USING `AWK` IN TEXSHOP

MICHAEL SHARPE

1. Overview

BabyAwk is a TEXShop Macro Menu item that provides access to some features of the Unix utility `[g]awk`, enabling a subset of `[g]awk`'s features to be applied to the selected text. This is useful when you have a text block that is sufficiently structured to permit separation into "columns" which can be rearranged and manipulated with great flexibility. In the simplest instances, `[g]awk` reads one line of text at a time, though what a line means is defined by the `[g]awk` variable `RS`, the record separator, and columns are normally separated by white space, though this is in fact controlled by the `[g]awk` variable `FS`, the field separator. After parsing a line, the fields are set to variables with names `$1`, `$2,...` (the original line is set to `$0`) and, based on whether the line matches the specified search pattern (eg, `/tex/` matches only lines containing `tex`) or the specified condition (eg, `$2>3`), it carries out `Action`, which in most cases involves a `print` ot `printf` statement.

Note that nothing is lost when calling `BabyAwk`—the original selection is commented out and the replacement text (the output of `awk`) is printed below it, so it is a trivial matter to undo the procedure.

Making optimal use of command-line `[g]awk` requires a mastery of regular expressions, a good understanding of `[g]awk`'s parsing of string constants and knowledge about characters that must be escaped. Such information does not stick in the memory unless used on a continuing basis. This limited interface to `[g]awk` allows you much of its power without a lot of learning. It should be said clearly that there is no substitute for knowing a real scripting language like PERL, python or ruby if you want to achieve the best results. This tool is not intended as a substitute for those skills. I strongly recommend that you use `gawk` rather than `awk` if you intend to use the search and replace capabilities—some of the program features simply don't work unless you use `gawk`.

The reason for calling this a baby version of `awk` is that the substitution functions `sub()`, `gsub()` and `gensub()` are not fully supported. The command line version of `gawk` allows a general regular expression for the search string, and this more modest front end does not.

The man page for `awk` is condensed but quite readable and informative, for a man page. The same cannot be said of the `gawk` man page. There is most likely a considerable overlap with the capabilities of OgreKit—an option for Find in TEXShop—but I just don't understand the meanings of its settings.

Here are some examples where I've found this of considerable utility:

- TEX tables, with fields separated by &;
- simpler forms of CSV, though the `datatool` package is much more capable;
- HTML tables, though this is better handled by importing to a spreadsheet and then importing to TEXShop with CSV2LaTeXTable;
- AFM files, where you need to parse two columns to write a minimal `.etx` file for use in fontinst;
- converting a collection of data rows like

Date: May 7, 2012.

```
bbS:-10:15:10
```

into a collection like

```
\shiftglyph{bbS}{-10}{-15}{10}
```

which is very useful when working with **fontinst** scripts;

- converting a group of duplicated lines like

  ```
  \skewkern{A}{35}
  ```

  so that the letter `A` increments over the group, resulting in

  ```
  \skewkern{A}{35}
  \skewkern{B}{35}
  \skewkern{C}{35}
  ...
  ```

2. Program usage

- Select the part of the text in your TEX document to which modifications should be made. The selection will in fact be applied to all lines touched by the selection, so at a minimum, it will apply to the entire line containing the cursor. Line here means "as divided by linefeeds", not screen lines.
- Select **BabyAwk** from the TEXShop **Macro Menu** to open the application **BabyAwk** in which you may enter your choices.
  - The choices are always the same as in your previous use of **BabyAwk**.
  - If **Input TeX text** is checked, all text on a line following the first unescaped % is not processed. You should not check this if you are processing a portion of text from a non-TeX source such as HTML.
  - The separators RS (record separator) and FS (field separator) affect how `awk` parses the input, while the other two (ORS, OFS) affect have output is formatted. Note that in the current version of `awk` (but not `gawk`), RS may be only a single character (by default, a linefeed character) but FS may be a general regular expression.
  - Awk first parses the input into records (usually, lines) controlled by RS and then parses each record into fields as specified by FS. The resulting fields are denoted `$1`, `$2`, ... The entire record is denoted by `$0` and the number of fields is set to the variable NF. The variable NR holds the index number of the current record, starting at 1. Any change to any one of `$1`, `$2`, etc, triggers a revaluation of `$0` with those changes.
  - The **BEGIN block** entry may be used to initialize variables used in the following blocks.
  - The **Search/condition** block is used to limit the lines which are processed. A search string must be delimited by /../, otherwise it is considered to be a condition. For example:

    ```
    /abc/   % apply only to lines containing abc
    /^abc/   % apply only to lines containing abc at the beginning
    /abc$/   % apply only to lines containing abc at the end
    /^abc$/   % apply only to lines exactly equal to abc
    !/abc/  % apply only to lines not containing abc
    $2 ~ /abc/  % apply only to lines where field 2 contains abc
    ```

```
$2 == /abc/  % apply only to lines where field 2 equals abc
NF > 0 % apply only to records of non-zero length
/[Aa]bc/  % apply to lines containing Abc or abc
```

If you leave this block blank, all records will be processed.

- The **Action** block specifies the output for each record. If you leave this blank (but the block **Search/condition** is not blank) the effect is the same as `print $0`; i.e., print the original line in its entirety if the line matches. The **Action** block usually contains **print** or **printf** statements to specify the output, but may also involve conditional statements. For example:

  ```
  print $3 % extract field 3 from every record (ie, output column 3)
  print $3 $4 % output columns 3 and 4 concatenated (no space)
  print $3, $4 % output columns 3 and 4 concatenated (with OFS)
  $NF="" print $0 % erase last field before printing line
  if ($1>1) {print $2 $3} % fields 2 and 3 with no separation, if field 1>1
  if ($1>1) {print $2,$3} % fields 2 and 3 separated by OFS, if field 1>1
  if ($1>1) {print $2,$3} % fields 2 and 3 separated by OFS, if field 1>1
  print "Record " NR+2, "Field 1: " $1, "Field 2: $2
  % the 4th record will output "Record 6 Field 1: $1 Field 2: $2"
  % without the quotes, and with $1 and $2 replaced by their actual values
  ```

- You may store a search with a name—fill in a name at the lower left corner and press **Store**. The search may be restored by choosing it via the popupmenu **Retrieve**.

3. Some useful regular expressions

The **FS** block in **BabyAwk** should have a number of predefined file separator patterns. It reads them from a file named `FSpatterns.txt` that should be in the same folder as the **BabyAwk** application. By default, that file reads

```
DEFAULT
COMMA [ \t]*,[ \t]*
COLON [ \t]*:[ \t]*
SEMICOLON [ \t]*;[ \t]*
AMPERSAND [ \t]*&[ \t]*
AMPER\\ [ \t]*&[ \t]*|[ \t]*\\\\\\\\[ \t]*
BRACES [ \t]*}[ \t]*{[ \t]*|[ \t]*}[ \t]*|[ \t]*}[ \t]*
BRACKETS [ \t]*\][ \t]*\[[ \t]*|[ \t]*\][ \t]*|[ \t]*\][ \t]*
PARENS [ \t]*\\)[ \t]*\\([ \t]*|[ \t]*\\([ \t]*|[ \t]*\\)[ \t]*
```

where the separator between the name and what follows is a tab character. You should not change the first line, but you can add or subtract other entries to meet your needs.

- The pattern

  ```
  [ \t]*,[ \t]*
  ```

  means zero or more spaces and/or tabs, a comma, then zero or more spaces and/or tabs. The pattern shows up in **babyAwk** under the name **COMMA**.

- The pattern **AMPERSAND** is similar:

  ```
  [ \t]*&[ \t]*
  ```

- The pipe character `|` signifies alternatives. The pattern `AMPERS\\` matches either `&` or `\\`, so is useful for parsing a row of a LaTeX table:

  ```
  [ \t]*&[ \t]*|[ \t]*\\\\\\\\[ \t]*
  ```

  You need eight backslashes here because `awk`'s string parsing mechanism reduces each `\\` to `\`, and then the matching reduces it by another factor of two.

- `}{|{|}` matches occurrences of either `}{` or `{` or `}`. That's good for splitting

  ```
  \shiftglyph{bbS}{-1}{-15}{10}
  ```

  into 6 fields (the last one empty), but does not work as expected if there are spaces or tabs between closing and opening braces. For example, if fed the string

  ```
  \shiftglyph{ bbS} {-1}{-15}{10}
  ```

  the matches would be `"{"`, then `"}"`, then `"{"`, then `"}{"`, then `"}{"`, then `"}"` giving 7 fields, not 6.

  To handle the most general case, disposing of all spaces and tabs beside a brace, use

  ```
  [ \t]*}[ \t]*{[ \t]*|[ \t]*}[ \t]*|[ \t]*}[ \t]*
  ```

  which is what you get when you choose **BRACES** in the FS menu.

  Explanation: this says to match either

  - zero or more spaces and/or tabs, then `}`, then zero or more spaces and/or tabs, then `{`, followed by zero or more spaces and/or tabs;
  - zero or more spaces or tabs, then `}`, then zero or more spaces and/or tabs;
  - zero or more spaces or tabs, then `{`, then zero or more spaces and/or tabs.

  Recall that matching in **[g]awk** is greedy—it matches the largest possible matching substring. So, when handed the string

  ```
  \shiftglyph {  bbS   } { -1}  { -15}{ 10}
  ```

  the first match will be `" {   "` (note that `"{"` would be the smallest possible match), the second `"    } { "`, then `"}  {   "`, then `"}{ "`, and finally `"}"`.

  Brackets may be handled similarly, but require `\[` and `\]` to remove ambiguity about their semantic meanings, and parentheses require double backslashes.

- The default behavior of FS is obtained by setting `FS=" "` (a single space character), but the actual effect is as if you had written `FS="[ \t]+"` (a run of one or more spaces and/or tabs), except for one peculiarity: the default form effectively trims all white space from the beginning and end of each record before separating the fields, though the white space will remain in `$0`, and the second form does not. **BabyAwk** tries to assist you here by trimming white space from each line if you specify a non-default value for `FS`. (This is effective only in case `RS` takes its default value `\n`.) A trick you may use to force the stripping of white space from the beginning and end in `$0` is an **Action** like

  ```
  $1 = $1; print ...
  ```

  which forces the re-computation of `$0` with trimmed values. (This works only when `FS` takes its default value.)

4. Examples

4.1. Rearrange column order in table. Suppose I have a LaTeX table whose columns I wish to rearrange. The rows are assumed to be arranged like

```
col 1 & col 2 & col 3\\
col 1a &col 2a& col 3a\\[2pt]
...
```

Select all rows and run the macro BabyAwk, filling in the fields as follows:

```
<FS> AMPERSAND
<OFS> &
<Action> print $2, $1, $3
```

and press Go. It's a little more work if the exchange involves the last column, as that contains the end of row marker. Use

```
<FS> AMPER\\
<OFS> &
<Action> print $3, $2, $1 "\\" $4
```

(Note that field 4 in this case is what follows \\.)

4.2. Enter rows with one column entry changing in numeric sequence. Suppose we wish to insert new rows in the document, like

```
\xyz{1}
\xyz{2}
%...
```

The way which may be conceptually simplest (involving least programming) is to enter an array of numbers using the macro Insert Sequence which can handle numbers and letters, increasing or decreasing. Select the entire block and call BabyAwk, entering just `print "\xyz{" $1 "}"` in the Action block, then press Go. (Note: if you were doing the equivalent command line awk command, this would be incorrect—you would have to write `print "\\xyz{" $1 "}"` to get the backslash past the awk string interpreter, which interprets single backslash string fragments like `\t` and `\n` as `tab` and `newline` respectively, and discards those that do not have special meaning. BabyAwk handles this for you by doubling every backslash in an Action block, except for those in a `\substr` block, which are handled with special rules. This is usually appropriate for TeX documents, but may not always produce what you intended.

You could also handle this in another way involving just one step. First, make a blank row in your document where you wish the new items to appear, and then, in the BabyAwk window, enter the following in the BEGIN block:

```
for(i=1;i<11;i++) print "\xyx{" i "}"
```

and press Go.

4.3. Add a fixed amount to a column. Suppose our data is a number of rows like

```
\xy{10}{2}
```

and we wish to add 30 to each entry in the last column. We have only to set FS to BRACES and set Action to

```
$3 = $3+30; print $0
```

and press Go.

4.4. Convert a vertical list to a comma-separated list.

The data is assumed to be like

```
ab
cd ef
g
...
```

Select the vertical list, run the BabyAwk macro and set the output record separator ORS to a single comma and Action to `print`, then press Go. The result should be

```
ab,cd ef,g
```

4.5. Add some text to the beginning of each non-empty line.

The data is assumed to be like

```
ab
cd ef

g
```

Select the lines, run the BabyAwk macro and set Search / condition to `length($0)>0` and Action to `print "XXX" $0`, then press Go. The result should be

```
XXXab
XXXcd ef
XXXg
```

4.6. Modify a column with incrementing letters.

The data is assumed to be like (eg, one line duplicated a number of times)

```
1 A 2 3
1 A 2 3
1 A 2 3
1 A 2 3
```

and we would like A to increment. Select the lines, run the BabyAwk macro and set Action to `$2=UC[NR]; print $0`, then press Go. The result should be

```
1 A 2 3
1 B 2 3
1 C 2 3
1 D 2 3
```

This may require a little explanation. If BabyAwk sees either of `UC[` or `lc[` in your code, it defines arrays UC[1]=A,...,UC[26]=Z, lc[1]=a, etc. Recall that `NR` is an `awk` variable containing the current record number. So, the action replaces field 2 with a letter, which updates `$0`, which you then print.

4.7. ReplaceAll within selected block.

This is better handled using the OgreKit 'Find'. You could do it in BabyAwk, though less intuitively. **CAUTION: this may produce unexpected results unless you use `gawk` instead of `awk`.**

The data is assumed to be like

```
abcd
cd ef
acd
g
```

and we would like to add replace all occurrences of `cd` in that block to `CD`. Select the lines, run the BabyAwk macro and set Action to `gsub(/cd/,"CD"); print`, then press Go. The result should be

```
abCD
CD ef
aCD
g
```

Note that changes take place only within individual lines, and that the `gsub` (global substitution) built-in function uses `$0` as the target if no string is specified as its third argument.

IMPORTANT NOTES:

- If you want to include a double-quote character in your replacement string, you must write it as `\"` so that it does not cause the `awk` string interpreter to misconstrue it as the end of the string.
- Any occurrence of / in the search string must be replaced by `\/`, for the same reason.
- BabyAwk partially parses your use of `sub()` and `gsub()` to warn you of syntax errors and to escape some special characters. In particular, if you include the character `&` is your arguments, it will be replaced by `\&` so that it searches for the literal character `&`.
- If BabyAwk, because of its limited abilities, fails to enter the correct awk command line and you are sufficiently skilled to correct it, you may do so by editing the last line immediately before pressing Go.