Introduction to Unix Shell Scripting

Day 2 - Building Blocks

Thomas Junier, Robin Engler

5, 12, 19 and 26 November 2025

Now that we understand shell operation, we can start discussing the main building blocks of a script.

- Parameters
  - Operations
- Control Statements
  - Conditionals
  - Loops
- Input and Output

Parameters I – setting and using simple parameters

Parameters

- A *parameter* (or *variable*) is just a name for a single value or a collection of values (known as an *array*).
- Scripts *store* values in variables for later use (often a transformation or decision).
- There are multiple ways of setting parameters, and many more ways of retrieving their values, including handy operations.
- We'll see about single-value parameters[1] now; arrays will be discussed later.

[1] I'll just call them "simple" parameters.

Setting: the = operator

Assigns a value to a variable:

```
answer=42                # no need to declare (!= C...)
human='Homo sapiens' # quotes!
date=$(date); shell_msg="I use $SHELL"
species=$human           # no word splitting
answer="nuts!"           # can reassign freely
```

- There is **no whitespace** around the equals sign!
- Variable names (*identifiers*) contain **letters, digits, or underscores**; they cannot not start with a digit.
- Expansions happen, but **word splitting is disabled**[2].

[2] Except for arrays - see below

Getting: ${}

Parameter expansion - retrieves the value of a variable.

```
place=Reykjavík  # set value
echo ${place} # get value: ${...}
echo $place   # short form
```

Unset and Null

- A *null* variable has the **empty string** (`''`) for a value.
- An *unset* variable **has no value** – it does not exist (and it's usually a mistake to refer to its value[3]).
- A variable can be deleted with `unset`.

```
place=Bogotá
place=''     # empty
place=       # same as place=''
unset place # (NOT $place!) - deleted
```

[3]Think of `NULL`, `null`, `nil`, `None`, or `Nothing` in your favorite language.

Beware of the short form!

- Say we have files like `<spc>_18S.dna`, etc. for several species;
- Now we want to remove the files for some species, stored in variable `$spc`.

```
spc=rat; rm $spc_18S.* # WRONG!
spc=rat; rm $spc_18S*  # WORSE!! DON'T DO THIS!
spc=rat; rm ${spc}_18S*  # correct
```

Can you guess what the rule is?

`set -u` guards against unset variables[4].

[4] See Appendix I (day 2 slides).

Type

By default, Bash treats simple values as **strings**[5]. For Bash to treat a value as a number, it must usually be told to do so:

```
a=20; b=30; echo $a+$b   # surprise!
a=20; b=30; echo $((a+b)) # arithmetic
# malformed number -> 0
bond=oo7; echo $((bond))
```

[5]IOW, Bash is a (very) weakly-typed language.

String Operations I

```
genus=Harpagofututor
${#genus}        # length -> 14
${genus:7}       # substring from 7 (0-based)
${genus:0:7}     # substring of length 7 from 0
${genus: -5:4}   # SPACE is required (:-)
a=7; b=14
${genus:a:b}     # arithmetic evaluation
```

String Operations II

```
prog=BLASTN
${prog/N/P}          # substitute 1st match
DNA=cgatgtattcag
RNA=${DNA//t/u}      # idem, all matches (t->u)
img=figure1.jpeg
${img%jp*g}svg       # delete pattern at end
ext=.png
${img/.jp?g/$ext} # expansion happens
```

Many more variants (`man bash`).

Arithmetic Expansion: $(( ))

Shell arithmetic: treats values as *integers*[6] rather than strings:

1. Parameters are expanded ($ not needed)
2. The resulting expression is evaluated numerically (operators on next slide)

```
a=2; b=5
echo $((a * b))
echo $((a < b)) # boolean - 1: true, 0: false
```

[6] As far as possible

Table 1: Main Arithmetic Operators, by decreasing precedence - `()` override.

operator	function
`!`	logical negation
`**`	exponentiation
`*`, `/`, `%`	multiplication, division, remainder
`+`, `-`	addition, subtraction
`<`, `>`, `<=`, `>=`	comparison
`==`, `!=`	logical equality and inequality
`&&`	logical AND
`||`	logical OR
`=`, `+=`, `*=`, ...	assignment

Simple Arithmetic Tests: `(())`[8]

- This is a *command*. It succeeds if the arithmetic expression between `((` and `))` result is **not** zero.
- Combined with `&&` and/or `||`, provides *simple*[7] decision making.

```
((N < 10)) && echo "sample too small"
((N < 10)) && echo "too small" || echo "large enough"
# ==, !=, <= ...
```

[7] There are other, more flexible forms.

[8] Not the same as `$(())` (arithmetic expansion)

Why can 0 signal both success and failure?

Historical aside

- Unix: (many) more ways to fail than to succeed $\rightarrow$ 0 for success, > 0 for various kinds of errors.
- Early shells: crude Boolean and arithmetic expressions (if at all).
- C language: Boolean algebra with 0 for false and nonzero for true.

$\Rightarrow$ When (or "if") arithmetic and Boolean expressions were added to shells, they were strongly influenced by C.[a]

[a] As was the `for (( ; ; ))` loop.

Simple Tests involving String Values: `[[ ]]`

Like `(( ))`, but with strings.

- `[[ ]]` is a command, and succeeds IFF the *conditional expression* within it evaluates to true.
- It can also[9] be combined with `&&`, `||`, etc.

```
[[ $s ]] # $s is non-empty
[[ $s = $t ]] # $s is the same (string) as $t
```

- Spaces matter
- Word splitting is **disabled** between `[[ ]]`, so `"$var"` can be simplified to `$var`

[9] Like any command, in fact

More String Comparison

<, >, etc. also available, but with some differences:

```
[[ Amanda > Pam ]] # Lexical -> false
[[ 2+2 == 4 ]]   # false!
[[ 10 < 2 ]]     # true!
```

`[[...]]` *can* compare numbers[10]

[10] Though perhaps it shouldn't be used for that.

File Properties

They take the form `-c filename`, where the character `c` denotes a file property. For example, to check if a file exists, use `-e`:

```
[[ -e file.txt ]] && echo "exists!"
# Make a dir unless it exists
[[ -d mydir ]] || mkdir mydir
```

Many properties can be tested in this way (see next slide).

Other file property test operators

operator	true if
`-f`	file exists and is a regular file (*e.g.*, not a dir)
`-r`	file is readable
`-w`	file is writable
`-x`	file is executable
`-s`	file exists and has a size greater than 0

There are also a few file *comparison* operators, such as `f1 -nt f2` which is true iff `f1` is newer than `f2`. Obviously, they expect *two* arguments.

Pattern Matching

If unquoted, the right-hand argument of a `==` or `!=` is treated as a pattern (“glob”):

```
[[ abc == ?bc ]]    # true
[[ abc == "?bc" ]] # false (quotes)
gene=lacZ; [[ $gene == lac? ]]
```

Note that the glob pattern isn’t matched against files, but against the **left-hand argument**.

It is also possible to match against regular expressions (see supplementary slides).

Expansion Happens

The above examples used mostly literals for simplicity's sake, but (most) expansions are entirely possible in test commands:

```
[[ -x $HOME ]] # better not be false
[[ $SHELL == /usr/bin/bash ]]
[[ /usr/bin/echo == $(which echo) ]]
a=1; b=1; [[ $a -eq $b ]]
(($(ls | wc -l) > 10)) && echo 'more than ten files'
```

Logical Operations

The expressions can be connected with the logical operators `(...)`, `!`, `&&`, and `||` (decreasing order of precedence).[11]

```
a=Ann; b=Ben; [[ $a == $a && ! ($b == $b) ]]
# Also works with (( ))
a=2; b=3; ((a != b || a == b))
```

[11] The *logical* operators `&&`, `||` and `!` are **not the same** as the *control* operators `&&`, `||` and `!`.

Practice

$\rightarrow$ Exercise 2.1

I/O - Getting data into and out of our script

Input

There are many ways of getting data into our script, including:

- Reusing **standard input**
- **Positional parameters**
- The `read` and `mapfile` functions
- Command substitution: `$(...)`
- Environment variables

Standard Input (stdin)

Commands called by a script inherit its standard input:

```bash
#!/bin/bash
# reuse_stdin.sh
grep Spo0
# at this point, stdin is used up!
```

```
./reuse_stdin.sh < ../data/Spo0A.msa
```

- grep's stdin is the same as `reuse_stdin.sh`'s, namely `../data/Spo0A.msa`.

Practice

$\rightarrow$ **Exercise 2.2** - Reusing standard input.

Arguments and Positional Parameters

A command's arguments are accessed through special parameters called *positional parameters*: **$1**, **$2**, etc., which hold the first, second, etc. arguments, respectively.

```
./show_args.sh -f -y optarg arg1 arg{2..4}
```

- **$0** holds the name of the script (or function).
- **$@** holds all argument values passed to the script (or function).
- **$#** holds the count of the number of argument values passed to the script (or function).

We can now change our script to:

```
#!/bin/bash
# pos_arg.sh
grep Spo0A "$1"
```

```
./pos_arg.sh ../data/Spo0A.msa
```

Sadly, the script now longer works with the < redirection, because then there is no `$1` (do `set -u` to check). [12]

[12] But there is a way to make a script work in both ways.

Reading by lines - `read`

Instead of reading whole files, as we did up to now…

… the `read` builtin[13] takes one or more identifiers, reads one line, word-splits it, and sets the corresponding identifiers:

```
read firstname lastname
echo "Hi, $firstname $lastname!"
```

[13] A builtin command works like a program but is part of the shell

Practice

$\rightarrow$ **Exercise 2.3** - Getting user input from arguments.

Environment Variables: `export`

```bash
#!/bin/bash

printf "NOT_EXPORTED: %s\n" "$NOT_EXPORTED"
printf "    EXPORTED: %s\n" "$EXPORTED"
```

Try the following:

```bash
unset EXPORTED NOT_EXPORTED
NOT_EXPORTED='not exported'
export EXPORTED=exported    # inherited
./use_env.sh
```

Use case: HPC clusters (can't pass parameters).

Output

- `echo`: just outputs its arguments, separated by spaces. By default, doesn't handle \-escapes[14]. Terminates lines with a `\n` unless told not to (`-n`).
- `printf`: formatted printing (see below); `\t` and `\n` work as expected.
- Any programs called will use our script's `stdout` and `stderr`.

[14]Compare `echo 'a\tb'` and `echo -e 'a\tb'`.

`printf`[15] - formatted printing

- Takes a *format string* and zero or more further arguments.
- *Placeholders* in the format string are replaced by the corresponding arguments (in order).
- *Formats* allow control over length, decimal places, padding, etc.

[15] Inherited from the C language, and copied by countless others.

`printf` - Examples

```
pi=3.1415926535; seq=CAGCACACCG;
printf "The value of Pi is about %.2f\n" "$pi"
printf "First 5 residues of seq: %.5s\n" "$seq"
printf "%02d\n" {1..10} # handy for sorting
```

- `%f` treats the argument as a floating-point value.
- `%s` treats the argument as a string.
- `%d` treats the argument as an integer.

→ use `echo` for simple tasks, `printf` for anything else.

Control Structures

Only the simplest scripts run all their code exactly once. Most execute at least *some* instructions more than once, or not at all.

For this, we use *control structures*:

- Loops (repetition) - shortly
- Conditionals (choice) - later
- Groups (sequential flow) - not covered

They start and stop with a specific **keyword** (`if...fi`, `{...}`, `while...done`, etc.)

Test Commands

As we will see shortly, many conditionals and loops involve *test commands*. They are typically one of the following (all already known!):

- a *simple command* - the test succeeds iff the command succeeds (returns 0);
- a *conditional expression* between `[[` and `]]`
- an *arithmetic expression* between `((` and `))`

Loops

Loops are **iterative** control structures: they **repeat** a sequence of commands (called the *body*), typically with minor modifications.

They operate either:

- For a **fixed** number of iterations (`for ... in`). No test command is involved in this form.
- Until some **condition** is met (`while`, `until`, `for ((...))`). This condition is expressed as a test command.

for loop - 1st form

```
for <name> in <words> ; do
  <commands>  # <- body
done
```

Expands `<words>`, and executes `<commands>`, binding `<name>` to each of the resulting values in turn.

Sometimes written with more (or fewer) `;` instead of newlines.

The second form of the `for` loop allows e.g. a dummy variable – see supplementary slides.

`for` loop - Example 1

`for` loops are frequently used with file globs. The following copies a set of `*.fasta` files to `*.fas`:

```
for file in *.fasta ; do
  cp $file ${file/.fasta/.fas}
done
```

`while` loop

In `while` loops, the body is executed **as long as `<test>` succeeds**.

```
while <test> ; do
  <commands> # body
done
```

(The similar `until` loop works in the same way, except that it loops **until** the test succeeds (see Supplementary slides)).

Example: the Collatz conjecture

```bash
#!/usr/bin/env bash

declare -i n=$1
while ((n > 1)); do
    printf "%d\n" "$n"
    if ((0 == n % 2)); then
        ((n /= 2))
    else
        ((n = 3*n +1))
    fi
done
```

Skipping iterations - the `continue` keyword

All loops (`for`, `while`, and `until`) support the use of the `continue` keyword, that makes execution of code **skip the remainder of the current loop**.

Example: print all multiples of 7 between 0 and 100.

```
for x in {0..100}; do
  ((x % 7 != 0)) && continue
  echo $x
done
```

Breaking out of Loops - the `break` keyword

`break` allows an early exit from a loop:

```
# Print the first line that contains some string.
while read line; do
  if [[ $line == *GAATTC* ]]; then
    printf "%s\n" "$line"
    break
  fi
done
```

To break out of *n* nested loops, use `break n`.

Practice

$\longrightarrow$ **Exercise 2.4** - a simple `for` loop.

Conditionals

Conditionals are *branching* control structures. They enable the script to **choose what to do** between two or more possibilities.

The main conditional constructs are:

- `if` - yes-or-no decisions (possibly nested), based on a *test command*
- `case` - multi-way decision, based on *pattern matching*

if

The basic idea:

```
if <test-command> ; then
   <statements> # iff test-command returns 0
fi
```

See `check_user.sh` for an example.

Before we look in detail at test commands, we need to see the full version:

```
if <test-command> ; then
   <statements>
elif <other-test-commands> ; then
   <other-statements>
else
   <default-commands>;
fi
```

- There can be 0 or more `elif` clauses.
- There can be 0 or 1 `else` clause.

Parameters II - Arrays

Arrays: Collections

While a simple parameter stores a single value, an array is a **collection**: it can store more than one value.

Kinds of Arrays

- *indexed* arrays[16] store **lists** of values referred to by a nonnegative integer, e.g. `lines[3]`, `lines[4]` and `lines[10]` could contain some lines from a file;
- *associative* arrays store **key-value pairs**, e.g. `nb_reads['rec_A']` for the number of reads mapping to the *recA* gene, `nb_reads['oxlT']` the number mapping to *oxlT*, etc.

We will cover indexed arrays, see the supplementary slides for the associative variety.

[16] or just "arrays" for short

Indexed Arrays

```
ary=(1 two 'Hey there') # create whole array
ary[3]=foo              # set individual element
ary+=(bar)              # append
unset ary[1]            # delete element
unset ary               # delete array
echo ${ary[0]}          # 0-based
declare -p ary          # inspect array
```

Accessing All Array Elements

Using `*` or `@` as index refers to all array elements, but with subtle differences depending on quoting:[17]

```
names=('Bilbo Baggins' Beorn Gollum)
showa ${names[@]}   # (or *): 4 arguments!
showa "${names[@]}" # 3 arguments
showa "${names[*]}" # 1 argument
IFS=','; echo "${names[*]}"; unset IFS
```

The # operator yields the number of elements

```
echo ${#names[@]} # (or *)
```

[17] And also on whether or not word splitting occurs.

Iterating over an Array

```
for e in "${array[@]}"; do ... ; done
```

Example of Array Usage

Cf. `../src/pascal.sh`

Arrays and Word Splitting

Word splitting is *NOT disabled* when creating arrays:

```
elements='A B "C D"' # no split
array=($elements) # split -> 4 elements!
```

Neither is file globbing:

```
pdf_glob=*.pdf; echo "$pdf_glob"
PDF_S=$pdf_glob ; echo "$PDF_S"
PDF_A=($pdf_glob); declare -p PDF_A
```

Array Caveats

- If no index is given, 0 is assumed:

```
names=(Frodo Lobelia Arwen)
echo $names # = ${names[0]}
            # -> Frodo, NOT whole array!
```

- Arrays can't be assigned as values:

```
# Try to make a copy of `names`
lotr_names=names # WRONG - string assignment
lotr_names=$names # WRONG - see above
lotr_names=(${lotr_names[@]}) # OK
```

Functions

Motivating Example

See `./src/func_motiv*.sh`

Functions

A *function* is like a miniature script that can be called from within another script. Calling a function causes its code to be executed, but does not by itself start a new process.

We write functions in order to:

- Re-use code (DRY principle).
- Improve the clarity of the code.
- Avoid creating new processes (they have a cost: see `noop_test.sh`)

Definition

Functions are defined with one of the following forms:

```
my_func() {
  <commands> # function body
}

function my_func {
 <commands> # function body
}
```

Call

A function is called just like a command. Arguments are passed in the same way:

```
my_func arg1 "$value"
```

Positional parameters (`$1`, `$@`, `shift`, etc.) work in the same way as in scripts.

Local Variables

By default, all variables in Bash are **global**: once set, they can be accessed from anywhere in the script.
Variables defined in a function can[18] be declared as **local**, so that they are only available in the namespace of that function:

```
my_var='a'
my_func() {
  local my_var='b'  # A different variable.
}
```

See `../src/func.sh`.

[18] And usually should

"Returning values" from functions

Shell functions **do not** return values[19]: instead, they return an exit status, again behaving like commands.

To pass data back to the caller: (i) have the function output the value of interest, and (ii) use command substitution over the function call.

```
my_func() { echo "Hi!"; }
result=$(my_func)  # Called just like any command.
```

The exit status is that of the last command in the function body, but it can be overridden with the `return` keyword.

[19] Unlike in pretty much every other language.

Practice

Let's practice using functions to avoid duplication.

$\rightarrow$ **Exercise 2.5** - Introduction to functions.

Conclusion

- We have seen how the shell operates, including tokenizing, parsing, the various forms of expansions, conditionals and arithmetic.
- We have used this knowledge (and more) to write a script that converts Fasta to CSV. It's not efficient, but it's easy to understand.
- You should now be able to start writing your own scripts.

May You Solve Interesting Problems

> *For some people, myself included, the satisfaction of solving a problem is the difference between work and drudgery.* [...] *Besides, once I accomplish my task, I congratulate myself on being clever. I feel like I have done a little bit of magic and spared myself some dull labor.*

Dale Dougherty and Arnold Robbins, *sed & awk* (2010)

Learning shell scripting - Resources

- `man bash` - always handy, if a bit terse.
- `help` - for Bash builtins etc.
- The Bash website and especially the Bash Manual (not the same as `man bash`!).
- The Advanced Bash Scripting Guide.
- The Bash Cheat Sheet.
- The Bash Programming Reference. is another cheatsheet, more specialized towards programming.

- Shellcheck - static analysis tool
- Bash Cookbook
- Bash Idioms

Thanks!

- Thank you all for your attention!
- Special thanks to Robin
- Thanks to the Course Organizers

Supplementary Slides

Arguments don't have to be filenames

- We have used `$1` to specify a data file (by its name).
- The effect is similar to redirecting `stdin` to that file.
- This is very frequent.
- But positional parameters can of course used for countless other purposes.

read (cont'd)

The command fails (returns 1) if it can't read anything.

There can be more than one identifiers:

```
read x y z <<< '1 2 3'
# x=1, y=2, y=3
```

read (final)

If there are more words (after splitting) than there are identifiers, the last identifier gets the extra words:

```
read x y z <<< '1 2 3 4 5'
# x: 1, y: 2, z: 3 4 5
```

If there is only one identifier, it captures the whole line:

```
read line
```

If there are fewer words than identifiers, the extra identifiers are set to an empty string.

for loop - Example 2

Another typical case is with a sequence, often generated by a brace expansion (here {1..10} expands to the numbers from 1 to 10):

```
# Compute the squares of numbers from 1 to 10.
for n in {1..10} ; do echo $((n**2)) ;  done
```

(recall that $((...)) is arithmetic expansion)

`for` loop - 2nd form ("C form")

Another way to write a `for` loop is using the **C syntax** [20].

```
for ((<start-cmd>; <condition>; <iteration-cmd>)); do
    <list>;
done
```

1. Evaluate `<start-cmd>`.
2. Evaluate `<condition>`; if true execute `<list>`, if not exit loop.
3. Evaluate `<iteration-cmd>` and go back to 2.

The evaluations are done in shell arithmetic.

[20] So named because it was copied from the C language

Example: `for` loop, 2nd form

```
# Powers of 2 smaller than 10000.
for ((p=1; p<10000; p=2*p)); do
  echo $p
done
```

Notes:

- Geometric, not arithmetic progression (as in `{1..10}`).
- Only numeric (no globs, etc.).

`until` loop

This works like the `while` loop, except that the body is executed as long as the test *fails*.

```
until <conditional> ; do
 <statements>
done
```

Constants

You can prevent a parameter from changing values

```
declare -r PI=3.14159
PI=4096
# -> bash: PI: readonly variable
```

Readonly variables can't be `unset`.

Defaults for Null and Unset

The “default” operator `:-` substitutes a value if a parameter is unset or null.

```
unset notset
echo $notset
echo ${notset:-default}
null=
echo $null
echo ${null:-default}
```

Variants

- := also *sets* the variable.
- Both have forms (-, =) that only check for unset (not null).

Notes:

- There must be **no whitespace** around `-`, `:-`, etc..
- The right-hand side is *expanded*:

```
unset var; msg="Unset var!"; echo ${var-$msg}
```

Associative Arrays

Associative arrays *must* be `declared` as such:

```
declare -A aar
aar[key1]=val1
declare -A aar=(K1 V1 K2 V2)
echo ${aar[key1]}
```

The behaviour is otherwise very similar to that of indexed arrays.

Note: `*` and `@` work on the *values*, to expand the *keys* use a `!`: `${!aar[*]}`, etc.

Associative Array Caveats

The order in which associative array elements are expanded is **unpredictable**. In particular, there is *no guarantee* that values (or keys) will be listed in the order they were added to the array.

Matching Regular Expressions

The =~ operator matches against POSIX regular expressions[21]

```
[[ abc =~ .bc ]]  # true
[[ abc =~ abc? ]] # true
[[ abc == abc? ]] # false (glob)
```

[21] à la grep, Perl, sed, and so many others... each with its own dialect, all different from globs :-(

No floating-point?

```
echo $((5/2))         # wtf?
echo $((5.0/2.0))     # fails!
```

→ use an external program[22]

```
echo "scale=1; $b/$a" | bc
bc <<< "$scale=1; $b/$a"
result=$(bc <<< "$scale=1; $b/$a") && echo $result
```

[22] In the same way you use `sed`, `awk` etc. when the shell's string functions are too limited

String Conditionals

The following operate on strings

Operator	True if
`s`	s is a non-empty string
`s1 == s2`	s1 and s2 are equal **strings**
`s1 = s2`	idem (note spaces!)
`s1 != s2`	s1 and s2 are different
`s1 < s2`	s1 comes before s2 **alphabetically**
`s1 > s2`	s1 comes after s2 **alphabetically**

Numeric Conditionals

The following operators also operate on strings, but *treat their operands numerically*[23]:

Operator	Meaning (numerically)
`s1 -eq s2`	s1 $=$ s2
`s1 -ne s2`	s1 $\neq$ s2
`s1 -lt s2`	s1 $<$ s2
`s1 -le s2`	s1 $\leq$ s2
`s1 -gt s2`	s1 $>$ s2
`s1 -ge s2`	s1 $\geq$ s2

[23] Using shell arithmetic, more on this shortly

This way things make more sense:

```
[[ 2+2 -eq 4 ]] # true
[[ 10 -lt 2 ]]  # false!
```

Appendix I: Sample Formats

General Feature Format (GFF)

```
##gff-version 3.1.26
##sequence-region ctg123 1 1497228
ctg123 . gene 1000 9000 . + . ID=gene00001;Name=EDEN
ctg123 . mRNA 1050 9000 . + . ID=mRNA00001;Parent=...
ctg123 . mRNA 1050 9000 . + . ID=mRNA00002;Parent=...
ctg123 . mRNA 1300 9000 . + . ID=mRNA00003;Parent=...
ctg123 . exon 1300 1500 . + . ID=exon00001;Parent=...
ctg123 . exon 1050 1500 . + . ID=exon00002;Parent=...
ctg123 . exon 3000 3902 . + . ID=exon00003;Parent=...
```

Variant Call Format (VCF)

```
##fileformat=VCFv4.3
##source=myImputationProgramV3.1
...
#CHROM POS     ID         REF ALT    QUAL FILTER INFO ...
20     14370   rs6054257  G   A      29   PASS   NS=3;...
20     17330   .          T   A      3    q10    NS=3;...
20     1110696 rs6040355  A   G,T    67   PASS   NS=2;...
20     1230237 .          T   .      47   PASS   NS=3;...
20     1234567 microsat1  GTC G,GTCT 50   PASS   NS=3;...
```