set to nonzero, and a warning is printed. This behavior is like what bash does with shopt -s failglob. There are exceptions, namely set <> and path <>, overriding variables in over? rides, count <> and for <>. Their globs will instead expand to zero arguments (so the com? mand won't see them at all), like with shopt -s nullglob in bash.

Examples:

```
# List the .foo files, or warns if there aren't any.
ls *.foo
# List the .foo files, if any.
set foos *.foo
if count $foos >/dev/null
    ls $foos
end
```

Unlike bash (by default), fish will not pass on the literal glob character if no match was found, so for a command like apt install that does the matching itself, you need to add quotes:

```
apt install "ncurses-*"
```

Variable expansion

One of the most important expansions in fish is the "variable expansion". This is the re? placing of a dollar sign ($) followed by a variable name with the _value_ of that variable.

A simple example:

```
echo $HOME
```

which will replace $HOME with the home directory of the current user, and pass it to echo <>, which will then print it.

Some variables like $HOME are already set because fish sets them by default or because fish's parent process passed them to fish when it started it. You can define your own vari? ables by setting them with set <>:

```
set my_directory /home/cooluser/mystuff
ls $my_directory
# shows the contents of /home/cooluser/mystuff
```

For more on how setting variables works, see Shell variables and the following sections.

Sometimes a variable has no value because it is undefined or empty, and it expands to noth? ing:

```
echo $nonexistentvariable
# Prints no output.
```

To separate a variable name from text you can encase the variable within double-quotes or braces:

```
set WORD cat
echo The plural of $WORD is "$WORD"s
# Prints "The plural of cat is cats" because $WORD is set to "cat".
echo The plural of $WORD is {$WORD}s
# ditto
```

Without the quotes or braces, fish will try to expand a variable called $WORDs, which may not exist.

The latter syntax {$WORD} is a special case of brace expansion.

If $WORD here is undefined or an empty list, the "s" is not printed. However, it is printed

if $WORD is the empty string (like after set WORD "").

For more on shell variables, read the Shell variables section.

Quoting variables

Variable expansion also happens in double quoted strings. Inside double quotes ("these"), variables will always expand to exactly one argument. If they are empty or undefined, it will result in an empty string. If they have one element, they'll expand to that element. If they have more than that, the elements will be joined with spaces, unless the variable is a path variable - in that case it will use a colon (:) instead [3].

Fish variables are all lists, and they are split into elements when they are set - that means it is important to decide whether to use quotes or not with set <>:

```
set foo 1 2 3 # a variable with three elements
rm $foo # runs the equivalent of `rm 1 2 3` - trying to delete three files: 1, 2 and 3.
rm "$foo" # runs `rm '1 2 3'` - trying to delete one file called '1 2 3'
set foo # an empty variable
rm $foo # runs `rm` without arguments
rm "$foo" # runs the equivalent of `rm ''`
set foo "1 2 3"
rm $foo # runs the equivalent of `rm '1 2 3'` - trying to delete one file
rm "$foo" # same thing
```

This is unlike other shells, which do what is known as "Word Splitting", where they split the variable when it is used in an expansion. E.g. in bash:

```
foo="1 2 3"
rm $foo # runs the equivalent of `rm 1 2 3`
rm "$foo" # runs the equivalent of `rm '1 2 3'`
```

This is the cause of very common problems with filenames with spaces in bash scripts.

In fish, unquoted variables will expand to as many arguments as they have elements. That means an empty list will expand to nothing, a variable with one element will expand to that element, and a variable with multiple elements will expand to each of those elements sepa? rately.

If a variable expands to nothing, it will cancel out any other strings attached to it. See the Combining Lists section for more information.

Most of the time, not quoting a variable is correct. The exception is when you need to en? sure that the variable is passed as one element, even if it might be unset or have multiple

elements. This happens often with test <>:

```
set -l foo one two three
test -n $foo
# prints an error that it got too many arguments, because it was executed like
test -n one two three
test -n "$foo"
# works, because it was executed like
test -n "one two three"
```

[3] Unlike bash or zsh, which will join with the first character of $IFS (which usually is space).

Dereferencing variables

The $ symbol can also be used multiple times, as a kind of "dereference" operator (the * in C or C++), like in the following code:

```
set foo a b c
set a 10; set b 20; set c 30
for i in (seq (count $$foo))
    echo $$foo[$i]
end
# Output is:
# 10
# 20
# 30
```

$$foo[$i] is "the value of the variable named by $foo[$i]".

This can also be used to give a variable name to a function:

```
function print_var
    for arg in $argv
        echo Variable $arg is $$arg
    end
end
set -g foo 1 2 3
set -g bar a b c
print_var foo bar
# prints "Variable foo is 1 2 3" and "Variable bar is a b c"
```

Of course the variable will have to be accessible from the function, so it needs to be global/universal or exported. It also can't clash with a variable name used inside the function. So if we had made $foo there a local variable, or if we had named it "arg" instead, it would not have worked.

When using this feature together with slices, the slices will be used from the inside out. $$foo[5] will use the fifth element of $foo as a variable name, instead of giving the fifth element of all the variables $foo refers to. That would instead be expressed as $$foo[1..-1][5] (take all elements of $foo, use them as variable names, then give the fifth element of those).

Some more examples:

```
set listone 1 2 3
set listtwo 4 5 6
set var listone listtwo
echo $$var
# Output is 1 2 3 4 5 6
echo $$var[1]
# Output is 1 2 3
echo $$var[2][3]
# $var[2] is listtwo, third element of that is 6, output is 6
echo $$var[..][2]
# The second element of every variable, so output is 2 5
```

Variables as command

Like other shells, you can run the value of a variable as a command.

```
> set -g EDITOR emacs
> $EDITOR foo # opens emacs, possibly the GUI version
```

If you want to give the command an argument inside the variable it needs to be a separate element:

```
> set EDITOR emacs -nw
> $EDITOR foo # opens emacs in the terminal even if the GUI is installed
> set EDITOR "emacs -nw"
> $EDITOR foo # tries to find a command called "emacs -nw"
```

Also like other shells, this only works with commands, builtins and functions - it will not work with keywords because they have syntactical importance.

For instance set if $if won't allow you to make an if-block, and set cmd command won't allow you to use the command <> decorator, but only uses like $cmd -q foo.

Command substitution

A command substitution is an expansion that uses the output of a command as the arguments to another. For example:

```
echo $(pwd)
```

This executes the pwd <> command, takes its output (more specifically what it wrote to the standard output "stdout" stream) and uses it as arguments to echo <>. So the inner command (the pwd) is run first and has to complete before the outer command can even be started.

If the inner command prints multiple lines, fish will use each separate line as a separate argument to the outer command. Unlike other shells, the value of $IFS is not used [4], fish splits on newlines.

Command substitutions can also be double-quoted:

```
echo "$(pwd)"
```

When using double quotes, the command output is not split up by lines, but trailing empty lines are still removed.

If the output is piped to string split or string split0 <> as the last step, those splits are used as they appear instead of splitting lines.

Fish also allows spelling command substitutions without the dollar, like echo (pwd). This variant will not be expanded in double-quotes (echo "(pwd)" will print (pwd)).

The exit status of the last run command substitution is available in the status variable if the substitution happens in the context of a set <> command (so if set -l (something) checks if something returned true).

To use only some lines of the output, refer to slices.

Examples:

```
# Outputs 'image.png'.
echo (basename image.jpg .jpg).png
# Convert all JPEG files in the current directory to the
# PNG format using the 'convert' program.
for i in *.jpg; convert $i (basename $i .jpg).png; end
# Set the ``data`` variable to the contents of 'data.txt'
# without splitting it into a list.
set data "$(cat data.txt)"
```

```
# Set ``$data`` to the contents of data, splitting on NUL-bytes.
set data (cat data | string split0)
```

Sometimes you want to pass the output of a command to another command that only accepts files. If it's just one file, you can usually pass it via a pipe, like:

```
grep fish myanimallist1 | wc -l
```

but if you need multiple or the command doesn't read from standard input, "process substitu? tion" is useful. Other shells allow this via foo <(bar) <(baz), and fish uses the psub <> command:

```
# Compare only the lines containing "fish" in two files:
diff -u (grep fish myanimallist1 | psub) (grep fish myanimallist2 | psub)
```

This creates a temporary file, stores the output of the command in that file and prints the filename, so it is given to the outer command.

Fish has a default limit of 1 GiB on the data it will read in a command substitution. If that limit is reached the command (all of it, not just the command substitution - the outer command won't be executed at all) fails and $status is set to 122. This is so command sub? stitutions can't cause the system to go out of memory, because typically your operating sys? tem has a much lower limit, so reading more than that would be useless and harmful. This limit can be adjusted with the fish_read_limit variable (0 meaning no limit). This limit also affects the read <> command.

[4] One exception: Setting $IFS to empty will disable line splitting. This is deprecated, use string split <> instead.

Brace expansion

Curly braces can be used to write comma-separated lists. They will be expanded with each el? ement becoming a new parameter, with the surrounding string attached. This is useful to save on typing, and to separate a variable name from surrounding text.

Examples:

```
> echo input.{c,h,txt}
input.c input.h input.txt
# Move all files with the suffix '.c' or '.h' to the subdirectory src.
> mv *.{c,h} src/
# Make a copy of `file` at `file.bak`.
> cp file{,.bak}
> set -l dogs hot cool cute "good "
```

```
> echo {$dogs}dog
hotdog cooldog cutedog good dog
```

If there is no "," or variable expansion between the curly braces, they will not be ex?panded:

```
# This {} isn't special
> echo foo-{}
foo-{}
# This passes "HEAD@{2}" to git
> git reset --hard HEAD@{2}
> echo {{a,b}}
{a} {b} # because the inner brace pair is expanded, but the outer isn't.
```

If after expansion there is nothing between the braces, the argument will be removed (see the Combining Lists section):

```
> echo foo-{$undefinedvar}
# Output is an empty line, like a bare `echo`.
```

If there is nothing between a brace and a comma or two commas, it's interpreted as an empty element:

```
> echo {,,/usr}/bin
/bin /bin /usr/bin
```

To use a "," as an element, quote or escape it.

The very first character of a command token is never interpreted as expanding brace, because it's the beginning of a compound statement <>:

```
> {echo hello, && echo world}
hello,
world
```

Combining lists

Fish expands lists like brace expansions:

```
>_ set -l foo x y z
>_ echo 1$foo
# Any element of $foo is combined with the "1":
1x 1y 1z
>_ echo {good,bad}" apples"
# Any element of the {} is combined with the " apples":
```

```
good apples bad apples
# Or we can mix the two:
>_ echo {good,bad}" "$foo
good x bad x good y bad y good z bad z
```

Any string attached to a list will be concatenated to each element.

Two lists will be expanded in all combinations - every element of the first with every ele?
ment of the second:

```
>_ set -l a x y z; set -l b 1 2 3
>_ echo $a$b # same as {x,y,z}{1,2,3}
x1 y1 z1 x2 y2 z2 x3 y3 z3
```

A result of this is that, if a list has no elements, this combines the string with no ele?
ments, which means the entire token is removed!

```
>_ set -l c # <- this list is empty!
>_ echo {$c}word
# Output is an empty line - the "word" part is gone
```

This can be quite useful. For example, if you want to go through all the files in all the
directories in PATH, use

```
for file in $PATH/*
```

Because PATH is a list, this expands to all the files in all the directories in it. And if
there are no directories in PATH, the right answer here is to expand to no files.

Sometimes this may be unwanted, especially that tokens can disappear after expansion. In
those cases, you should double-quote variables - echo "$c"word.

This also happens after command substitution. To avoid tokens disappearing there, make the
inner command return a trailing newline, or double-quote it:

```
>_ set b 1 2 3
>_ echo (echo x)$b
x1 x2 x3
>_ echo (printf '%s' '')banana
# the printf prints nothing, so this is nothing times "banana",
# which is nothing.
>_ echo (printf '%s\n' '')banana
# the printf prints a newline,
# so the command substitution expands to an empty string,
```

```
# so this is `"banana`
banana
>_ echo "$(printf '%s' ")"banana
# quotes mean this is one argument, the banana stays
```

Slices

Sometimes it's necessary to access only some of the elements of a list (all fish variables are lists), or some of the lines a command substitution outputs. Both are possible in fish by writing a set of indices in brackets, like:

```
# Make $var a list of four elements
set var one two three four
# Print the second:
echo $var[2]
# prints "two"
# or print the first three:
echo $var[1..3]
# prints "one two three"
```

In index brackets, fish understands ranges written like a..b ('a' and 'b' being indices). They are expanded into a sequence of indices from a to b (so a a+1 a+2 ... b), going up if b is larger and going down if a is larger. Negative indices can also be used - they are taken from the end of the list, so -1 is the last element, and -2 the one before it. If an index doesn't exist the range is clamped to the next possible index.

If a list has 5 elements the indices go from 1 to 5, so a range of 2..16 will only go from element 2 to element 5.

If the end is negative the range always goes up, so 2..-2 will go from element 2 to 4, and 2..-16 won't go anywhere because there is no way to go from the second element to one that doesn't exist, while going up. If the start is negative the range always goes down, so -2..1 will go from element 4 to 1, and -16..2 won't go anywhere because there is no way to go from an element that doesn't exist to the second element, while going down.

A missing starting index in a range defaults to 1. This is allowed if the range is the first index expression of the sequence. Similarly, a missing ending index, defaulting to -1 is al? lowed for the last index in the sequence.

Multiple ranges are also possible, separated with a space.

Some examples:

```
echo (seq 10)[1 2 3]
# Prints: 1 2 3
# Limit the command substitution output
echo (seq 10)[2..5]
# Uses elements from 2 to 5
# Output is: 2 3 4 5
echo (seq 10)[7..]
# Prints: 7 8 9 10
# Use overlapping ranges:
echo (seq 10)[2..5 1..3]
# Takes elements from 2 to 5 and then elements from 1 to 3
# Output is: 2 3 4 5 1 2 3
# Reverse output
echo (seq 10)[-1..1]
# Uses elements from the last output line to
# the first one in reverse direction
# Output is: 10 9 8 7 6 5 4 3 2 1
# The command substitution has only one line,
# so these will result in empty output:
echo (echo one)[2..-1]
echo (echo one)[-3..1]
```

The same works when setting or expanding variables:

```
# Reverse path variable
set PATH $PATH[-1..1]
# or
set PATH[-1..1] $PATH
# Use only n last items of the PATH
set n -3
echo $PATH[$n..-1]
```

Variables can be used as indices for expansion of variables, like so:

```
set index 2
set letters a b c d
echo $letters[$index] # returns 'b'
```

However using variables as indices for command substitution is currently not supported, so:

```
echo (seq 5)[$index] # This won't work
set sequence (seq 5) # It needs to be written on two lines like this.
echo $sequence[$index] # returns '2'
```

When using indirect variable expansion with multiple $ ($$name), you have to give all in? dices up to the variable you want to slice:

```
> set -l list 1 2 3 4 5
> set -l name list
> echo $$name[1]
1 2 3 4 5
> echo $$name[1..-1][1..3] # or $$name[1][1..3], since $name only has one element.
1 2 3
```

Home directory expansion

The ~ (tilde) character at the beginning of a parameter, followed by a username, is expanded into the home directory of the specified user. A lone ~, or a ~ followed by a slash, is ex? panded into the home directory of the process owner:

```
ls ~/Music # lists my music directory
echo ~root # prints root's home directory, probably "/root"
```

Combining different expansions

All of the above expansions can be combined. If several expansions result in more than one parameter, all possible combinations are created.

When combining multiple parameter expansions, expansions are performed in the following or? der:

? Command substitutions

? Variable expansions

? Bracket expansion

? Wildcard expansion

Expansions are performed from right to left, nested bracket expansions and command substitu? tions are performed from the inside and out.

Example:

If the current directory contains the files 'foo' and 'bar', the command echo a(ls){1,2,3} will output abar1 abar2 abar3 afoo1 afoo2 afoo3.

Putting it together, here is a quick reference to fish's operators, all of the special sym?

bols it uses:

```
??????????????????????????????????????????????????????????????????????
? Symbol     ? Meaning                  ? Example                 ?
??????????????????????????????????????????????????????????????????????
? $          ? Variable expansion       ? echo $foo               ?
??????????????????????????????????????????????????????????????????????
? $() and () ? Command substitution     ? cat (grep foo bar) or cat ?
?            ?                          ? $(grep foo bar)         ?
??????????????????????????????????????????????????????????????????????
? < and >    ? Redirection, like command ? git shortlog -nse . > au? ?
?            ? > file                   ? thors                   ?
??????????????????????????????????????????????????????????????????????
? |          ? Pipe, connect two or more ? foo | grep bar | grep baz ?
?            ? commands                 ?                         ?
??????????????????????????????????????????????????????????????????????
? ;          ? End of the  command,  in? ? command1; command2      ?
?            ? stead of a newline       ?                         ?
??????????????????????????????????????????????????????????????????????
? &          ? Backgrounding            ? sleep 5m &              ?
??????????????????????????????????????????????????????????????????????
? {}         ? Brace expansion          ? ls {/usr,}/bin          ?
??????????????????????????????????????????????????????????????????????
? && and ||  ? Combiners                ? mkdir foo && cd foo or rm ?
?            ?                          ? foo || exit             ?
??????????????????????????????????????????????????????????????????????
? * and **   ? Wildcards                ? cat   *.fish   or   count ?
?            ?                          ? **.jpg                  ?
??????????????????????????????????????????????????????????????????????
? \\         ? Escaping                 ? echo  foo\nbar  or   echo ?
?            ?                          ? \$foo                   ?
??????????????????????????????????????????????????????????????????????
? " and ""   ? Quoting                  ? rm  "file with spaces" or ?
```

? ? ? echo '$foo' ?

??

? ~ ? Home directory expansion ? ls ~/ or ls ~root/ ?

??

? # ? Comments ? echo Hello # this isn't ?

? ? ? printed ?

??

SHELL VARIABLES

Variables are a way to save data and pass it around. They can be used just by the shell, or they can be "exported", so that a copy of the variable is available to any external command the shell starts. An exported variable is referred to as an "environment variable".

To set a variable value, use the set <> command. A variable name can not be empty and can contain only letters, digits, and underscores. It may begin and end with any of those char? acters.

Example:

To set the variable smurf_color to the value blue, use the command set smurf_color blue.

After a variable has been set, you can use the value of a variable in the shell through variable expansion.

Example:

```
set smurf_color blue
echo Smurfs are usually $smurf_color
set pants_color red
echo Papa smurf, who is $smurf_color, wears $pants_color pants
```

So you set a variable with set, and use it with a $ and the name.

Variable Scope

All variables in fish have a scope. For example they can be global or local to a function or block:

```
# This variable is global, we can use it everywhere.
set --global name Patrick
# This variable is local, it will not be visible in a function we call from here.
set --local place "at the Krusty Krab"
function local
    # This can find $name, but not $place
```

```
    echo Hello this is $name $place
    # This variable is local, it will not be available
    # outside of this function
    set --local instrument mayonnaise
    echo My favorite instrument is $instrument
    # This creates a local $name, and won't touch the global one
    set --local name Spongebob
    echo My best friend is $name
  end
  local
  # Will print:
  # Hello this is Patrick
  # My favorite instrument is mayonnaise
  # My best friend is Spongebob
  echo $name, I am $place and my instrument is $instrument
  # Will print:
  # Patrick, I am at the Krusty Krab and my instrument is
```

There are four kinds of variable scopes in fish: universal, global, function and local vari?
ables.

? Universal variables are shared between all fish sessions a user is running on one com?
puter. They are stored on disk and persist even after reboot.

? Global variables are specific to the current fish session. They can be erased by explic?
itly requesting set -e.

? Function variables are specific to the currently executing function. They are erased ("go
out of scope") when the current function ends. Outside of a function, they don't go out of
scope.

? Local variables are specific to the current block of commands, and automatically erased
when a specific block goes out of scope. A block of commands is a series of commands that
begins with one of the commands for, while , if, function, begin or switch, and ends with
the command end. Outside of a block, this is the same as the function scope.

Variables can be explicitly set to be universal with the -U or --universal switch, global
with -g or --global, function-scoped with -f or --function and local to the current block
with -l or --local. The scoping rules when creating or updating a variable are:

? When a scope is explicitly given, it will be used. If a variable of the same name exists in a different scope, that variable will not be changed.

? When no scope is given, but a variable of that name exists, the variable of the smallest scope will be modified. The scope will not be changed.

? When no scope is given and no variable of that name exists, the variable is created in function scope if inside a function, or global scope if no function is executing.

There can be many variables with the same name, but different scopes. When you use a variable, the smallest scoped variable of that name will be used. If a local variable exists, it will be used instead of the global or universal variable of the same name.

Example:

There are a few possible uses for different scopes.

Typically inside functions you should use local scope:

```
function something
    set -l file /path/to/my/file
    if not test -e "$file"
        set file /path/to/my/otherfile
    end
end
# or
function something
    if test -e /path/to/my/file
        set -f file /path/to/my/file
    else
        set -f file /path/to/my/otherfile
    end
end
```

If you want to set something in config.fish, or set something in a function and have it available for the rest of the session, global scope is a good choice:

```
# Don't shorten the working directory in the prompt
set -g fish_prompt_pwd_dir_length 0
# Set my preferred cursor style:
function setcursors
  set -g fish_cursor_default block
```

```
    set -g fish_cursor_insert line
    set -g fish_cursor_visual underscore
  end
  # Set my language
  set -gx LANG de_DE.UTF-8
```

If you want to set some personal customization, universal variables are nice:

```
  # Typically you'd run this interactively, fish takes care of keeping it.
  set -U fish_color_autosuggestion 555
```

Here is an example of local vs function-scoped variables:

```
  function test-scopes
    begin
      # This is a nice local scope where all variables will die
      set -l pirate 'There be treasure in them thar hills'
      set -f captain Space, the final frontier
      # If no variable of that name was defined, it is function-local.
      set gnu "In the beginning there was nothing, which exploded"
    end
    # This will not output anything, since the pirate was local
    echo $pirate
    # This will output the good Captain's speech
    # since $captain had function-scope.
    echo $captain
    # This will output Sir Terry's wisdom.
    echo $gnu
  end
```

When a function calls another, local variables aren't visible:

```
  function shiver
    set phrase 'Shiver me timbers'
  end
  function avast
    set --local phrase 'Avast, mateys'
    # Calling the shiver function here can not
    # change any variables in the local scope
```

```
    # so phrase remains as we set it here.
    shiver
    echo $phrase
  end
  avast
  # Outputs "Avast, mateys"
```

When in doubt, use function-scoped variables. When you need to make a variable accessible everywhere, make it global. When you need to persistently store configuration, make it uni? versal. When you want to use a variable only in a short block, make it local.

Overriding variables for a single command

If you want to override a variable for a single command, you can use "var=val" statements before the command:

```
  # Call git status on another directory
  # (can also be done via `git -C somerepo status`)
  GIT_DIR=somerepo git status
```

Unlike other shells, fish will first set the variable and then perform other expansions on the line, so:

```
  set foo banana
  foo=gagaga echo $foo
  # prints gagaga, while in other shells it might print "banana"
```

Multiple elements can be given in a brace expansion:

```
  # Call bash with a reasonable default path.
  PATH={/usr,}/{s,}bin bash
```

Or with a glob:

```
  # Run vlc on all mp3 files in the current directory
  # If no file exists it will still be run with no arguments
  mp3s=*.mp3 vlc $mp3s
```

Unlike other shells, this does not inhibit any lookup (aliases or similar). Calling a com? mand after setting a variable override will result in the exact same command being run.

This syntax is supported since fish 3.1.

Universal Variables

Universal variables are variables that are shared between all the user's fish sessions on the computer. Fish stores many of its configuration options as universal variables. This

means that in order to change fish settings, all you have to do is change the variable value once, and it will be automatically updated for all sessions, and preserved across computer reboots and login/logout.

To see universal variables in action, start two fish sessions side by side, and issue the following command in one of them set fish_color_cwd blue. Since fish_color_cwd is a univer? sal variable, the color of the current working directory listing in the prompt will in? stantly change to blue on both terminals.

Universal variables are stored in the file .config/fish/fish_variables. Do not edit this file directly, as your edits may be overwritten. Edit the variables through fish scripts or by using fish interactively instead.

Do not append to universal variables in config.fish, because these variables will then get longer with each new shell instance. Instead, set them once at the command line.

Exporting variables

Variables in fish can be exported, so they will be inherited by any commands started by fish. In particular, this is necessary for variables used to configure external commands like PAGER or GOPATH, but also for variables that contain general system settings like PATH or LANGUAGE. If an external command needs to know a variable, it needs to be exported. Ex? ported variables are also often called "environment variables".

This also applies to fish - when it starts up, it receives environment variables from its parent (usually the terminal). These typically include system configuration like PATH and locale variables.

Variables can be explicitly set to be exported with the -x or --export switch, or not ex? ported with the -u or --unexport switch. The exporting rules when setting a variable are similar to the scoping rules for variables - when an option is passed it is respected, oth? erwise the variable's existing state is used. If no option is passed and the variable didn't exist yet it is not exported.

As a naming convention, exported variables are in uppercase and unexported variables are in lowercase.

For example:

```
set -gx ANDROID_HOME ~/.android # /opt/android-sdk
set -gx CDPATH . ~ (test -e ~/Videos; and echo ~/Videos)
set -gx EDITOR emacs -nw
set -gx GOPATH ~/dev/go
```

```
set -gx GTK2_RC_FILES "$XDG_CONFIG_HOME/gtk-2.0/gtkrc"
set -gx LESSHISTFILE "-"
```

Note: Exporting is not a scope, but an additional state. It typically makes sense to make exported variables global as well, but local-exported variables can be useful if you need something more specific than Overrides. They are copied to functions so the function can't alter them outside, and still available to commands. Global variables are accessible to functions whether they are exported or not.

Lists

Fish can store a list (or an "array" if you wish) of multiple strings inside of a variable:

```
> set mylist first second third
> printf '%s\n' $mylist # prints each element on its own line
first
second
third
```

To access one element of a list, use the index of the element inside of square brackets, like this:

```
echo $PATH[3]
```

List indices start at 1 in fish, not 0 like in other languages. This is because it requires less subtracting of 1 and many common Unix tools like seq work better with it (seq 5 prints 1 to 5, not 0 to 5). An invalid index is silently ignored resulting in no value (not even an empty string, no argument at all).

If you don't use any brackets, all the elements of the list will be passed to the command as separate items. This means you can iterate over a list with for:

```
for i in $PATH
    echo $i is in the path
end
```

This goes over every directory in PATH separately and prints a line saying it is in the path.

To create a variable smurf, containing the items blue and small, write:

```
set smurf blue small
```

It is also possible to set or erase individual elements of a list:

```
# Set smurf to be a list with the elements 'blue' and 'small'
set smurf blue small
```

```
# Change the second element of smurf to 'evil'
set smurf[2] evil
# Erase the first element
set -e smurf[1]
# Output 'evil'
echo $smurf
```

If you specify a negative index when expanding or assigning to a list variable, the index will be taken from the end of the list. For example, the index -1 is the last element of the list:

```
> set fruit apple orange banana
> echo $fruit[-1]
banana
> echo $fruit[-2..-1]
orange
banana
> echo $fruit[-1..1] # reverses the list
banana
orange
apple
```

As you see, you can use a range of indices, see slices for details.

All lists are one-dimensional and can't contain other lists, although it is possible to fake nested lists using dereferencing - see variable expansion.

When a list is exported as an environment variable, it is either space or colon delimited, depending on whether it is a path variable:

```
> set -x smurf blue small
> set -x smurf_PATH forest mushroom
> env | grep smurf
smurf=blue small
smurf_PATH=forest:mushroom
```

Fish automatically creates lists from all environment variables whose name ends in PATH (like PATH, CDPATH or MANPATH), by splitting them on colons. Other variables are not automatically split.

Lists can be inspected with the count <> or the contains <> commands:

```
> count $smurf
2
> contains blue $smurf
# blue was found, so it exits with status 0
# (without printing anything)
> echo $status
0
> contains -i blue $smurf
1
```

A nice thing about lists is that they are passed to commands one element as one argument, so once you've set your list, you can pass it:

```
set -l grep_args -r "my string"
grep $grep_args . # will run the same as `grep -r "my string"` .
```

Unlike other shells, fish does not do "word splitting" - elements in a list stay as they are, even if they contain spaces or tabs.

Argument Handling

An important list is $argv, which contains the arguments to a function or script. For exam? ple:

```
function myfunction
    echo $argv[1]
    echo $argv[3]
end
```

This function takes whatever arguments it gets and prints the first and third:

```
> myfunction first second third
first
third
> myfunction apple cucumber banana
apple
banana
```

That covers the positional arguments, but commandline tools often get various options and flags, and $argv would contain them intermingled with the positional arguments. Typical unix argument handling allows short options (-h, also grouped like in ls -lah), long options (--help) and allows those options to take arguments (--color=auto or --position anywhere or

complete -C"git ") as well as a -- separator to signal the end of options. Handling all of these manually is tricky and error-prone.

A more robust approach to option handling is argparse <>, which checks the defined options and puts them into various variables, leaving only the positional arguments in $argv. Here's a simple example:

```
function mybetterfunction
    # We tell argparse about -h/--help and -s/--second
    # - these are short and long forms of the same option.
    # The "--" here is mandatory,
    # it tells it from where to read the arguments.
    argparse h/help s/second -- $argv
    # exit if argparse failed because
    # it found an option it didn't recognize
    # - it will print an error
    or return
    # If -h or --help is given, we print a little help text and return
    if set -ql _flag_help
        echo "mybetterfunction [-h|--help] [-s|--second] [ARGUMENT ...]"
        return 0
    end
    # If -s or --second is given, we print the second argument,
    # not the first and third.
    # (this is also available as _flag_s because of the short version)
    if set -ql _flag_second
        echo $argv[2]
    else
        echo $argv[1]
        echo $argv[3]
    end
end
```

The options will be removed from $argv, so $argv[2] is the second positional argument now:

```
> mybetterfunction first -s second third
second
```

For more information on argparse, like how to handle option arguments, see the argparse doc?
umentation <>.

PATH variables

Path variables are a special kind of variable used to support colon-delimited path lists in?
cluding PATH, CDPATH, MANPATH, PYTHONPATH, LANGUAGE (for localization <>) etc. All variables that end in "PATH" (case-sensitive) become PATH variables by default.

PATH variables act as normal lists, except they are implicitly joined and split on colons.

```
set MYPATH 1 2 3
echo "$MYPATH"
# 1:2:3
set MYPATH "$MYPATH:4:5"
echo $MYPATH
# 1 2 3 4 5
echo "$MYPATH"
# 1:2:3:4:5
```

Path variables will also be exported in the colon form, so set -x MYPATH 1 2 3 will have ex?
ternal commands see it as 1:2:3.

```
> set -gx MYPATH /bin /usr/bin /sbin
> env | grep MYPATH
MYPATH=/bin:/usr/bin:/sbin
```

This is for compatibility with other tools. Unix doesn't have variables with multiple ele?
ments, the closest thing it has are colon-lists like PATH. For obvious reasons this means no element can contain a :.

Variables can be marked or unmarked as PATH variables via the --path and --unpath options to set.

Special variables

You can change the settings of fish by changing the values of certain variables.

PATH A list of directories in which to search for commands. This is a common unix variable also used by other tools.

CDPATH A list of directories in which the cd <> builtin looks for a new directory.

Locale Variables

Locale variables such as LANG, LC_ALL, LC_MESSAGES, LC_NUMERIC and LC_TIME set the language option for the shell and subprograms. See the section Locale variables for

more information.

Color variables

A number of variable starting with the prefixes fish_color and fish_pager_color. See Variables for changing highlighting colors <#variables-color> for more information.

fish_term24bit

If this is set to 0, fish will not output 24-bit RGB true-color sequences but the nearest color on the 256 color palette (or the 16 color palette, if fish_term256 is 0). See also set_color <>. The default is 1 but for historical reasons, fish de? faults to behaving as if it was 0 on some terminals that are known to not support true-color sequences.

fish_term256

If this is set to 0 and fish_term24bit is 0, translate RGB colors down to the 16 color palette. Also, if this is set to 0, set_color <> commands such as set_color ff0000 red will prefer the named color.

fish_ambiguous_width

controls the computed width of ambiguous-width characters. This should be set to 1 if your terminal renders these characters as single-width (typical), or 2 if dou? ble-width.

fish_emoji_width

controls whether fish assumes emoji render as 2 cells or 1 cell wide. This is neces? sary because the correct value changed from 1 to 2 in Unicode 9, and some terminals may not be aware. Set this if you see graphical glitching related to emoji (or other "special" characters). It should usually be auto-detected.

fish_autosuggestion_enabled

controls if Autosuggestions <#autosuggestions> are enabled. Set it to 0 to disable, anything else to enable. By default they are on.

fish_transient_prompt

If this is set to 1, fish will redraw prompts with a --final-rendering argument be? fore running a commandline, allowing you to change it before pushing it to the scrollback. This enables transient prompts <#transient-prompt>.

fish_handle_reflow

determines whether fish should try to repaint the commandline when the terminal re? sizes. In terminals that reflow text this should be disabled. Set it to 1 to enable,

anything else to disable.

fish_key_bindings

the name of the function that sets up the keyboard shortcuts for the command-line ed? itor <#editor>.

fish_escape_delay_ms

sets how long fish waits for another key after seeing an escape, to distinguish pressing the escape key from the start of an escape sequence. The default is 30ms. Increasing it increases the latency but allows pressing escape instead of alt for alt+character bindings. For more information, see the chapter in the bind documenta? tion <#cmd-bind-escape>.

fish_sequence_key_delay_ms

sets how long fish waits for another key after seeing a key that is part of a longer sequence, to disambiguate. For instance if you had bound \cx\ce to open an editor, fish would wait for this long in milliseconds to see a ctrl-e after a ctrl-x. If the time elapses, it will handle it as a ctrl-x (by default this would copy the current commandline to the clipboard). See also Key sequences <#interactive-key-sequences>.

fish_complete_path

determines where fish looks for completion. When trying to complete for a command, fish looks for files in the directories in this variable.

fish_cursor_selection_mode

controls whether the selection is inclusive or exclusive of the character under the cursor (see Copy and Paste <#killring>).

fish_function_path

determines where fish looks for functions. When fish autoloads a function, it will look for files in these directories.

fish_greeting

the greeting message printed on startup. This is printed by a function of the same name that can be overridden for more complicated changes (see funced <>)

fish_history

the current history session name. If set, all subsequent commands within an interac? tive fish session will be logged to a separate file identified by the value of the variable. If unset, the default session name "fish" is used. If set to an empty string, history is not saved to disk (but is still available within the interactive

session).

fish_trace

if set and not empty, will cause fish to print commands before they execute, similar to set -x in bash. The trace is printed to the path given by the --debug-output option to fish or the FISH_DEBUG_OUTPUT variable. It goes to stderr by default.

FISH_DEBUG

Controls which debug categories fish enables for output, analogous to the --debug option.

FISH_DEBUG_OUTPUT

Specifies a file to direct debug output to.

fish_user_paths

a list of directories that are prepended to PATH. This can be a universal variable.

umask the current file creation mask. The preferred way to change the umask variable is through the umask <> function. An attempt to set umask to an invalid value will always fail.

BROWSER

your preferred web browser. If this variable is set, fish will use the specified browser instead of the system default browser to display the fish documentation.

Fish also provides additional information through the values of certain environment variables. Most of these variables are read-only and their value can't be changed with set.

_ the name of the currently running command (though this is deprecated, and the use of status current-command is preferred).

argv a list of arguments to the shell or function. argv is only defined when inside a function call, or if fish was invoked with a list of arguments, like fish myscript.fish foo bar. This variable can be changed.

argv_opts

argparse <> sets this to the list of successfully parsed options, including option-arguments. This variable can be changed.

CMD_DURATION

the runtime of the last command in milliseconds.

COLUMNS and LINES

the current size of the terminal in height and width. These values are only used by fish if the operating system does not report the size of the terminal. Both variables

must be set in that case otherwise a default of 80x24 will be used. They are updated when the window size changes.

fish_kill_signal

the signal that terminated the last foreground job, or 0 if the job exited normally.

fish_killring

a list of entries in fish's kill ring <#killring> of cut text.

fish_read_limit

how many bytes fish will process with read <> or in a command substitution.

fish_pid

the process ID (PID) of the shell.

history

a list containing the last commands that were entered.

HOME the user's home directory. This variable can be changed.

hostname

the machine's hostname.

IFS the internal field separator that is used for word splitting with the read <> builtin. Setting this to the empty string will also disable line splitting in command substitution. This variable can be changed.

last_pid

the process ID (PID) of the last background process.

PWD the current working directory.

pipestatus

a list of exit statuses of all processes that made up the last executed pipe. See exit status.

SHLVL the level of nesting of shells. Fish increments this in interactive shells, otherwise it only passes it along.

status the exit status of the last foreground job to exit. If the job was terminated through a signal, the exit status will be 128 plus the signal number.

status_generation

the "generation" count of $status. This will be incremented only when the previous command produced an explicit status. (For example, background jobs will not increment this).

USER the current username. This variable can be changed.

EUID the current effective user id, set by fish at startup. This variable can be changed.

version

the version of the currently running fish (also available as FISH_VERSION for back? ward compatibility).

As a convention, an uppercase name is usually used for exported variables, while lowercase variables are not exported. (CMD_DURATION is an exception for historical reasons). This rule is not enforced by fish, but it is good coding practice to use casing to distinguish between exported and unexported variables.

Fish also uses some variables internally, their name usually starting with __fish. These are internal and should not typically be modified directly.

The status variable

Whenever a process exits, an exit status is returned to the program that started it (usually the shell). This exit status is an integer number, which tells the calling application how the execution of the command went. In general, a zero exit status means that the command ex? ecuted without problem, but a non-zero exit status means there was some form of problem.

Fish stores the exit status of the last process in the last job to exit in the status vari? able.

If fish encounters a problem while executing a command, the status variable may also be set to a specific value:

? 0 is generally the exit status of commands if they successfully performed the requested operation.

? 1 is generally the exit status of commands if they failed to perform the requested opera? tion.

? 121 is generally the exit status of commands if they were supplied with invalid arguments.

? 123 means that the command was not executed because the command name contained invalid characters.

? 124 means that the command was not executed because none of the wildcards in the command produced any matches.

? 125 means that while an executable with the specified name was located, the operating sys? tem could not actually execute the command.

? 126 means that while a file with the specified name was located, it was not executable.

? 127 means that no function, builtin or command with the given name could be located.

If a process exits through a signal, the exit status will be 128 plus the number of the sig?

nal.

The status can be negated with not <> (or !), which is useful in a condition. This turns a status of 0 into 1 and any non-zero status into 0.

There is also $pipestatus, which is a list of all status values of processes in a pipe. One difference is that not <> applies to $status, but not $pipestatus, because it loses informa? tion.

For example:

```
not cat file | grep -q fish
echo status is: $status pipestatus is $pipestatus
```

Here $status reflects the status of grep, which returns 0 if it found something, negated with not (so 1 if it found something, 0 otherwise). $pipestatus reflects the status of cat (which returns non-zero for example when it couldn't find the file) and grep, without the negation.

So if both cat and grep succeeded, $status would be 1 because of the not, and $pipestatus would be 0 and 0.

It's possible for the first command to fail while the second succeeds. One common example is when the second program quits early.

For example, if you have a pipeline like:

```
cat file1 file2 | head -n 50
```

This will tell cat to print two files, "file1" and "file2", one after the other, and the head will then only print the first 50 lines. In this case you might often see this constel? lation:

```
> cat file1 file2 | head -n 50
# 50 lines of output
> echo $pipestatus
141 0
```

Here, the "141" signifies that cat was killed by signal number 13 (128 + 13 == 141) - a SIG? PIPE. You can also use fish_kill_signal to see the signal number. This happens because it was still working, and then head closed the pipe, so cat received a signal that it didn't ignore and so it died.

Whether cat here will see a SIGPIPE depends on how long the file is and how much it writes at once, so you might see a pipestatus of "0 0", depending on the implementation. This is a general unix issue and not specific to fish. Some shells feature a "pipefail" feature that

will call a pipeline failed if one of the processes in it failed, and this is a big problem with it.

Locale Variables

The "locale" of a program is its set of language and regional settings. In UNIX, these are made up of several categories. The categories used by fish are:

LANG This is the typical environment variable for specifying a locale. A user may set this variable to express the language they speak, their region, and a character en? coding. The encoding part is ignored, fish always assumes UTF-8. The actual values are specific to their platform, except for special values like C or POSIX.

The value of LANG is used for each category unless the variable for that category was set or LC_ALL is set. So typically you only need to set LANG.

Example values are en_US.UTF-8 for the American English or de_AT.UTF-8 for Austrian German. Your operating system might have a locale command that you can call as lo? cale -a to see a list of defined locales.

LANGUAGE

This is treated like LC_MESSAGES except that it can hold multiple values, which al? lows to specify a priority list of languages for translation. It's a PATH variable, like in GNU gettext <https://www.gnu.org/software/gettext/manual/html_node/The-LANGUAGE-variable.html>.

Language identifiers without a region specified (e.g. zh) result in all available variants of this language being tried in arbitrary order. In this example, we might first look for messages in the zh_CN catalog, followed by zh_TW, or the other way around. This is different from GNU gettext, which uses a "default" variant of the language instead. If you prefer a certain variant, specify it earlier in the list, e.g. zh_TW:zh if your preferred language is zh_TW, and you prefer any other variants of zh over the English default. If zh_TW is the only variant of zh you want, speci? fying zh_TW in the LANGUAGE variable will result in messages which are not available in zh_TW being displayed in English.

See also builtin _ (underscore) <>.

LC_ALL Overrides the LANG and all other LC_* variables. Please use LC_ALL only as a tempo? rary override.

LC_MESSAGES

Determines the language in which messages are displayed, see builtin _ (underscore)

<>.

LC_NUMERIC

Sets the locale for formatting numbers <>.

LC_TIME

Determines how date and time are displayed. Used in the history <#history-show-time> builtin.

BUILTIN COMMANDS

Fish includes a number of commands in the shell directly. We call these "builtins". These include:

? Builtins that manipulate the shell state - cd <> changes directory, set <> sets variables

? Builtins for dealing with data, like string <> for strings and math <> for numbers, count <> for counting lines or arguments, path <> for dealing with path

? status <> for asking about the shell's status

? printf <> and echo <> for creating output

? test <> for checking conditions

? argparse <> for parsing function arguments

? source <> to read a script in the current shell (so changes to variables stay) and eval <> to execute a string as script

? random <> to get random numbers or pick a random element from a list

? read <> for reading from a pipe or the terminal

For a list of all builtins, use builtin -n.

For a list of all builtins, functions and commands shipped with fish, see the list of commands <>. The documentation is also available by using the --help switch.

COMMAND LOOKUP

When fish is told to run something, it goes through multiple steps to find it.

If it contains a /, fish tries to execute the given file, from the current directory on.

If it doesn't contain a /, it could be a function, builtin, or external command, and so fish goes through the full lookup.

In order:

1. It tries to resolve it as a function.

? If the function is already known, it uses that

? If there is a file of the name with a ".fish" suffix in fish_function_path, it loads that. (If there is more than one file only the first is used)

? If the function is now defined it uses that

2. It tries to resolve it as a builtin.

3. It tries to find an executable file in PATH.

? If it finds a file, it tells the kernel to run it.

? If the kernel knows how to run the file (e.g. via a #! line - #!/bin/sh or #!/usr/bin/python), it does it.

? If the kernel reports that it couldn't run it because of a missing interpreter, and the file passes a rudimentary check, fish tells /bin/sh to run it.

If none of these work, fish runs the function fish_command_not_found <> and sets status to 127.

You can use type <> to see how fish resolved something:

```
> type --short --all echo
echo is a builtin
echo is /usr/bin/echo
```

QUERYING FOR USER INPUT

Sometimes, you want to ask the user for input, for instance to confirm something. This can be done with the read <> builtin.

Let's make up an example. This function will glob the files in all the directories it gets as arguments, and if there are more than five <> it will ask the user if it is supposed to show them, but only if it is connected to a terminal:

```
function show_files
    # This will glob on all arguments. Any non-directories will be ignored.
    set -l files $argv/*
    # If there are more than 5 files
    if test (count $files) -gt 5
        # and both stdin (for reading input)
        # and stdout (for writing the prompt)
        # are terminals
        and isatty stdin
        and isatty stdout
        # Keep asking until we get a valid response
        while read --nchars 1 -l response --prompt-str="Are you sure? (y/n)"
            or return 1 # if the read was aborted with ctrl-c/ctrl-d
```

```
            switch $response
                case y Y
                    echo Okay
                    # We break out of the while and go on with the function
                    break
                case n N
                    # We return from the function without printing
                    echo Not showing
                    return 1
                case '*'
                    # We go through the while loop and ask again
                    echo Not valid input
                    continue
            end
        end
    end
    # And now we print the files
    printf '%s\n' $files
end
```

If you run this as show_files /, it will most likely ask you until you press Y/y or N/n. If you run this as show_files / | cat, it will print the files without asking. If you run this as show_files ., it might print something without asking because there are fewer than five files.

SHELL VARIABLE AND FUNCTION NAMES

The names given to variables and functions (so-called "identifiers") have to follow certain rules:

? A variable name cannot be empty. It can contain only letters, digits, and underscores. It may begin and end with any of those characters.

? A function name cannot be empty. It may not begin with a hyphen ("-") and may not contain a slash ("/"). All other characters, including a space, are valid. A function name also can't be the same as a reserved keyword or essential builtin like if or set.

? A bind mode name (e.g., bind -m abc ...) must be a valid variable name.

Other things have other restrictions. For instance what is allowed for file names depends on

your system, but at the very least they cannot contain a "/" (because that is the path sepa?
rator) or NULL byte (because that is how UNIX ends strings).

CONFIGURATION FILES

When fish is started, it reads and runs its configuration files. Where these are depends on build configuration and environment variables.

The main file is ~/.config/fish/config.fish (or more precisely $XDG_CONFIG_HOME/fish/con?
fig.fish).

Configuration files are run in the following order:

? Configuration snippets (named *.fish) in the directories:

  ? $__fish_config_dir/conf.d (by default, ~/.config/fish/conf.d/)

  ? $__fish_sysconf_dir/conf.d (by default, /etc/fish/conf.d/)

  ? Directories for others to ship configuration snippets for their software:

    ? the directories under $__fish_user_data_dir (usually ~/.local/share/fish, controlled by the XDG_DATA_HOME environment variable)

    ? a fish/vendor_conf.d directory in the directories listed in $XDG_DATA_DIRS (default /usr/share/fish/vendor_conf.d and /usr/local/share/fish/vendor_conf.d)

    These directories are also accessible in $__fish_vendor_confdirs. Note that changing that in a running fish won't do anything as by that point the directories have already been read.

  If there are multiple files with the same name in these directories, only the first will be executed. They are executed in order of their filename, sorted (like globs) in a nat?
  ural order (i.e. "01" sorts before "2").

? System-wide configuration files, where administrators can include initialization for all users on the system - similar to /etc/profile for POSIX-style shells - in $__fish_sysconf_dir (usually /etc/fish/config.fish).

? User configuration, usually in ~/.config/fish/config.fish (controlled by the XDG_CON?
  FIG_HOME environment variable, and accessible as $__fish_config_dir).

~/.config/fish/config.fish is sourced after the snippets. This is so you can copy snippets and override some of their behavior.

These files are all executed on the startup of every shell. If you want to run a command only on starting an interactive shell, use the exit status of the command status --is-inter?
active to determine if the shell is interactive. If you want to run a command only when us?
ing a login shell, use status --is-login instead. This will speed up the starting of non-in?

teractive or non-login shells.

If you are developing another program, you may want to add configuration for all users of fish on a system. This is discouraged; if not carefully written, they may have side-effects or slow the startup of the shell. Additionally, users of other shells won't benefit from the fish-specific configuration. However, if they are required, you can install them to the "vendor" configuration directory. As this path may vary from system to system, pkg-config should be used to discover it: pkg-config --variable confdir fish.

For system integration, fish also ships a file called __fish_build_paths.fish. This can be customized during build, for instance because your system requires special paths to be used.

FUTURE FEATURE FLAGS

Feature flags are how fish stages changes that might break scripts. Breaking changes are introduced as opt-in, in a few releases they become opt-out, and eventually the old behavior is removed.

You can see the current list of features via status features:

```
> status features
stderr-nocaret          on  3.0 ^ no longer redirects stderr
qmark-noglob            on  3.0 ? no longer globs
regex-easyesc           on  3.1 string replace -r needs fewer \\'s
ampersand-nobg-in-token on  3.4 & only backgrounds if followed by a separating character
remove-percent-self     off 4.0 %self is no longer expanded (use $fish_pid)
test-require-arg        off 4.0 builtin test requires an argument
mark-prompt             on  4.0 write OSC 133 prompt markers to the terminal
ignore-terminfo         on  4.1 do not look up $TERM in terminfo database
query-term              on  4.1 query the TTY to enable extra functionality
```

Here is what they mean:

? stderr-nocaret was introduced in fish 3.0 and cannot be turned off since fish 3.5. It can still be tested for compatibility, but a no-stderr-nocaret value will be ignored. The flag made ^ an ordinary character instead of denoting an stderr redirection. Use 2> instead.

? qmark-noglob was also introduced in fish 3.0 (and made the default in 4.0). It makes ? an ordinary character instead of a single-character glob. Use a * instead (which will match multiple characters) or find other ways to match files like find.

? regex-easyesc was introduced in 3.1 (and made the default in 3.5). It makes it so the re? placement expression in string replace -r does one fewer round of escaping. Before, to es?

cape a backslash you would have to use string replace -ra '([ab])' '\\\\\\\\$1'. After, just '\\\\$1' is enough. Check your string replace calls if you use this anywhere.

? ampersand-nobg-in-token was introduced in fish 3.4 (and made the default in 3.5). It makes it so a & i no longer interpreted as the backgrounding operator in the middle of a token, so dealing with URLs becomes easier. Either put spaces or a semicolon after the &. This is recommended formatting anyway, and fish_indent will have done it for you already.

? remove-percent-self turns off the special %self expansion. It was introduced in 4.0. To get fish's pid, you can use the fish_pid variable.

? test-require-arg removes builtin test <>'s one-argument form (test "string". It was intro? duced in 4.0. To test if a string is non-empty, use test -n "string". If disabled, any call to test that would change sends a debug message <#debugging-fish> of category "depre? cated-test", so starting fish with fish --debug=deprecated-test can be used to find of? fending calls.

? mark-prompt makes fish report to the terminal the beginning and and of both shell prompts and command output.

? ignore-terminfo disables lookup of $TERM in the terminfo database. Use no-ignore-terminfo to turn it back on.

? query-term allows fish to query the terminal by writing escape sequences and reading the terminal's response. This enables features such as scrolling <# term-compat-cursor-position-report>. If you use an incompatible terminal, you can -- for the time being -- work around it by running (once) set -Ua fish_features no-query-term.

These changes are introduced off by default. They can be enabled on a per session basis:

```
> fish --features qmark-noglob,regex-easyesc
```

or opted into globally for a user:

```
> set -U fish_features regex-easyesc qmark-noglob
```

Features will only be set on startup, so this variable will only take effect if it is uni? versal or exported.

You can also use the version as a group, so 3.0 is equivalent to "stderr-nocaret" and "qmark-noglob". Instead of a version, the special group all enables all features.

Prefixing a feature with no- turns it off instead. E.g. to reenable the ? single-character glob:

```
set -Ua fish_features no-qmark-noglob
```

When defining a new function in fish, it is possible to make it into an event handler, i.e. a function that is automatically run when a specific event takes place. Events that can trigger a handler currently are:

? When a signal is delivered

? When a job exits

? When the value of a variable is updated

? When the prompt is about to be shown

Example:

To specify a signal handler for the WINCH signal, write:

```
function my_signal_handler --on-signal WINCH
    echo Got WINCH signal!
end
```

Fish already has the following named events for the --on-event switch:

? fish_prompt is emitted whenever a new fish prompt is about to be displayed.

? fish_preexec is emitted right before executing an interactive command. The commandline is passed as the first parameter. Not emitted if command is empty.

? fish_posterror is emitted right after executing a command with syntax errors. The command? line is passed as the first parameter.

? fish_postexec is emitted right after executing an interactive command. The commandline is passed as the first parameter. Not emitted if command is empty.

? fish_exit is emitted right before fish exits.

? fish_cancel is emitted when a commandline is cleared.

? fish_focus_in is emitted when fish's terminal gains focus.

? fish_focus_out is emitted when fish's terminal loses focus.

Events can be fired with the emit <> command, and do not have to be defined before. The names just need to match. For example:

```
function handler --on-event imdone
    echo generator is done $argv
end
function generator
    sleep 1
    # The "imdone" is the name of the event
    # the rest is the arguments to pass to the handler
```

```
    emit imdone with $argv
  end
```

If there are multiple handlers for an event, they will all be run, but the order might change between fish releases, so you should not rely on it.

Please note that event handlers only become active when a function is loaded, which means you need to otherwise source <> or execute a function instead of relying on autoloading. One approach is to put it into your configuration file.

For more information on how to define new event handlers, see the documentation for the function <> command.

DEBUGGING FISH SCRIPTS

Fish includes basic built-in debugging facilities that allow you to stop execution of a script at an arbitrary point. When this happens you are presented with an interactive prompt where you can execute any fish command to inspect or change state (there are no debug com? mands as such). For example, you can check or change the value of any variables using printf <> and set <>. As another example, you can run status print-stack-trace <> to see how the current breakpoint was reached. To resume normal execution of the script, type exit <> or ctrl-d.

To start a debug session insert the builtin command <> breakpoint at the point in a function or script where you wish to gain control, then run the function or script. Also, the default action of the TRAP signal is to call this builtin, meaning a running script can be actively debugged by sending it the TRAP signal (kill -s TRAP <PID>). There is limited support for interactively setting or modifying breakpoints from this debug prompt: it is possible to in? sert new breakpoints in (or remove old ones from) other functions by using the funced func? tion to edit the definition of a function, but it is not possible to add or remove a break? point from the function/script currently loaded and being executed.

Another way to debug script issues is to set the fish_trace variable, e.g. fish_trace=1 fish_prompt to see which commands fish executes when running the fish_prompt <> function.

Profiling fish scripts

If you specifically want to debug performance issues, fish can be run with the --profile /path/to/profile.log option to save a profile to the specified path. This profile log in? cludes a breakdown of how long each step in the execution took.

For example:

```
  > fish --profile /tmp/sleep.prof -ic 'sleep 3s'
```

```
> cat /tmp/sleep.prof
Time    Sum     Command
3003419 3003419 > sleep 3s
```

This will show the time for each command itself in the first column, the time for the com?

mand and every subcommand (like any commands inside of a function or command substitutions)

in the second and the command itself in the third, separated with tabs.

The time is given in microseconds.

To see the slowest commands last, sort -nk2 /path/to/logfile is useful.

For profiling fish's startup there is also --profile-startup /path/to/logfile.

See fish <> for more information.

Author

fish-shell developers

Copyright