# bawk – bad awk

Ashley An (aya2121), Christine Hsu (chh2132), Mel Sawyer (ms5346), Victoria Yang (vjy2102)

## Language Description

Our proposed language is a text processing language based off of awk. awk is best suited for dealing with files formatted into rows and columns, but our goal is to create a text processing language that can easily be applied to plain text files. bawk aims to make it easy to read, analyze, and write to text files in a simple and intuitive syntax mimicking that of awk.

## Applications of Language

Our language can be used to take a file and break it down into words and their corresponding frequencies. It would have file reading capabilities and data structures that users can use to store the words that are read. Users would be able to create functions that can manipulate the frequencies and words and that can create ASCII word art (e.g. where the size of the art corresponds to the frequency of the word). Ultimately, bawk would contain and enhance the features of awk that are best suited for files that aren't formatted as tables or spreadsheets.

## Language Features

- **Regex & string matching**

  **Important distinction:** in traditional awk, the input stream is stored as lines (e.g. chunks of text separated by the newline character). The types of files our language is designed to accommodate primarily won't have this structure. Instead of lines, we'll be separating the input by sentences. The example below shows how this difference might materialize:

  awk:

  Input file:
  this is a line \n
  another line \n
  another \n

  ```
  awk '/another/{ print $0 }'
  ```

  output: another line \n
  another \n

  bawk:

  Input file: The quick brown fox jumps over the lazy dog. The yellow cactus oozes purple blood.

  ```
  bawk '/yellow/{ print $0 }'
  ```

  output: The yellow cactus oozes purple blood.

| Type | Description |
|---|---|
| / and / | Indicates the beginning and end of pattern to search for |
| . | Replaces any character in the input string (ex. p.n would look for pan, pbn, etc.) |

| | |
|---|---|
| ^ | Finds records starting with entry (ex. ^X would look for sentences starting with a capital X) |
| $ | Finds records ending with entry |
| [...] | Bracket expression: matches any character within brackets |
| [^...] | Complementary bracket expression: does not match any of the characters within brackets |
| \| | Alternation operator: allows alternatives (e.g. 'hi\|bye' would look for either hi or bye |
| * | Allows a symbol to be repeated as many times as possible to find a match (ex. ab*c would look abc, abbc, abbbc, etc.) |

- **String formatting**
  This is a feature of the Standard C library- I'm unsure if we could port it over or if we'd have to rewrite the functionality ourselves.

| **Type** | **Description** |
|---|---|
| printf | Version of `print` which allows for a format string |
| - | Left justify (ex. printf "%-6s", "foo" results in "foo  " |
| width (%3s) | Specify how much space the string should take up, extra is padded by spaces. |
| .precision | Indicates the desired precision for decimal representations |

- **File reading**
  Both awk and bawk read files through the standard input. This can either be passed as a parameter or piped through another program.

  For our language, when the user passes a file through standard in, the contents of the file will be read and stored in an array called `FILE` where each element is a word from the file.

- **Data Types**:

| **Type** | **Description** |
|---|---|
| int | Integer |
| String | Ordered list of characters. |
| char | ASCII definition of character |
| bool | Boolean value that can be assigned true or false. |

| | |
|---|---|
| float | Floating point. |
| array | Series of ordered values of the same type. |
| map | Stores String key and associated Integer value. |
| arr = [] | Initialize an empty array |
| map = {} | Initialize an empty map |

- **Operators:**

| **Type** | **Description** |
|---|---|
| + | Used for addition of integers/floats as well as string concatenation. |
| - | Subtraction of integers/floats. |
| * | Multiplication of integers/floats. |
| / | Division of integers/floats. |
| ++, -- | Incrementation and decrementation of integers. |
| [] | Indexing arrays. |
| x <= y | String/integer comparison - x less than or equal to y. |
| x => y | String/integer comparison - x greater than or equal to y. |
| x == y | String/integer comparison - x equal to y. |
| x != y | String/integer comparison - x not equal to y. |
| x ~ y | String x matches the regexp denoted by y. |
| x !~ y | String x does not match the regexp denoted by y. |
| in | Used to check if value is present in array. |

- **Keywords:**

| **Type** | **Description** |
|---|---|
| if…else | Conditional statement. |
| for (...) | Loops through a block of code a number of times specified by a corresponding number of iterations. |
| while (...) | Loops through a block of code until its corresponding conditional statement is |

| | |
|---|---|
| | not met. |
| BEGIN | Executed once at the beginning of the program before the first input is read. |
| END | Executed once at the end of the program before the first input is read. |
| # | Indicates the start of a comment. |
| function | How the user starts a function declaration. |
| FILE | The array in which file contents are stored upon reading where each element is a word from the file. |
| return | Returns data type at the end of a function. |

## Sample Program

```
# passed in file will automatically be stored in FILE (see above)
allWords = {};

allWords = getFreq(allWords);
sum = getSum(allWords);

# print table with word frequencies as percentages
for (word in FILE) {
      freq = allWords[word]/sum;
      printf("%s\t|%.6f", word, freq);
}

# function to create a dictionary mapping each word to the number of
# times it appears in the input file
function getFreq(wordsMap) {
      for (word in FILE) {
            if !(word in allWords) {
                  allWords[word] = 0;
            }
            else {
                  allWords[word] = allWords[word] + 1;
            }

      }
      return allWords;
}

# returns the total number of instances of words in input file
function getSum(wordsMap) {
      sum = 0;
      for (key in wordsMap){
            sum = sum + wordsMap[key];
      }
      return sum;
}
```