

OpenBSD's pd ksh succinct reference

Wildcards

***** – Zero, or more characters

? – One character

Character from a range of characters:

[aBc] – a, or B, or c

[a-c] – a, or b, or c

[a-z] – Character from a to z

[a-zA-Z0-7] – Character from a to z, from A to Z, and Digit from 0 to 7

[!k-w5] – Any Character not in range from k to w, and not 5

***(abc|xyz)** – "", abc, xyz, abcxyzabc

+(abc|xyz) – abc, xyz, abcxyzabc

?(abc|xyz) – "", abc, xyz

@(abc|xyz) – abc, xyz

!(abc|xyz) – "", not abc, not xyz

Character from a character class: `ls -d ./[[:upper:]][[:digit:]]*/` `ls -d ./[![:digit:]]*/`

[[:print:]] – Non-blank character, or space

[[:graph:]] – Non-blank character

[[:alnum:]] – Digit, lowercase, or uppercase character

[[:alpha:]] – Lowercase, or uppercase character

[[:lower:]] – Lowercase character

[[:upper:]] – Uppercase character

[[:xdigit:]] – Hexadecimal digit [0-9a-fA-F]

[[:digit:]] – Digit

[[:punct:]] – Punctuation character `~!@#\$%^&*()-_+=[]{}<>?/\|";:.,

[[:cntrl:]] – Control character

[[:blank:]] – Tab, or space

[[:space:]] – Blank character (tab, space, newline)

Variable expansion modifiers

typeset variableName=value (local, integer, autoload, functions)

typeset [[+-lprtUux] [-L[n]] [-R[n]] [-Z[n]] [-i[n]] | -f [-tu]] [name [=value] ...]

set -A arrayName -- value01 value02 value03 value04

unset variableName

unset -f functionName

\$variable, \${variable}, \${array[@]}, \${array[*]}, \${array[2]}

\${variable:-value} – Substitutes **variable**'s **value** if **variable** is **unset**, or **null**

\${variable:=value} – Assigns **value** to **variable** if **variable** is **unset**, or **null**

\${variable:+value} – Substitutes **variable**'s **value** if **variable** is **set**, and **not null**

\${variable:?value} – If **variable** is **unset**, or **null**, throws **value** and **exits** with code 1

↑ Colon (:) instructs ksh to treat the **value** of **null** as the **value** of **unset** ↑

\${variable-value} – Substitutes **variable**'s **value** if **variable** is **unset**

\${variable=value} – Assigns **value** to **variable** if **variable** is **unset**

\${variable+value} – Substitutes **variable**'s **value** if **variable** is **set**

\${variable?value} – If **variable** is **unset**, throws **value** and **exits** with code 1

\${#variable} – Substitutes **variable**'s **value** with the **length** of **variable**'s **value**

\${#array[*]} – Substitutes **variable**'s **value** with the **length** of the **array**

\${#array[@]} – Substitutes **variable**'s **value** with the **length** of the **array**

\${variable#wildcard} – **Non-greedy** search for **wildcard** from the **beginning** of **variable**'s **value**, substitutes **variable**'s **value** with the **unmatched** part

\${variable##wildcard} – **Greedy** search for **wildcard** from the **beginning** of **variable**'s **value**, substitutes **variable**'s **value** with the **unmatched** part

\${variable%wildcard} – **Non-greedy** search for **wildcard** from the **end** of **variable**'s **value**, substitutes **variable**'s **value** with the **unmatched** part

\${variable%%wildcard} – **Greedy** search for **wildcard** from the **end** of **variable**'s **value**, substitutes **variable**'s **value** with the **unmatched** part

Subexpressions and tests

- ()** – Executes commands in a subshell with separate environment
- { }** – Groups commands for execution **! ((x)) && { print -u2 'Error'; exit 1; }**
- \$()** – Command substitution in a subshell with separate environment
- \$(())** – Arithmetic expansion, **null** evaluates to **0**
- ((x == 3))** – Arithmetic comparison, same as **let 'x==3'** **==, !=, >, >=, <, <=**
- [\$x == '3']** – String comparison, same as **test \$x == '3'**
- [[\$x == '3']]** – The arguments to **==** and **!=** are **wildcards** **[[\$x == 3*]]** && print 'match'
- [\$x == '3' -o \$foo == 'bar' -a \$w == 'hello']** **[[\$x == '3' || \$foo == 'bar' && \$w == 'hello']]**

File test conditions

- e file** – File exists (same as **-a**)
- r file** – File is readable
- w file** – File is writeable
- x file** – File is executable
- f file** – File exists and is a regular file
- O file** – User owns file
- d file** – File is a directory
- s file** – File exists and is not empty
- c file** – File is character special
- b file** – File is block special
- p file** – File is named pipe
- u file** – File has setuid bit set
- g file** – File has setgid bit set
- k file** – File has sticky bit set
- L file** – File is a symbolic link (same as **-h**)
- G file** – File is in same group as user
- S file** – It is a socket

if [-r file] – Same as **if test -r file**

Miscellaneous test conditions

- z string** – string is zero length
- n string** – string is non-zero length
- ==, !=, >, <** are string conditional tests
- o option** – ksh option is on
- t fd** – **fd** is open on a terminal: **[-t 7]**
- eq** – numbers are equal
- ne** – numbers are not equal
- gt** – left number is greater than right
- ge** – greater than, or equal
- lt** – left number is less than right
- le** – less than, or equal
- nt** – left file is newer than right file
- ot** – left file is older than right file
- ef** – both refer to the same file

if [2 -lt 3] – the above applies only to **test**

Options

- getopts** – Processes **option arguments** (single letter, preceded by hyphen, e.g. **-v**)
- OPTIND** – Variable with value of **1**, incremented after each **getopts** invocation
- OPTARG** – Stores an argument for option, if it's succeeded by colon or number sign (**-a: -i#**)
 - shift** – removes number of arguments (**shift 2** – removes **2** args, **3rd** arg becomes **1st**)
- example* – `./myScript.ksh -vzk -a ./input.txt -i 255 -- arg1 arg2 arg3 arg4 arg5`
- example* – **while getopts ':vza:i#' optionVariable; do**
- leading : – Sets value of optionVariable to **?**, sets value of **OPTARG** to invalid option (**k**)
 - v, z** – Normal options
 - a:** – option requires an argument, which is stored in **OPTARG**
 - if argument is missing, sets value of optionVariable to **:**, value of **OPTARG** to **a**
 - i#** – option accepts only numeric argument, which is stored in **OPTARG**
 - – Marks an end of the option arguments
- arg1 arg2** – Normal arguments
- example* – **case \$optionVariable in**
 - v)** `print -- 'Option -v';;`
 - z)** `print -- 'Option -z';;`
 - a)** `print -- "-a has an argument $OPTARG";;`
 - i)** `print -- "-i has numeric argument $OPTARG";;`
 - :)** `print -u2 -- "$OPTARG is missing an argument"; exit 2;;`
 - \?)** `print -u2 -- "invalid argument $OPTARG"; exit 3;;`
- esac; done**
- shift** `'OPTIND - 1'`
- for** normalArg in "\$@"; do
 - `print -- "$normalArg"`
- done**

File descriptors

- exec 3< f1** – Makes **file1** available for **reading** on **file descriptor** number 3
- exec 4> f2** – Makes **file2** available for **writing** on **file descriptor** number 4 (**overwrite** file)
- exec 4>| f2** – If **set -o noclobber**, then **>|** is required to **overwrite** the file
- exec 4>> f2** – Makes **file2** available for **appending** on **file descriptor** number 4
- exec 5<> f3** – Makes **file3** available for **reading** and **writing** on **file descriptor** number 5
- exec 3<&-** – Closes **file descriptor** number 3
- exec 4>&-** – Closes **file descriptor** number 4
- 2>&1** – Sends error messages to **stdout** (**file descriptor** number 1)
- 2>&6** – Sends error messages to **file descriptor** number 6
- read -u3** – Reads line from **file descriptor** number 3 (**read -u3 -- strVariable**)
If no variable specified, **read** uses variable **REPLY** to store the value
Bourne shell syntax works and has the same result: **read -- strVariable <&3**
- print -u4** – Prints to **file descriptor** number 4 (**print -u4 -- "\$strVariable"**)
- echo 7 >&4** – Prints number 7 to **file descriptor** number 4 (Bourne shell syntax)
- ls -R / >| files.txt 2>&1** – redirects both **stdout** and **stderr** to **files.txt**
- (read -- strVariable; print -- "\$strVariable") < files.txt**
- { print -- 'string'; date; df; } >| sample.txt**
- if ((\$(date +%s) & 1)); then print -- 'A'; else print -- 'B'; fi >> date.txt**
- num=11; case \$num in 2) print -- 'text'; 11) print -- \$num;; *) print -- 'error';; esac >| t.txt**
- for item in 'A' 'B' 'C'; do print -- "\$item"; done >| list.txt**
- i=0; while ((++i <= 3)); do print -- \$i; done > numbers.txt**
- i=0; until ((++i == 3)); do print -- \$i; done >| numbers.txt**

File descriptors:

- 0** – **stdin** (Standard input)
- 1** – **stdout** (Standard output)
- 2** – **stderr** (Standard error)
- 3 .. 9** – **File descriptors** from 3 to 9 are available

Co-processes

dc |& – Runs **dc** as **co-process**

print -p – Prints to **co-process** (**print -p -- '3 5 + p'**)

read -p – Reads from **co-process** (**read -p -- intResult**)

exec 4>&p – Write access to **co-process** on **file descriptor** number **4** (**print -u4 -- '3 5 + p'**)

exec 3<&p – Read access to **co-process** on **file descriptor** number **3** (**read -u3 -- intResult**)

echo 7 >&4 – Prints number 7 to **co-process** (on **file descriptor** number **4**)

functionName arg1 arg2 |& – Runs **function** as a **co-process**

functionName arg1 arg2 & – Runs **function** as a **background job**

To close the **co-process**:

Redirect **co-process**'s input/output to **file descriptors**, if not already

Then close **file descriptors**

exec 4>&p; exec 4>&-

exec 3<&p; exec 3<&-

Indirect variables

read – can be used to create indirect variables

var='hello'; read -- "\${var}"_world – will create **hello_world** variable

eval – can be used to create indirect variables (example: can be used as **eval typeset -i2**)

var='hello'; eval \${var}_world="\\$(date '+%d.%m.%Y')"

same result with double quotes around **\${var}**:

var='hello'; eval "\${var}"_world="\\$(date '+%d.%m.%Y')"

a=1;b=2; eval a\$a=100; eval a\$b=200 – result: **\$a1 == 100 \$a2 == 200**

same result with double quotes:

a=1;b=2; eval a"\$a"=100; eval a"\$b"=200 – result: **\$a1 == 100 \$a2 == 200**

eval c='\$a'\$b – returns **c=\$a2**, which is equals to 200

var=150; if [[\$var -lt \$(eval print -- '\$a'\$b)]]; then print -u2 -- 'diag info'; fi

same result with double quotes around **\$b**:

var=150; if [[\$var -lt \$(eval print -- '\$a'"\$b"')]]; then print -u2 -- 'diag info'; fi

str='abc'; if [["\$str" != "\$(eval print -- '\$a'\$b)"]]; then print -u2 -- 'diag info'; fi

same result with double quotes around **\$b**:

str='abc'; if [["\$str" != "\$(eval print -- '\$a'"\$b"')"]]; then print -u2 -- 'diag info'; fi

eval will return – **print -- \$a2**

double quotes around **\$b** and escaped double quotes around whole variable:

str='abc'; if [["\$str" != "\$(eval print -- "\"\$a\""\$b\"")"]]; then print -u2 -- 'diag info'; fi

eval will return – **print -- "\$a2"**

x=8;y=5; eval x"\$y\$x"=\one two three four five

eval will return – **x58='one two three four five'**

for item in **\$(eval print -- '\$x'"\$y\$x"); do print -- \$item; done**

eval will return – **print -- \$x58**

item4=\$(eval print -- \'"\$x'"\$y\$x"\ | awk '{ print \$4}')

eval will return – **print -- "\$x58"**

eval string=\'"\$x'"\$y\$x"\

eval will return – **string="\$x58"**, which is "one two three four five"

eval item"\$y"=\\$(print -- "\$string" | awk -v "z=\$y" '{ print \$z }')

eval will return – **item5="five"**

eval item"\$y"=\\$(eval print -- \'"\$x'"\$y\$x"\ | awk -v "z=\$y" '{ print \$z }')

subshell **eval** will return – **print -- "\$x58"**

parent shell **eval** will return – **item5="five"**

if **eval** [[\'"\$item'"\$y"\ == 'five']]; then print -u2 -- 'diag info'; fi

eval will return – **[["\$item5" == 'five']]**, which sets \$? to 0

let – can be used to create indirect variables

n=4;m=6; ((n\$m = \$m\$n\$m))

let will return – **n6=646**

n=9;m=1; let "m\$n = \$m\$m\$n"

let will return – **m9=119**

shift – can be used to create indirect variables same way as **let**

j=3;i3=1 shift "i\$j" – will shift the arguments by value of 1

Miscellaneous

: – **no-op** operator. Returns true ($\$? == 0$). Expands variables.

in script **typeset** var1="\$1" var2="\$2" var3="\$3"

: "\${var1:=default 1} \${var2:=default 2} \${var3:=default 3}"

if var1 is **unset** or equals to an **empty string**, it's value will be set to 'default 1', ...

{ ..., ... } – **bracket expansion**

mkdir ./docs{1,2}

mkdir will create directories '**docs1**' and '**docs2**'

str='./docs{1,2}/{sample,test}.{txt,nfo,log}'

for item in \$str; do print -- "\${date}" >| "\$item"; done

creates *sample.txt*, *sample.nfo*, *sample.log*, *test.txt*, *test.nfo*, *test.log* in **both** dirs

set – can be used to assign positional arguments to a script or interactive shell

set -- \$(print -- 'hello' 'world'); var1="\$1"; var2="\$2"

\$# – the variable contains the **number of positional arguments** to a script or shell

in script while ((**\$#**)); do print -- "\$1"; shift; done

loop will print 1st arg, then it will remove 1st arg, 2nd argument will become 1st, ...

"\$(" ")" – Quoting in a subshell

strVariable="\$(ls "\$myPath")"

double quotes in parent shell have no effect on **double quotes** in a subshell

while read – can accept output in different ways

`cmd | while read -- strVariable; do ...; done` – this loop is done in a **subshell**

`cmd |& while read -p -- strVariable; do ...; done`

`while read -- strVariable; do ...; done < inputFile`

`while read -- strVariable; do ...; done < inputFile > outputFile`

`while read -- strVariable; do ...; done < inputFile >&4`

`t=$(cmd); while read -- strVariable; do ...; done <<EOF`

`$t`

`EOF`

`t=$(cmd); while read -- strVariable; do ...; done >| outputFile <<EOF`

`$t`

`EOF`

`function myFunc {`

`while read -- strVariable; do ...; done`

`} < inputFile`

function will read *inputFile* instead of **stdin**

Copyright (c) 2021 Ivan Bityutskiy

Permission to use, copy, modify, and distribute this documentation for any purpose with or without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies.

THE DOCUMENTATION IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS DOCUMENTATION INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS DOCUMENTATION.