The AWK book's 60-line version of Make


benhoyt.com/writings/awk-make/

In the wonderful book *The AWK Programming Language* by Aho, Weinberger, and Kernighan, there are a few pages at the end of chapter 7 that present a simplified version of the Make utility – written in a single page of AWK code.

Before we look at that, I want to mention that the second edition of the AWK book is coming out next month. Brian Kernighan's done a great job of updating it, most notably with a new chapter on exploratory data analysis, and adding proper CSV support to AWK to enable this. I was honoured to be asked to review a draft of the second edition.

AWK still shines for exploring data in 2023, especially with the new `--csv` option. CSV mode has also been added to Gawk (GNU AWK), the most widely-installed version of AWK. My own GoAWK implementation has had proper CSV support for some time, and I've added the `--csv` option to match the others.

The second edition of the book still includes the Make program, though it's been made more readable with the addition of some "spacing and bracing" – this took it from 50 lines to 62 lines.

This article presents the Make program, to show how AWK is not just great for one-liners, but can be used as a scripting language too – though whether you *should* or not is another question.

I'm then going to compare what the same program would look like in Python, and briefly discuss when you'd choose AWK or Python for this kind of thing.

It should go without saying, but I intend this purely as a learning exercise (for me and my readers), not a program I'd recommend you use to build your projects!

Original AWK version

The second edition of the book introduces the Make program as follows. (For what it's worth, I find the term "target" confusing here – I think "source" or "dependency" would fit better.)

This section develops a rudimentary updating program, patterned after the Unix `make` command, that is based on the depth-first search technique of the previous section.

To use the updater, one must explicitly describe what the components of the system are, how they depend upon one another, and what commands are needed to construct them. We'll assume these dependencies and commands are stored in a file, called a `makefile`, that contains a sequence of rules of the form

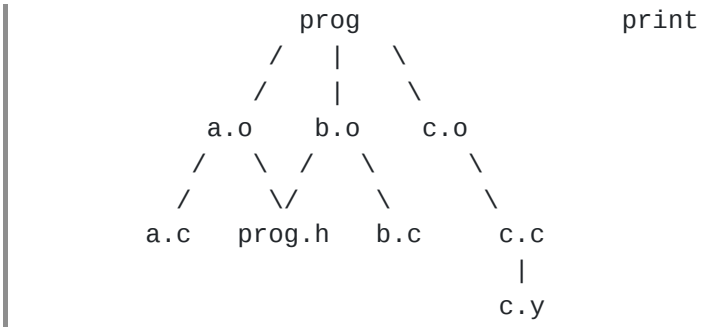
```
name:   t1 t2 ... tn
        commands
```

The first line of a rule is a dependency relation that states that the program or file *name* depends on the targets *t1*, *t2*, …, *tn* where each *ti* is a filename or another *name*. Following each dependency relation may be one or more lines of *commands* that list the commands necessary to generate *name*. Here is an example of a `makefile` for a small program with two C files called `a.c` and `b.c`, and a `yacc` grammar file `c.y`, a typical program-development application.

```
prog:   a.o b.o c.o
        gcc a.o b.o c.o -ly -o prog
a.o:    prog.h a.c
        gcc -c prog.h a.c
b.o:    prog.h b.c
        gcc -c prog.h b.c
c.o:    c.c
        gcc -c c.c
c.c:    c.y
        yacc c.y
        mv y.tab.c c.c
print:
        pr prog.h a.c b.c c.y
```

The first line states that `prog` depends on the target files `a.o`, `b.o`, and `c.o`. The second line says that `prog` is generated by using the C compiler command `gcc` to link `a.o`, `b.o`, `c.o`, and a `yacc` library `y` into the file `prog`. The next rule (third line) states that `a.o` depends on the targets `prog.h` and `a.c` and is created by compiling these targets; `b.o` is the same. The file `c.o` depends on `c.c`, which in turn depends on `c.y`, which has to be processed by the `yacc` parser generator. Finally, the name `print` does not depend on any target; by convention, for targetless names `make` will always perform the associated action, in this case printing all the source files with the command `pr`.

The dependency relations in the `makefile` can be represented by a graph in which there is an edge from node *x* to node *y* whenever there is a dependency rule with *x* on the left side and *y* one of the targets on the right. For a rule with no targets, a successorless node with the name on the left is created. For the `makefile` above, we have the following dependency graph:



It's a highly-simplified version of Make, of course, but still has the core concepts of outputs, dependencies, and build commands.

Before we look at how it works, I've included the full source code below, as it appears in the second edition of the AWK book. Click on the bold text to expand it, or skip down to "How it works" to see the code explained in detail.

▶ **The AWK book's Make program (full source code).**

How it works

There's an explanation of how the program works in the book, but I'll explain it in my own words here, focussing on the aspects I find interesting.

The `BEGIN` block is the main entry point for a program like this. Unlike most AWK programs which implicitly read lines from standard input, this one uses an explicit loop with `getline` to read the `makefile`:

```
BEGIN {
    while (getline <"makefile" > 0) {
        if ($0 ~ /^[A-Za-z]/) {  #  $1: $2 $3 ...
            sub(/:/, "")
            if (++names[nm = $1] > 1)
                error(nm " is multiply defined")
            for (i = 2; i <= NF; i++) # remember targets
                slist[nm, ++scnt[nm]] = $i
        } else if ($0 ~ /^\t/) {      # remember cmd for
            cmd[nm] = cmd[nm] $0 "\n" #   current name
        } else if (NF > 0) {
            error("illegal line in makefile: " $0)
        }
    }
    ...
}
```

The `getline <filename` is a redirect clause that opens `makefile` (the first time) and reads it line-by-line until the end. If the line (`$0`) starts with a letter (`/^[A-Za-z]/`), it's considered a `name: targets` rule.

The `sub(/:/, "")` call removes the colon from the current line (the `$0` is implicit in the two-argument form of `sub`).

We then ensure that this rule hasn't already been defined by checking the `names` array. An AWK array is actually an *associative array*, an old-school term for a key-value map.

The inner `for` loop adds each target (or dependency) to the `slist` / `scnt` data structure. This is really a map of lists, but it's flattened to work around the fact that AWK doesn't support nested collections. The body of the loop is very terse:

```
for (i = 2; i <= NF; i++)
    slist[nm, ++scnt[nm]] = $i
```

This loops through each dependency: every field `$i` from field 2 to `NF` (the number of fields in the line).

For each dependency, it increments `scnt[nm]`, the count of sources for the current rule (`nm`). Then, store the dependency `$i` in `slist`, indexed by the multi-key name and count. AWK simulates multi-dimensional or multi-key arrays by creating a concatenated key where each key is separated by the `SUBSEP` separator (which defaults to `"\x1c"`).

After the loop, in the `prog` example we'd end up with `slist` and `scnt` looking like this:

```
slist
    a.o,1:  prog.h
    a.o,2:  a.c
    b.o,1:  prog.h
    b.o,2:  b.c
    c.c,1:  c.y
    c.o,1:  c.c
    prog,1: a.o
    prog,2: b.o
    prog,3: c.o

scnt
    a.o:  2
    b.o:  2
    c.c:  1
    c.o:  1
    prog: 3
```

Coming back up, if the line starts with a tab, it's a command, so we append it to the name's command string:

```
cmd[nm] = cmd[nm] $0 "\n"
```

Otherwise, if the line is not a blank line (`NF > 0`), it's a `makefile` error.

Finally, after reading the `makefile` in the `while` loop, we uses `ages()` to compute the ages of all files in the current directory, and then call `update(ARGV[1])` to update the rule passed on the command line:

```
BEGIN {
    ...
    ages()      # compute initial ages

    if (ARGV[1] in names) {
        if (update(ARGV[1]) == 0)
            print ARGV[1] " is up to date"
    } else {
        error(ARGV[1] " is not in makefile")
    }
}
```

The `ages` function is where things start to get interesting:

```
function ages(      f,n,t) {
    for (t = 1; ("ls -t" | getline f) > 0; t++)
        age[f] = t          # all existing files get an age
    close("ls -t")

    for (n in names)
        if (!(n in age))    # if n has not been created
            age[n] = 9999   # make n really old
}
```

The parameter names `f`, `n`, and `t` are prefixed with a bunch of spaces to show they're actually local variables, and not expected as arguments. This is an AWK quirk (which Kernighan regrets): the only way to define local variables is as function parameters, and if a function is called with fewer arguments than it has parameters, the extras take on the default value (0 for numbers, `""` for strings). So you'll see these extra spaces a lot in AWK function definitions.

The next thing is quite neat: AWK supports shell-like `|` syntax to pipe the output of a program, one `getline` at a time, to a variable (in this case `f`). The `ls -t` command lists files in the current directory ordered by modification time, newest first.

After the loop that's assigned each file's age to `age[f]`, we call `close` to close the `ls -t` pipe and avoid too many open file handles.

Finally, we loop through the rule names and assign an arbitrary large number to `age[n]` to pretend that files that haven't been created are really old and need to be updated.

Next is the recursive `update` function, where the meat of the algorithm lives:

```
function update(n,   changed,i,s) {
    if (!(n in age))
        error(n " does not exist")
    if (!(n in names))
        return 0
    changed = 0
    visited[n] = 1
    for (i = 1; i <= scnt[n]; i++) {
        if (visited[s = slist[n, i]] == 0)
            update(s)
        else if (visited[s] == 1)
            error(s " and " n " are circularly defined")
        if (age[s] <= age[n])
            changed++
    }
    visited[n] = 2
    if (changed || scnt[n] == 0) {
        printf("%s", cmd[n])
        system(cmd[n])  # execute cmd associated with n
        ages()          # recompute all ages
        age[n] = 0      # make n very new
        return 1
    }
    return 0
}
```

Once again you'll note the parameter list: `n` is an expected argument (the name to update), and `changed,i,s` are the locals.

After initial checks, we loop through the list of dependencies by iterating from `slist[n, 1]` to `slist[n, scnt[n]]`. If we haven't visited this dependency yet, we perform a depth-first traversal of the dependency graph by recursively calling `update` to see if we need to update that dependency first:

```
if (visited[s = slist[n, i]] == 0)
    update(s)
```

The recursion is terminated by the `if (!(n in names)) return 0` block near the top. We stop when the file being updated isn't in the list of rule names – which is a leaf node in the dependency graph.

The block `if (age[s] <= age[n]) changed++` increments the `changed` count if any dependency is newer than the age of the current file being updated.

After the traversal loop, if any of the dependencies or sub-dependencies had changed, we run the associated command using `system()`, recompute the ages of all files, and `return 1` to the caller to indicate we did make an update.

The `scnt[n] == 0` clause handles the case where the rule being updated doesn't have any dependencies specified, like the `print` rule in the example. In that case, always re-run its command.

And there you have it! A minimalist Make in one page of AWK.

Python version

For interest, I ported the book's AWK Make to Python, and have included it below. Once again, click the bold text to expand the program.

▶ **My Python port of the Make program (full source code).**

It's very similar in structure to the original AWK version, though I made two simplifications which I think make it somewhat easier to understand:

1. Simpler data structures to avoid the `slist` / `scnt` quirkiness – in Python we can just use a dictionary of lists. (See diff.)
2. Determine ages more directly using `os.stat()` to fetch file modification times (mtimes), rather than using the `ls -t` trick. This also removes the need for the `age` map and the `ages` function. (See diff.)

I didn't plan for this, but even if you include the `import` line and the `if __name__ == '__main__'` dance, it's 58 lines of code – basically the same length as the AWK program.

When making the Python version, I realized we could simplify the AWK version in a similar way:

1. It's conceptually simpler to store the `slist` directly as an AWK array: a key-value map where the key is the rule name and the value is the list of dependencies as a space-separated string (just like in the `makefile`). We can use `split` as needed to turn the dependencies string into a list (an array from 1 to the number of dependencies). This avoids the need for `scnt` and `names` altogether. (See diff.)
2. Similar to the Python version, we can get the mtime directly by shelling out to `stat`, instead of listing all files in age order with `ls -t`. I've used `stat --format %y` to do this. I believe this is a GNU extension, so it's not as portable as `ls -t`, but it's simpler and avoids the need for recomputing the `age` array. (See diff.)

Update: Volodymyr Gubarkov pointed out that the `stat` version "adds multiple external process invocations", and he's quite right. It might be more direct, but it is significantly slower.

For what it's worth, the modified version is four lines shorter than the original. I think the simpler `slist` is clearer, and I like the more direct approach to fetching mtimes, though I realize the lack of portability of `stat --format` is a downside (macOS's `stat` looks quite different).

Conclusion

The AWK Make program is a neat little piece of code that shows how useful a language AWK is, even for medium-sized scripts.

However, Python is definitely a nicer language for this kind of thing: it has much richer data types, better tools like `os.stat`, and local variables without quirky syntax.

I consider AWK amazing, but I think it should remain where it excels: for exploratory data analysis and for one-liner data extraction scripts.

As the author of GoAWK, which has had native CSV support for a while, I'm especially pleased to see both Kernighan's "one true AWK" and Gawk gain proper CSV support in the form of the `--csv` option. Kernighan's AWK updates will be merged soon, and Gawk will include this feature in version 5.3.0, which is coming out soon.

You can also view my awkmake repo on GitHub, which contains the full source for both the AWK book's Make program and my Python version, as well as a runnable example project based on the example in the AWK book.