MYFIND — Subject

version #

ASSISTANTS C/UNIX 2022 <assistants@tickets.assistants.epita.fr>

Copyright

Contents

*https://intra.assistants.epita.fr

Obligations

*Obligations are **fundamental** rules shared by all subjects. They are non-negotiable and to not apply them means to face sanctions. Therefore, do not hesitate to ask for explanations if you do not understand one of these rules.*

Obligation #0: **Cheating**, as well as sharing source code, tests, test tools or coding-style correction tools is **strictly forbidden** and penalized by not being graded, being flagged as a cheater and reported to the academic staff.

Obligation #1: If you do not submit your work before the deadline, it will not be graded.

Obligation #2: Your submission repository must be **clean**. Except for special cases, which (if any) are **explicitely** mentioned in this document, an *unclean* repository may contain:

- binary files;[1]
- files with inappropriate privileges;
- forbidden files: `*~`, `*.swp`, `*.o`, `*.a`, `*.so`, `*.class`, `*.log`, `*.core`, etc.;
- a file tree that does not follow our specifications.

Obligation #3: All your files must be encoded in ASCII or UTF-8 without BOM.

Obligation #4: When examples demonstrate the use of an output format, you must follow it scrupulously.

Obligation #5: The coding-style needs to be respected at all times.

Obligation #6: **Global variables** are forbidden, unless they are **explicitly** authorized

Obligation #7: Anything that is not **explicitly** allowed is **disallowed**.

Obligation #8: Your code must compile with the flags:

```
-std=c99 -pedantic -Werror -Wall -Wextra
```

Advice

- ▷ Read the *whole* subject.
- ▷ If the slightest project-related problem arise, you can get in touch with the assistants.

  Post to the dedicated **newsgroup** (with the appropriate **tag**) for questions about this document, or send a **ticket** to **<assistants@tickets.assistants.epita.fr>** otherwise.
- ▷ In examples, `42sh$` is our prompt: use it as a reference point.
- ▷ Do **not** wait for the last minute to start your project!

[1] If an executable file is required, please provide its sources **only**. We will compile it ourselves.

1 Getting started

myfind is a simplified version of `find(1)`.

1.1 Instructions

Your goal is to write a program whose behavior follows `find(1)`.

The program's binary name must be `myfind` and must be generated at the root of your repository by your `Makefile` when the rule `all` is used.

In addition to your main assignment, you have to implement a testsuite (in your `./tests/` directory).

Do not underestimate the testsuite, it will allow you to ensure previously implemented features are still working (called regression testing).

Be careful!

You are free to implement your testsuite as you want. However, it **must not** alter your program nor its compilation.

1.2 Goals

With `myfind`, you will learn how to read files' metadata and recursively look through directories using the Unix API.

You will implement simple command line parsing and an AST that will help you evaluate search expressions. Finally, you will have to fork and execute some commands while your program is running.

2 Subject Rules

Files to submit:

- ./Makefile
- ./src/*
- ./tests/*

Makefile: Your makefile should define at least the following targets:

- all: Produce the myfind binary
- check: Run your testsuite with myfind binary
- clean: Delete everything produced by make

Forbidden functions: You can use all the functions of the standard C library except:

- glob(3)
- regexec(3)
- wordexp(3)
- system(3)
- popen(3)
- syscall(2)
- ftw(3)
- nftw(3)
- fts_open(3)
- fts_read(3)
- fts_children(3)
- fts_set(3)
- fts_close(3)

3 Core features

Implementing properly all the core features is barely sufficient to pass. We expect you to implement some additional features to get a decent grade and make up for your errors.

3.1 Error handling

For this project, if you encounter an error while parsing the command line, you must write an *explicit* message on `stderr` and return `1`.

If you encounter an error while processing files and directories (with expressions), you must write an *explicit* message on `stderr` and continue execution. When finished, your program must return `1`.

Otherwise, your program must return `0`.

Error and warning messages must begin with `"myfind: "`. Apart from that, their content will not be tested. Nonetheless, we strongly advise you to write explicit messages as they will help you when you are debugging your code.

Tips

You can use `err(3)` and `warn(3)` to properly print error and warning messages.

3.2 Basic Find

To start off, you will need a basic version of `myfind` that handles `myfind`'s most basic use: printing files. `myfind` must be able to take zero or several arguments and recursively print the filename and/or the contents of the directory given as argument. Arguments are processed from left to right. You must not follow symbolic links.

To begin with, your program must parse command lines of the following format:

```
42sh$ ./myfind [starting-point...]
```

Tips

Look at the `opendir(3)`, `readdir(3)` and `closedir(3)` functions.

Examples:

```
42sh$ ls -RA
foo    myfind qux

./foo:
bar

./qux:
baz
42sh$ ./myfind
.
./qux
```

(continues on next page)

(continued from previous page)

```
./qux/baz
./myfind
./foo
./foo/bar
```

3.3 Expressions

Your will now need to handle zero or more expressions. For now, your program must parse command lines of the following format:

```
42sh$ ./myfind [starting-point...] [expressions]
```

Expressions must be evaluated for every encountered files.

An expression can be one of the following types:

- test
- action
- operator

In order for you to parse and evaluate expressions efficiently, you will need to use an Abstract Syntax Tree to describe an expression as a data structure and evaluate it easily.

The principle of an AST is pretty simple, it is a tree that will contain the different parts of an expression: leaves will contain operands and inner nodes will hold the operators.

There are a lot of ways to build an AST. We strongly suggest you take a look at the Shunting Yard algorithm.

Once your AST is built, you simply have to evaluate it starting from the leaves in order to get the result of you expression.

Tips

We **strongly** advise you to read the man page of `find(1)` in order to clearly understand the purpose of expressions (cf. the `EXPRESSION` part).

3.3.1 Print action

`-print` prints the path of the currently examined file. It always returns `true`.

Examples:

```
42sh$ ./myfind foo
foo
foo/baz
foo/bar
42sh$ ./myfind foo -print
foo
```

(continues on next page)

(continued from previous page)

```
foo/baz
foo/bar
```

3.3.2 Name test

`-name` takes a pattern as parameter and returns `true` if the current filename matches the parameter. You must handle globbing.

Tips

Look at the `fnmatch(3)` function.

Example:

```
42sh$ ./myfind foo
foo
foo/bar
foo/baz
42sh$ ./myfind foo -name bar
foo/bar
42sh$ ./myfind foo -name foo
foo
42sh$ ./myfind foo -name 'ba?'
foo/bar
foo/baz
```

3.3.3 Type test

`-type` takes a parameter and returns `true` if the type of the current file matches the parameter. You must handle the following types:

- `b`: special files in block mode
- `c`: special files in character mode
- `d`: directories
- `f`: regular files
- `l`: symbolic links
- `p`: named pipes (FIFO)
- `s`: sockets

The case where the parameter is invalid should be considered as an error.

Tips

Look at the `readdir(3)` function and the `stat(2)` syscall.

Example:

```
42sh$ ls *
myfind

foo:
bar

qux:
baz
42sh$ ./myfind -type d
.
./foo
./qux
```

Be careful!

Be careful if you decide to use the `d_type` field in `struct dirent`:

You must properly handle the `DT_UNKNOWN` case using `stat(2)`. For more information, please refer to the `readdir(3)` man page.

3.3.4 Or operator

`-o` is an operator placed between two expressions, as follows: `expr1 -o expr2`. If `expr1` is evaluated to `true`, `expr2` will not be executed.

`expr1 -o expr2` is equivalent to one expression.

Example:

```
42sh$ ./myfind foo
foo
foo/baz
foo/bar
42sh$ ./myfind foo -name bar
foo/bar
42sh$ ./myfind foo -name bar -o -name baz
foo/baz
foo/bar
```

3.3.5 And operator

`-a` is an operator placed between two expressions, as follows: `expr1 -a expr2`. If `expr1` is evaluated to `false`, `expr2` will not be executed.

`expr1 -a expr2` is equivalent to one expression.

Example:

```
42sh$ ./myfind foo
foo
foo/baz
foo/bar
```

(continues on next page)

(continued from previous page)

```
42sh$ ./myfind foo -name bar
foo/bar
42sh$ ./myfind foo -name bar -a -name baz
42sh$
```

The default operator between two expressions is `and`, which is why the next expression will not output anything.

```
42sh$ ./myfind foo -name bar -name baz
42sh$
```

Be careful!

Be aware that you **must** handle operator priority.

3.3.6 Newer test

`-newer` takes a file as parameter and returns `true` if the currently-examined file has a last modification date more recent than the file given as argument.

You have to handle nanosecond-level differences: refer to the `st_mtim.tv_nsec` field of `struct stat`.

Tips

Look at the `stat(2)` syscall, especially the `st_mtime` field of `struct stat`.

Be careful!

The behavior of your program will change if you choose to handle options (additional features): if the file is a symbolic link and the `-H` option or the `-L` option is in effect, the modification time of the file it points to is always used.

Be careful!

The AFS does not support nanosecond precision for the time of last modification of a file. Thus, you should test this behavior in another location of your filesystem.

Example:

```
42sh$ mkdir foo
42sh$ touch foo/bar
42sh$ touch foo/baz
42sh$ find foo/* -newer foo/bar
foo/baz
42sh$ find foo/* -newer foo/baz
42sh$
```

4 Additional features

4.1 Myfind options

From now on, your program must be able to parse options before files. It must parse command lines following this format:

```
42sh$ ./myfind [options] [starting-point...] [expressions]
```

You have to implement the following options:

- `-d`: `myfind` should process each directory's content before the directory itself. This option follows the BSD-family find, and not GNU-find, where `-d` is not considered as an option, but as an expression always evaluating to `true`. By default, `myfind` visits directories in pre-order (before their content);
- `-H`: `myfind` does not follow symbolic links, except while processing command line arguments;
- `-L`: `myfind` follows symbolic links;
- `-P`: `myfind` never follows symbolic links. This is the default behavior.

As stated in `find(1)`'s man page: `If more than one of -H, -L and -P is specified, each overrides the others; the last one appearing on the command line takes effect.` Your program must follow this behavior.

Tips

You do not have to handle the case where a filename starts with a `-`.

Examples:

```
42sh$ ls foo
bar baz
42sh$ ./myfind foo
foo
foo/bar
foo/baz
42sh$ ./myfind -d foo
foo/bar
foo/baz
foo
```

```
42sh$ ls
foo myfind
42sh$ ls foo
bar baz
42sh$ ln -s foo qux
42sh$ ./myfind qux
qux
42sh$ ./myfind -H qux
qux
qux/bar
qux/baz
```

```
42sh$ ls
qux foo myfind
42sh$ file qux
qux: symbolic link to foo
42sh$ ./myfind .
.
./myfind
./foo
./foo/bar
./foo/baz
./qux
42sh$ ./myfind -L .
.
./myfind
./foo
./foo/bar
./foo/baz
./qux
./qux/bar
./qux/baz
```

4.2 Operators

In this additional part for expressions, you must implement the `!` and `()` operators. The expression `-name toto -o -name tata -a -type f` is equivalent to `-name toto -o ( -name tata -a -type f )`.

4.2.1 Not operator

`!` is an operator than can be placed before an expression, as follows: `! expr`. It returns `true` if expr is false, and vice-versa.

`! expr` is equivalent to one expression.

Examples:

```
42sh$ ./myfind foo
foo
foo/baz
foo/bar
42sh$ ./myfind -name bar
./foo/bar
42sh$ ./myfind '!' -name bar
.
./myfind
./foo
./foo/baz
```

4.2.2 Parentheses operator

You must handle parentheses. A pair of parentheses is an expression that returns `true` if the wrapped expressions returned `true`, `false` otherwise.

`( expr )` is equivalent to one expression.

Examples:

```
42sh$ ./myfind foo
foo
foo/baz
foo/bar
42sh$ ./myfind \( -name bar -o -name baz \)
./foo/baz
./foo/bar
42sh$ ./myfind \! \( -name bar -o -name baz \)
./foo
```

4.3 Exec-family actions

`myfind` must be able to execute commands on matched files, using the following actions:

- `-exec`
- `-execdir`
- `-exec ... +`

4.3.1 Exec

The `-exec` action executes the command passed by argument, delimited by a semicolon. Every `{}` string encountered in the command must be replaced by the current filename. It returns `true` if the command returned 0, `false` otherwise.

Examples:

```
42sh$ pwd
/tmp
42sh$ ./myfind foo
foo
foo/bar
foo/baz
42sh$ ./myfind foo -exec pwd \; -exec echo -- {} -- \;
/tmp
-- foo --
/tmp
-- foo/baz --
/tmp
-- foo/bar --
42sh$ ./myfind foo -exec md5sum {} \; -exec echo ok \;
md5sum: foo: Is a directory
```

(continues on next page)

(continued from previous page)

```
d41d8cd98f00b204e9800998ecf8427e  foo/baz
ok
d41d8cd98f00b204e9800998ecf8427e  foo/bar
ok
```

4.3.2 Execdir

`-execdir` behavior is similar to `-exec`, except it executes the command in the current file's directory. `{}` placeholders will be replaced by the current file name, preceded by `./` (relative file path).

Examples:

```
42sh$ ./myfind foo -execdir pwd \; -execdir echo -- {} -- \;
/tmp
-- ./foo --
/tmp/foo
-- ./baz --
/tmp/foo
-- ./bar --
```

4.3.3 Exec +

The `-exec command {} +` action is similar to the `-exec` action. The main difference is that the command line is built by appending the matching filenames at the end. Thus, instead of invoking the command once for each matched file, it is executed once for many of them at the same time.

The command line must end with `{}`. Any other instance of this placeholder in the command line will result in an error.

Examples:

```
42sh$ pwd
/tmp
42sh$ ./myfind foo
foo
foo/baz
foo/bar
42sh$ ./myfind foo -exec echo {} \;
foo
foo/baz
foo/bar
42sh$ ./myfind foo -exec echo {} \+
foo foo/baz foo/bar
42sh$ ./myfind foo -exec echo {} a \+
myfind: missing argument to `-exec'
42sh$ echo $?
1
42sh$ ./myfind foo -exec echo {} {} \+
myfind: only one instance of {} is supported with -exec ... +
42sh$ echo $?
1
```

4.3.4 Resource Leaks

You shall close every file descriptor that you opened before executing the command (which means everything but `STDIN_FILENO`, `STDOUT_FILENO` and `STDERR_FILENO` from `unistd.h`).

You shall not leave zombie processes after `myfind` exits (c.f. `waitpid(2)`).

4.4 Delete action

`-delete` deletes files and returns `true` if it succeeded. If the deletion fails, an error message is displayed.

Use of `-delete` implies use of `-d`. Meaning that your program must behave as if it was called with the `-d` option.

Examples:

```
42sh$ ./myfind
.
./myfind
./foo
./foo/baz
./foo/bar
42sh$ ./myfind foo -name bar -delete -delete
myfind: cannot delete ‘foo/bar’: No such file or directory
42sh$ ./myfind
.
./myfind
./foo
./foo/baz
42sh$ ./myfind foo -delete
42sh$ ./myfind
.
./myfind
```

4.5 Perm test

`-perm` takes a mode in octal as parameter and returns `true` if the current file's permission bits match exactly the mode.

`-perm -` takes a mode in octal as parameter and returns `true` if all the permission bits in mode are set for the file.

`-perm /` takes a mode in octal as parameter and return `true` if any of the current file's permission bits is set in mode.

Examples:

```
42sh$ ls -l foo
total 0
-rw-r--r-- 1 login_x login_x 0 oct.  3 20:42 bar
-rw-r--r-- 1 login_x login_x 0 oct.  3 20:42 baz
```

(continues on next page)

(continued from previous page)

```
42sh$ ./myfind -perm 644
./foo/baz
./foo/bar
42sh$ chmod 123 foo/baz
42sh$ ./myfind -perm 644
./foo/bar
```

```
42sh$ ls -l foo
total 0
-rw-r--r-- 1 login_x login_x 0 oct.  3 20:42 bar
-rw-r--r-- 1 login_x login_x 0 oct.  3 20:42 baz
42sh$ ./myfind -perm -640
.
./myfind
./foo
./foo/baz
./foo/bar
42sh$ chmod 123 foo/baz
42sh$ ./myfind -perm -102
./foo/baz
```

```
42sh$ ls -l foo
total 0
-rw-r--r-- 1 login_x login_x 0 oct.  3 20:42 bar
-rw-r--r-- 1 login_x login_x 0 oct.  3 20:42 baz
42sh$ ./myfind -perm 644
./foo/baz
./foo/bar
42sh$ chmod o-r foo/baz
42sh$ ./myfind foo -perm /004
foo
foo/bar
```

4.6 User and Groups tests

`-user` takes a `username` and returns `true` if the file is owned by the user `username`.

`-group` takes a `groupname` and returns `true` if the file belongs to the group `groupname`.

Examples:

```
42sh$ ls -l foo
-rw-r--r-- 1 login_x login_x 0 oct.  3 20:42 bar
-rw-r--r-- 1 login_x test    0 oct.  3 20:42 baz
-rw-r--r-- 1 toto    test    0 oct.  3 20:42 qux
42sh$ ./myfind foo/* -user login_x
foo/baz
foo/bar
42sh$ ./myfind foo/* -group test
foo/qux
foo/baz
```

It is my job to make sure you do yours.