bawk Language Reference Manual

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

Table of Contents

Introduction

bawk 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.

In bawk, programs are made up of four blocks (three of which are required but can be empty). These blocks can contain functions, statements, or expressions. An expression in bawk has a value – for example, assignment, initialization, and all operators. Statements do not have values and are executed in sequence as a method of control flow – for example, loops, if statements, and function declarations.

Lexical Conventions

Identifiers

Identifiers are primarily used for variable declaration or function declaration, and can be any combination of numbers , letters, and the _ symbol as long as the resulting string isn't an existing keyword.

Valid	**Invalid**
myVariable my_variable myVariable1 _	1variable function my.variable My-variable

Literals

We have 4 types of literals: array literals, integer literals, string literals, and regex literals.

Array

Array literals are defined as follows:

```
arr[] a = [1, 2, 3];
```

Integer

Integer literals are defined as a sequence of one or more digits representing an integer, with the leading digit being non-zero (i.e. [1-9][0-9]*).

```
int x = 100;
```

String
String literals are defined as a sequence of ASCII characters enclosed by a pair of double quotation marks.

```
string s = “Hello”;
```

Regex
Regex (regular expression) literals are denoted with a rgx type and enclosed in a pair of single quotation marks.

```
rgx example = ‘p.n’;
```

Some of the syntax of regex is listed in the “Regular Expressions” section below.

Comments

Comments are denoted as follows:

```
# Your comment here
```

There are no multi-line comments.

Keywords

Type	Description	Example

if...else	if...else denotes a conditional statement.	`int x = 0;`
		`if (x < 10){`
		`    x++;`
		`}`
		`else{`
		`    print(x);`
		`}`

for (...)	for loops through a block of code a number of times specified by a corresponding number of iterations.
	There are 2 possible structures for what is contained in the parentheses:
	1) For a normal for loop, the parentheses must contain 3 statements delimited by semicolons. The first segment
		`for (int i = 0; i < 10; i++) {`
		`    print(x);`
		`}`
		
		`for (int i in array) { … }`
		
		`for ((int k, string v) in map { … }`

	contains the declaration and initialization of a counter variable, the second is the condition under which the for loop will continue looping, and the third is what happens to the counter variable after each iteration. 2) bawk also offers an enhanced for loop to iterate over arrays and maps using the keyword "in". For arrays, bawk requires declaring a variable and its type that matches the type of the array, followed by "in" and the name of the array. For maps, bawk requires declaring a (key, value) tuple initialized with the same types as the key and value of the map, followed by "in" and the name of the map. Variables declared inside a for loop (within both the condition and body) are locally scoped within the brackets that define the for loop.	

while (...)	while loops through a block of code until its corresponding conditional expression is not met.
	Variables declared inside the while loop are locally scoped within the brackets that define the while loop.
		`int x = 0;`
		`while` `(x < 10) {`
		`    print(x);`
		`    x++;`
		`}`

in	A keyword used in enhanced for loops to iterate over all elements in an array or (key, value) elements in a map.	`for (int i in array) { … }`
		
		`for ((int k, string v) in map { … }`

CONFIG	Denotes the beginning of the	**`CONFIG`** `{`

	configuration block which can be used to redefine FS and RS	`    RS = ‘.’`
		`    FS = ‘,’`
		`}`

BEGIN	Executed once at the beginning of the program before the first input is read. Function declarations can only occur in the BEGIN block, but they can occur anywhere in the block. Variables declared in the BEGIN block are global across all blocks.	`BEGIN {`
		`  function int add(int a, int b){...}`
		`  function int mul(int a, int b){...}`
		`}`

LOOP	Code inside the LOOP block is continuously executed until the file is completely read through.	**`LOOP`** `{`
		`  print($1);`
		`}`

END	Executed once at the end of the program. You cannot declare any new functions in END.	`#arr[] a is instantiated and`
		`#functions are called on it in`
		`#BEGIN and LOOP`
		`END {`
		`   print (a);`
		`}`

function	function denotes how the user starts a function declaration; an explicit return type is required. Nested functions are not allowed.	**`function`** `int add(int x, int y) {`
		`    return x + y;`
		`}`

return	Returns data type at the end of a function. Will halt execution and any code after return statement does not run.	`function int add(int x, int y) {`
		`    `**`return`**` x + y;`
		`}`

RS	A built-in string variable for the “record separator”. The default value is a newline character, but this can be changed in the CONFIG block.	**`RS`** `= ‘\r\n’`

FS	A built-in string variable for the “field separator”. The default value is a space, but this can be changed in the CONFIG block.	`# Fields within a record should be`
		`# separated by commas.`
		`FS` `= ‘,’`

NF	A built-in special variable that retrieves the number of fields in the current record that is being read in LOOP.	`# Checks if the number of fields`
		`# in the current record is greater`
		`# than 2.`
		`if` **`NF`** `> 2`

$	A special variable that retrieves the string field in the current record specified by the number after $.	`$0 # refers to the whole record`
		`$1 # first field in current record`
		`$2 # second field in current record`

true	A boolean keyword that means something is true.	`boolean flag =` **`true`**`;`
		`if (flag){`
		`  print(“hello”);`
		`}`

false	A boolean keyword that means something is false.	`boolean flag = true;`
		`int x = 6;`
		`if (x>5){`
		`  flag =` **`false`**`;`
		`}`
		`if (!flag){`
		`  print (“x is greater than 5”);`

Operators

Type	Description
+	+ is used for the addition of integers.
&	& is used for string concatenation, e.g. `string s = “hello” & “world.”`

&&	&& represents logical AND, e.g.
	`int i = 5;`
	`if (i == 5` **`&&`** `i < 6){`
	`   print (i);`
	`}`

||	|| represents logical OR, e.g.
	`int i = 5;`
	`if (i < 6` **`||`** `i > 10){`
	`   print (i);`
	`}`

-	- is used for the subtraction of integers.

*	* is used for the multiplication of integers.
/	/ is used for the division of integers.
++, --	++ and -- are used for the incrementation and decrementation respectively of integers by 1.
+=, -=	+= and -= are incrementation and decrementation of integers respectively by a specified integer, e.g. `x += 5`

[]	[] is used to read from or write to arrays at a certain index, e.g.:
	
	`if (array[0] == 5){`
	`    return array[0];`
	`}`

{}	{} is used to read from or write to maps elements with a given key, e.g.
	
	`# map: {2 : “yo”, 3 : “hey”, 5 : “hello”}`
	`if (map{5} == “hello”) {`
	`    return true;`
	`}`
	`# returns true`

x <= y	The operator <= is used for string or integer comparison and returns a boolean that indicates if x less than or equal to y.
x => y	The operator => is used for string or integer comparison and returns a boolean that indicates if x greater than or equal to y.
x < y	The operator < is used for string or integer comparison and returns a boolean that indicates if x less than y.
x > y	The operator > is used for string or integer comparison and returns a boolean that indicates if x greater than y.
x == y	The operator == is used for string, integer, or boolean comparison and returns a boolean that indicates if x is equal to y.
x != y	The operator != is used for string, integer, or boolean comparison and returns a boolean that indicates if x not equal to y.
!	!= is the logical not operator.
;	; indicates the end of an expression.

Regular Expressions

Unlike traditional awk, bawk introduces the concept of the regular expression type, indicated by `rgx example = ‘t.x’`

Regex Operators

x ~ y	String x matches the rgx denoted by y.
x !~ y	String x does not match the rgx denoted by y.
x % y	Rgx x equals the rgx denoted by y.
x !% y	Rgx x does not equal the rgx denoted by y.

Regex & string matching

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. Note that this is different from the $ special variable when defined in a regex expression enclosed in single quotes.
[...]	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.)

Types

Data Types

Type	**Description**
int	Integer.
string	Ordered list of characters.
bool	Boolean value that can be assigned true or false.
rgx	Regular expression type. See the “Regular Expression” section for more information about how to create and use these types.
int[]	Series of ordered values of type int. Dynamically resizable.
string[]	Series of ordered values of type string. Dynamically resizable.
bool[]	Series of ordered values of type bool. Dynamically resizable.
rgx[]	Series of ordered values of type rgx. Dynamically resizable.
map<type, type>	Stores key and associated value. Key and value can be any type and can be different from each other, and are determined by the types in the triangle brackets (first for key, and second for value). Similar to associative arrays, implemented using hcreate_r. Dynamically resizable.
int[] arr1 = [] string[] arr2 = [] bool[] arr3 = [] rgx[] arr4 = []	Initialize an empty array where the type of the array is declared before the name of the array. If a value with incorrect type is added, an error is thrown.
map<type, type> name = {}	Initialize an empty map where the variable name is name and the types of the key and value are determined by the types declared in the triangle brackets. If a pair with incorrect type is added, an error is thrown. Ex: `map<int, string> alpha = {}` A map literal can be represented as a series of key : value pairs separated by commas, e.g. `{1 : “a”, 2 : “b”, 3 : “c”, 4 : “d”, 5 : “e”}`

Type Checking
bawk is strongly typed language. This means that each data type is predefined as part of the language, and these predefined data types must be used to describe all constants or variables in a given program.

Special Features

File I/O
Both awk and bawk read files through the standard input. This can either be passed as a parameter or piped through another program. A file must be passed as the command-line argument to a bawk program, otherwise a "file not found" error will be thrown.

Example:
```
    ./bawk_test < test.txt | ./bawk_test2 > output.txt
```

The output of bawk will be sent to standard out. This output can be piped to the input of another bawk program. Input is read in automatically, and each line is split into fields which can be accessed using the $n special variable.

BEGIN, END, and LOOP Blocks
Functions should be declared in BEGIN, and no code can be written outside of these three blocks. The LOOP block between the BEGIN and END blocks should loop through the entire file.

Config File
A config file can be included to change the two built-in variables RS and FS (both are described above in the Keywords section).

Compilation
The program will be compiled into an executable. An input file must be piped into the newly formed executable to run it (see File I/O Section).

Standard Library Features

Functions
`int string_to_int(string a)`

This function converts a string into an int provided that the string follows the regex [0-9]*.

`string int_to_string(int a)`
This function converts an int into a string.

`int length(arr[] a)`
This function returns the length of the array a.

`int size(map a)`
This function returns the size of the map a.

`arr keys(map a)`
This function returns an array of the keys of map a.

`arr values(map a)`
This function returns an array of the value of map a.

`bool contains(var1, arr[] a):`
This function is used to check if value is present in array or map. This can also be used to determine whether an array index has been assigned or not. Returns `true` if item in array, `false` otherwise, e.g.:

```
if (!contains($0, array)) {
     array[length(array) - 1] = $0
}
```

`int indexOf(arr[] a, var)`
Returns the index of the first instance of `var` within the array `a` by value. If var is not in a, return -1.

```
arr[] a = {1, 2, 3, 2};
a.contains(2) # returns 1
a.contains(0) # returns -1
```

`void print(var1, var2, …)`
This function prints any number of inputs, separated by a space. `print()` can take parameters of different types, as long as the different types are separated by a comma. Examples of this distinction shown below:

```
print("hello", 1.2, 3)
```
outputs:
```
        hello 1.2 3
```

```
print("hello" + 1.3, 2)
```
throws an error because a string cannot be added to a float.

`void println(var1, var2, ...)`
This function prints any number of inputs, separated by a newline.

```
    println("hello", 1.2, 3)
```

outputs:

```
        hello
        1.2
        3
```

Sample Programs

Word Frequency

```
# This program stores each unique word in a file as the key in an
# associative array and the number of times it appears in the file as
# the value.
BEGIN {
    map wordmap = {};
    void function addToMap(map wordmap, int i) {
        if (wordmap.contains(i)) {
            wordmap[i] += 1;
        }
        else {
            wordmap[i] = 1;
        }
    }

}

LOOP {
    # Loop through all fields (words), store in word map
    for (int i = 1; i <= NF; i++) {
        addToMap(wordmap, $i);
    }

}


END {
    for (int i = 0; i < length(keys(wordmap)); i++) {
        print (keys(wordmap)[i], wordmap[keys(wordmap)[i]])
    }

}
```

Regular expression matching

This program searches every record for lines starting with an integer and replaces them with an empty string. Then it prints each line with no integer at the beginning.

```
BEGIN {
}

LOOP {
     rgx reg = ‘[0-9]*’;
     if ($1 ~ reg) {       # if the first line is an integer
          for (int i=2; i < NF; i++){
               print($i & “ “);
          }
     }
     else {
          print($0);       # prints entire line
     }
}

END {
}
```