Λειτουργικά Συστήματα: Εργαστηριακή Άσκηση 2

Εβδομάδα 30 Οκτωβρίου - 3 Νοεμβρίου 2017

fish tutorial

Why fish?

fish is a fully-equipped command line shell (like bash or zsh) that is smart and user-friendly. fish supports powerful features like syntax highlighting, autosuggestions, and tab completions that just work, with nothing to learn or configure.

If you want to make your command line more productive, more useful, and more fun, without learning a bunch of arcane syntax and configuration options, then fish might be just what you're looking for!

Learning fish

This tutorial assumes a basic understanding of command line shells and Unix commands, and that you have a working copy of fish.

If you have a strong understanding of other shells, and want to know what fish does differently, search for the magic phrase *unlike other shells*, which is used to call out important differences.

When you start fish, you should see this:

```
Welcome to fish, the friendly interactive shell
Type help for instructions on how to use fish
you@hostname ~>
```

fish comes with a default prompt that shows your username, hostname, and working directory. You'll see how to change your prompt further down. From now on, we'll pretend your prompt is just a '>' to save space.

Running Commands

fish runs commands like other shells: you type a command, followed by its arguments. Spaces are separators:

```
> echo hello world
hello world
```

You can include a literal space in an argument with a backslash, or by using single or double quotes:

```
> mkdir My\ Files
> cp ~/Some\ File 'My Files'
> ls "My Files"
Some File
```

Commands can be chained with semicolons.

Getting Help

fish has excellent help and man pages. Run help to open help in a web browser, and man to open it in a man page. You can also ask for help with a specific command, for example, help set to open in a web browser, or man set to see it in the terminal.

```
> man set
set - handle shell variables
  Synopsis...
```

Syntax Highlighting

You'll quickly notice that fish performs syntax highlighting as you type. Invalid commands are colored red by default:

```
> /bin/mkd
```

A command may be invalid because it does not exist, or refers to a file that you cannot execute. When the command becomes valid, it is shown in a different color:

```
> /bin/mkdir
```

fish will underline valid file paths as you type them:

```
> cat ~/somefi
```

This tells you that there exists a file that starts with 'somefi', which is useful feedback as you type.

These colors, and many more, can be changed by running fish_config, or by modifying variables directly.

Wildcards

fish supports the familiar wildcard *. To list all JPEG files:

```
> ls *.jpg
lena.jpg
meena.jpg
santa maria.jpg
```

You can include multiple wildcards:

```
> ls l*.p*
lena.png
lesson.pdf
```

Especially powerful is the recursive wildcard ** which searches directories recursively:

```
> ls /var/**.log
/var/log/system.log
/var/run/sntp.log
```

If that directory traversal is taking a long time, you can *Control*-**C** out of it.

Pipes and Redirections

You can pipe between commands with the usual vertical bar:

```
> echo hello world | wc
      1       2      12
```

stdin and stdout can be redirected via the familiar < and >. Unlike other shells, stderr is redirected with a caret ^

```
> grep fish < /etc/shells > ~/output.txt ^ ~/errors.txt
```

Autosuggestions

fish suggests commands as you type, and shows the suggestion to the right of the cursor, in gray. For example:

```
> /bin/hostname
```

It knows about paths and options:

```
> grep --ignore-case
```

And history too. Type a command once, and you can re-summon it by just typing a few letters:

```
> r<\@args{ync} \ ssh . myname@somelonghost.com:/some/long/path/doo/dee/doo/dee/doo}
```

To accept the autosuggestion, hit → or *Control*-**F**. To accept a single word of the autosuggestion, *Alt*-→ (right arrow). If the autosuggestion is not what you want, just ignore it.

Tab Completions

fish comes with a rich set of tab completions, that work "out of the box."

Press **Tab**, and fish will attempt to complete the command, argument, or path:

```
> /pri @key{Tab} → /private/
```

If there's more than one possibility, it will list them:

```
> ~/stuff/s @key{Tab}
~/stuff/script.sh (Executable, 4.8kB) ~/stuff/sources/ (Directory)
```

Hit tab again to cycle through the possibilities.

fish can also complete many commands, like git branches:

```
> git merge pr @key{Tab} → git merge prompt_designer
> git checkout b @key{Tab}
builtin_list_io_merge (Branch) builtin_set_color (Branch) busted_events (Tag)
```

Try hitting tab and see what fish can do!

Variables

Like other shells, a dollar sign performs variable substitution:

```
> echo My home directory is $HOME
My home directory is /home/tutorial
```

Variable substitution also occurs in double quotes, but not single quotes:

```
> echo "My current directory is $PWD"
My current directory is /home/tutorial
> echo 'My current directory is $PWD'
My current directory is $PWD
```

Unlike other shells, `fish` has no dedicated syntax for setting variables. Instead it has an ordinary command: `set`, which takes a variable name, and then its value.

```
> set name 'Mister Noodle'
> echo $name
Mister Noodle
```

(Notice the quotes: without them, `Mister` and `Noodle` would have been separate arguments, and `$name` would have been made into a list of two elements.)

Unlike other shells, variables are not further split after substitution:

```
> mkdir $name
> ls
Mister Noodle
```

In bash, this would have created two directories "Mister" and "Noodle". In `fish`, it created only one: the variable had the value "Mister Noodle", so that is the argument that was passed to `mkdir`, spaces and all. Other shells use the term "arrays", rather than lists.

Exit Status

Unlike other shells, `fish` stores the exit status of the last command in `$status` instead of `$?`.

```
> false
> echo $status
1
```

Zero is considered success, and non-zero is failure.

Exports (Shell Variables)

Unlike other shells, `fish` does not have an export command. Instead, a variable is exported via an option to `set`, either `--export` or just `-x`.

```
> set -x MyVariable SomeValue
> env | grep MyVariable
MyVariablem=SomeValue
```

You can erase a variable with -e or -- erase

```
> set -e MyVariable
> env | grep MyVariable
(no output)
```

Lists

The set command above used quotes to ensure that Mister Noodle was one argument. If it had been two arguments, then name would have been a list of length 2. In fact, all variables in fish are really lists, that can contain any number of values, or none at all.

Some variables, like $PWD, only have one value. By convention, we talk about that variable's value, but we really mean its first (and only) value.

Other variables, like $PATH, really do have multiple values. During variable expansion, the variable expands to become multiple arguments:

```
> echo $PATH
/usr/bin /bin /usr/sbin /sbin /usr/local/bin
```

Note that there are three environment variables that are automatically split on colons to become lists when fish starts running: PATH, CDPATH, MANPATH. Conversely, they are joined on colons when exported to subcommands. All other environment variables (e.g., LD_LIBRARY_PATH) which have similar semantics are treated as simple strings.

Lists cannot contain other lists: there is no recursion. A variable is a list of strings, full stop.

Get the length of a list with count:

```
> count $PATH
5
```

You can append (or prepend) to a list by setting the list to itself, with some additional arguments. Here we append /usr/local/bin to $PATH:

```
> set PATH $PATH /usr/local/bin
```

You can access individual elements with square brackets. Indexing starts at 1 from the beginning, and -1 from the end:

```
> echo $PATH
/usr/bin /bin /usr/sbin /sbin /usr/local/bin
> echo $PATH[1]
/usr/bin
> echo $PATH[-1]
/usr/local/bin
```

You can also access ranges of elements, known as "slices:"

```
> echo $PATH[1..2]
/usr/bin /bin
> echo $PATH[-1..2]
/usr/local/bin /sbin /usr/sbin /bin
```

You can iterate over a list (or a slice) with a for loop:

```
> for val in $PATH
echo "entry: $val"
end
entry: /usr/bin/
entry: /bin
entry: /usr/sbin
entry: /sbin
entry: /usr/local/bin
```

Lists adjacent to other lists or strings are expanded as cartesian products unless quoted (see Variable expansion):

```
> set -l a 1 2 3
> set -l 1 a b c
> echo $a$1
1a 2a 3a 1b 2b 3b 1c 2c 3c
> echo $a" banana"
1 banana 2 banana 3 banana
> echo "$a banana"
1 2 3 banana
```

This is similar to Brace expansion.

Command Substitutions

Command substitutions use the output of one command as an argument to another. Unlike other shells, fish does not use backticks ‘ for command substitutions. Instead, it uses parentheses:

```
> echo In (pwd), running (uname)
In /home/tutorial, running FreeBSD
```

A common idiom is to capture the output of a command in a variable:

```
> set os (uname)
> echo $os
Linux
```

Command substitutions are not expanded within quotes. Instead, you can temporarily close the quotes, add the command substitution, and reopen them, all in the same argument:

```
> touch "testing_"(date +%s)".txt"
> ls *.txt
testing_1360099791.txt
```

Combiners (And, Or, Not)

Unlike other shells, fish does not have special syntax like && or || to combine commands. Instead it has commands and, or, and not.

```
> cp file1.txt file1_bak.txt; and echo "Backup successful"; or echo "Backup failed"
Backup failed
```

Conditionals (If, Else, Switch)

Use `if`, `else if`, and `else` to conditionally execute code, based on the exit status of a command.

```
if grep fish /etc/shells
echo Found fish
else if grep bash /etc/shells
echo Found bash
else
echo Got nothing
end
```

There is also a `switch` command:

```
switch (uname)
case Linux
echo Hi Tux!
case Darwin
echo Hi Hexley!
case FreeBSD NetBSD DragonFly
echo Hi Beastie!
case '*'
echo Hi, stranger!
end
```

Note that `case` does not fall through, and can accept multiple arguments or (quoted) wildcards.

Functions

A `fish` function is a list of commands, which may optionally take arguments. Unlike other shells, arguments are not passed in "numbered variables" like `$1`, but instead in a single list `$argv`. To create a function, use the `function` builtin:

```
> function say_hello
echo Hello $argv
end
> say_hello
Hello
> say_hello everybody!
Hello everybody!
```

Unlike other shells, `fish` does not have aliases or special prompt syntax. Functions take their place.

You can list the names of all functions with the `functions` keyword (note the plural!). `fish` starts out with a number of functions:

```
> functions
        alias, cd, delete-or-exit, dirh, dirs, down-or-search, eval, export, fish_comm
```

You can see the source for any function by passing its name to `functions`:

```
> functions ls
function ls --description 'List contents of directory'
command ls -G $argv
end
```

Loops

While loops:

```
> while true
echo "Loop forever"
end
Loop forever
Loop forever
Loop forever
...
```

For loops can be used to iterate over a list. For example, a list of files:

```
> for file in *.txt
cp $file $file.bak
end
```

Iterating over a list of numbers can be done with seq:

```
> for x in (seq 5)
touch file_$x.txt
end
```

Prompt

Unlike other shells, there is no prompt variable like PS1. To display your prompt, fish executes a function with the name fish_prompt, and its output is used as the prompt.

You can define your own prompt:

```
> function fish_prompt
echo "New Prompt % "
end
New Prompt %
```

Multiple lines are OK. Colors can be set via set_color, passing it named ANSI colors, or hex RGB values:

```
> function fish_prompt
set_color purple
date "+%m/%d/%y"
set_color FF0
echo (pwd) '>'
set_color normal
end
02/06/13
/home/tutorial >
```

You can choose among some sample prompts by running fish_config prompt. fish also supports RPROMPT through fish_right_prompt.

$PATH

$PATH is an environment variable containing the directories in which fish searches for commands. Unlike other shells, $PATH is a list, not a colon-delimited string.

To prepend /usr/local/bin and /usr/sbin to $PATH, you can write:

```
> set PATH /usr/local/bin /usr/sbin $PATH
```

You can do so directly in config.fish, like you might do in other shells with .profile. See this example.

A faster way is to modify the $fish_user_paths universal variable, which is automatically prepended to $PATH. For example, to permanently add /usr/local/bin to your $PATH, you could write:

```
> set -U fish_user_paths /usr/local/bin $fish_user_paths
```

The advantage is that you don't have to go mucking around in files: just run this once at the command line, and it will affect the current session and all future instances too. (Note: you should NOT add this line to config.fish. If you do, the variable will get longer each time you run fish!)

Startup (Where's .bashrc?)

fish starts by executing commands in ~/.config/fish/config.fish. You can create it if it does not exist.

It is possible to directly create functions and variables in config.fish file, using the commands shown above. For example:

```
> cat ~/.config/fish/config.fish

set -x PATH $PATH /sbin/

function ll
ls -lh $argv
end
```

However, it is more common and efficient to use autoloading functions and universal variables.

Autoloading Functions

When fish encounters a command, it attempts to autoload a function for that command, by looking for a file with the name of that command in ~/.config/fish/functions/.

For example, if you wanted to have a function ll, you would add a text file ll.fish to ~/.config/fish/functions:

```
> cat ~/.config/fish/functions/ll.fish
function ll
ls -lh $argv
end
```

This is the preferred way to define your prompt as well:

```
> cat ~/.config/fish/functions/fish_prompt.fish
function fish_prompt
echo (pwd) "> "
end
```

See the documentation for funced and funcsave for ways to create these files automatically.

Universal Variables

A universal variable is a variable whose value is shared across all instances of fish, now and in the future – even after a reboot. You can make a variable universal with set -U:

```
> set -U EDITOR vim
```

Now in another shell:

```
> echo $EDITOR
vim
```

Ready for more?

If you want to learn more about fish, there is lots of detailed documentation, an official mailing list, the IRC channel #fish on irc.oftc.net, and the github page.

Regular Expressions

Μια regular έκφραση είναι ένα πλήθος χαρακτήρων που καθορίζουν ένα πρότυπο αυτών. Τα Regular expressions χρησιμοποιούνται όταν θέλουμε να ψάξουμε μέσα σε γραμμές κειμένου για κείμενο συγκεκριμένης δομής ή συγκεκριμένου προτύπου. Τα περισσότερα UNIX εργαλεία – εντολές λειτουργούν με βάση την ASCII κωδικοποίηση. Τα regular expressions ψάχνουν για πρότυπα ανά γραμμή κειμένου. Πρότυπα που αρχίζουν σε μια γραμμή αλλά τελειώνουν σε κάποια άλλη δεν μπορούν να εντοπιστούν με την χρήση regular expressions.

Είναι εύκολο κανείς να αναζητήσει μια συγκεκριμένη λέξη μέσα σε ένα κείμενο. Οι περισσότεροι εξάλλου editors προσφέρουν αυτήν την επιλογή. Τα Regular expressions είναι περισσότερο δυναμικά και ευέλικτα από μια απλή αναζήτηση λέξης. Μπορούν να μας διευκολύνουν στην αναζήτηση λέξης συγκεκριμένου μεγέθους. Μπορούν να μας βοηθήσουν στην αναζήτηση μια λέξης, για παράδειγμα, με 5 ή περισσότερα φωνήεντα, που να τελειώνει με το γράμμα "s".

Οι ειδικοί χαρακτήρες: ^ and $

Οι περισσότερες επιλογές για αναζήτηση κείμενου στα συστήματα UNIX περιορίζονται από τα ίδια τα όρια των γραμμών του κειμένου. Η αναζήτηση προτύπων κειμένου που μπορεί να υπάρχουν σε πολλαπλές γραμμές είναι κάτι γενικά το δύσκολο. Ο χαρακτήρας ^ οριοθετεί την αρχή του προτύπου ενώ ο χαρακτήρας ".^{A}A θα ταιριάξει όλες τις γραμμές που τελειώνουν με κεφαλαίο Α. Εάν οι ειδικοί χαρακτήρες ^ και $ δε χρησιμοποιηθούν σωστά στα κατάλληλα σημεία του προτύπου τότε παύουν πλέον να λειτουργούν ως ειδικοί χαρακτήρες. Έτσι ο χαρακτήρας ^ λειτουργεί ως ειδικός εάν είναι στην αρχή της έκφρασης. Αντίστοιχα ο χαρακτήρας $ είναι ειδικός

εάν είναι στην τελευταία θέση της έκφρασης. Για παράδειγμα στην έκφραση $1 το σύμβολο $ δεν εκπροσωπεί ειδικό χαρακτήρα. Ομοίως και για το 1ˆ.Παρακάτω παρατίθενται συνοπτικά η χρήση των ειδικών χαρακτήρων.

Πρότυπο	Αντιστοιχία
^A	A στην αρχή του αρχείου
A$	A στο τέλος του αρχείου
A^	A^ οπουδήποτε στην γραμμή
$A	$A οπουδήποτε στην γραμμή
^^	^ στην αρχή του αρχείου
$$	$ στο τέλος του αρχείου

Ο χαρακτήρας "." είναι ένας ειδικός μετά –χαρακτήρας. Χρησιμοποιείται για την δημιουργία προτύπων που περιέχουν ένα συγκεκριμένο χαρακτήρα. Το πρότυπο που θα ταιριάξει μια γραμμή με ένα χαρακτήρα είναι το :

```
^.$
```

Καθορίζοντας μια περιοχή χαρακτήρων με τα σύμβολα

Εάν θέλετε να ορίσετε μια περιοχή με περισσότερους χαρακτήρες ως πρότυπο, μπορείτε να χρησιμοποιήσετε τα σύμβολα [και]. Για παράδειγμα το πρότυπο που θα ταιριάξει οποιαδήποτε γραμμή κειμένου που θα περιέχει έναν αριθμό είναι το:

```
^[0123456789]$
```

Μπορεί επίσης να οριοθετήσει περιοχή χαρακτήρων με τη χρήση του συμβόλου «-», για παράδειγμα:

```
^[0-9]$
```

Τέλος μπορούν να γίνουν συνδυασμοί χαρακτήρων με περιοχές χαρακτήρων. Για παράδειγμα το ακόλουθο πρότυπο θα ταιριάξει οποιοδήποτε χαρακτήρα που είναι γράμμα, αριθμός ή το σύμβολο «_»:

```
[A-Za-z0-9]
```

Το εργαλείο Grep

Το Grep είναι ένα command-line εργαλείο για την αναζήτηση απλού κειμένου μέσα σε γραμμές, που να ταιριάζει με κάποιο δοσμένο regular expression. Το Grep αρχικά δημιουργήθηκε για το UNIX αλλά στη συνέχεια επεκτάθηκε και σε αλλά Unix-like συστήματα. Το όνομα του προέρχεται από την εντολή ed g/re/p (globally search a regular expression and print), η οποία είχε την ίδια ακριβώς λειτουργία: καθολική αναζήτηση με βάση το δοσμένο regular expression και εκτύπωση των γραμμών που ταιριάζουν.

Ένα απλό παράδειγμα της εντολής grep είναι το παρακάτω όπου γίνεται η αναζήτηση στο αρχείο fruitlist.txt για γραμμές που περιέχουν τη λέξη apple:

```
grep apple fruitlist.txt
```

Η αντιστοίχιση θα γίνει όταν η συγκεκριμένη αλληλουχία χαρακτήρων εντοπιστεί, για παράδειγμα οι γραμμές που θα περιέχουν τις λέξεις pineapple ή apples, θα εκτυπωθούν ανεξάρτητα από τα όρια της λέξης. Παρόλο αυτά το όρισμα apple που χρησιμοποιήθηκε στην παραπάνω εντολή είναι case sensitive εξορισμού και επομένως στο παράδειγμα μας δεν πρόκειται να εκτυπωθούν οι γραμμές που θα περιλαμβάνουν την λέξη Apple (με κεφαλαίο Α). Αντιστοιχία με βάση την Case-insensitive λογική θα πραγματοποιηθεί ότι χρησιμοποιηθεί η παράμετρος -i (ignore case) στη σύνταξη της εντολής.

Πολλαπλά ονόματα μπορούν να χρησιμοποιηθούν ως ορίσματα στην εντολή Grep για αντιστοίχιση. Για παράδειγμα όλα τα αρχεία που έχουν επέκταση .txt σε ένα συγκεκριμένο κατάλογο μπορούν να ανιχνευτούν με τη χρήση του συμβόλου * ως τμήμα του ονόματος του αρχείου:

```
grep apple *.txt
```

Τα Regular expressions μπορούν να χρησιμοποιηθούν για περισσότερο σύνθετες αναζητήσεις. Για παράδειγμα η παρακάτω εντολή τυπώνει όλες τις γραμμές του αρχείου fruitlist.txt που ξεκινούν με το γράμμα a, στη συνέχεια έχουν οποιοδήποτε ένα χαρακτήρα και έχουν στη συνέχεια το τμήμα “ple”.

```
grep ^a.ple fruitlist.txt
```