COMS W4115 – Programming Languages & Translators

# The Bawk Language Reference Manual

Kevin Graney
kmg2165@columbia.edu

June 28, 2013

## Contents

# 1 Introduction

The bawk language, who's name is derived from 'binary awk', is intended to be a small, special-purpose language for the parsing of regularly formatted binary files. In the spirit of awk, bawk will match rules in binary files and extract data based on these rules. Bawk attempts to solve the problem of quickly decoding a regularly formatted binary file to extract information, much the way awk solves this same problem for regularly formatted text files.

# 2 Lexical Conventions

A program consists of a single bawk language character file. It is translated in several phases as described in §7.

## 2.1 Tokens

There are six classes of tokens: identifiers, keywords, constants, string literals, operators, and separators. White space characters are ignored except as they separate tokens or when they appear in string literals.

## 2.2 Comments

Comments follow the ANSI C style, beginning with `/*` and ending with `*/`. They do not nest, and they cannot be present in quoted strings.

## 2.3 Identifiers

An identifier is a sequence of uppercase and lowercase letters A–Z, the numerals 0–9, and the under score character. The first character of an identifier can not be a numeral.

⟨*identifier*⟩ ::= ⟨*letter*⟩
| ⟨*identifier*⟩ (⟨*letter*⟩ | ⟨*digit*⟩)

## 2.4 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise: `if`, `else`, `return`.

## 2.5 Constants

Bawk only supports a single kind of constant: the integer constant. The semantic meaning of integer constants depends on the context in which they occur. Except for inside of pattern expressions (see §6.1), the following is true. An integer constant consisting of a sequence of digits is taken to be in decimal. A sequence of hexadecimal digits prefaced by `0x` is taken to be in hexadecimal (base 16).

## 2.6 String Literals

A string is a sequence of characters enclosed in double quotes.

# 3 Meaning of Identifiers

Identifiers can have several different meanings based on how they are first used. Once an identifier is associated with a particular meaning it cannot be disassociated with that meaning.

### 3.1 Function Name

Previously unused identifiers become function names once a function is declared with a given name.

### 3.2 Variable Name

Previously unused identifiers become variable names once an assignment expression (see §5.7) executes with the identifier on the left hand side.

### 3.3 Pattern Binding Variable Name

Previously unused identifiers become binded variable names once a pattern expression with a binding pattern token (see §6.2) matches.

### 3.4 Special Identifiers

There are certain special identifiers that behave like variables, but whose meaning affects the program execution.

- `RP` – The *file pointer* variable indicates where in the file pattern statements begin matching (see §4.2). The value contains the number of bytes from the beginning of the file. At the start of the bawk program this value defaults to 0.
- `LE` – Force *little-endian* interpretation of integers and strings while pattern matching in the file. A value of 0 indicates big-endian interpretation and a value of 1 indicates little-endian interpretation. The default value is 0.
- `_` – The `trash` variable is a write-only identifier that cannot be read. It is useful for binding don't-care values in pattern matching expressions.

## 4 Statements

The expressions described in §5 are a specific form of statement in the bawk language. There are three different types of statements, and a list of statements forms a bawk program. The three different types of statements are: expressions (described in §5), block statements, and pattern statements.

⟨*program*⟩ ::= ⟨*statement-list*⟩

⟨*statement-list*⟩ ::= $\epsilon$
| ⟨*statement*⟩
| ⟨*statement-list*⟩ ⟨*statement*⟩

⟨*statement*⟩ ::= ⟨*expression-statement*⟩
| ⟨*block-statement*⟩
| ⟨*pattern-statement*⟩
| ⟨*function-declaration*⟩

### 4.1 Block Statements

Block statements are used to combine a sequence of statements into a single statement. The sequence of statements inside the block is executed in order when the block itself is executed. Variables first used inside the block are locally scoped, but names from outside the block are also available.

⟨*block-statement*⟩ ::= '`{`' ⟨*statement-list*⟩ '`}`'

### 4.2 Pattern Statements

Pattern statements allow a type of pattern matching to be performed on the binary file. Pattern statements consist of a pattern expression (see §6) and a statement. If the pattern expression matches the data file at the location of `RP` at the time of evaluation of the pattern statement, then the corresponding statement is executed. If the pattern does not match, then the statement is not executed.

⟨*pattern-statement*⟩ ::= '`/`' ⟨*pattern-expression*⟩ '`/`' ⟨*statement*⟩

### 4.3 Expression Statements

An expression statement is an expression followed by a semicolon. Expressions are described in §5.

⟨*expression-statement*⟩ ::= ⟨*expression*⟩ '`;`'

### 4.4 Function Declaration

A function declaration is a statement declaring a new function. Function declarations consist of an identifer to serve as the function name, followed by an open parenthesis, an optional comma separated list of identifiers to serve as function parameter names, a closing parenthesis, and a statement to serve as the function body.

⟨*function-declaration*⟩ ::= ⟨*identifier*⟩ '`(`' ⟨*function-decl-params*⟩ '`)`' ⟨*statement*⟩

⟨*function-decl-params*⟩ ::= $\epsilon$
| ⟨*identifier*⟩
| ⟨*function-decl-params*⟩ '`,`' ⟨*identifier*⟩

## 5 Expressions

### 5.1 Primary Expression

⟨*primary-expression*⟩ ::= ⟨*constant*⟩
| ⟨*identifier*⟩
| ⟨*string-literal*⟩
| ⟨*function-call*⟩
| '`(`' ⟨*expression*⟩ '`)`'

⟨*expression*⟩ ::= ⟨*assignment-expression*⟩
| ⟨*additive-expression*⟩

## 5.2 Function Calls

A function call is an identifier, known as the function name, followed by a pair of parenthesis containing a, possibly empty, comma-separated list of expressions. These expressions constitute arguments to the function. When functions are called a copy of each argument is made and used within the function, that is, all function calls pass arguments by value.

⟨*function-call*⟩ ::= ⟨*identifier*⟩ '`(`' ⟨*function-call-params*⟩ '`)`'

⟨*function-call-params*⟩ ::= $\epsilon$
| ⟨*expression*⟩
| ⟨*function-call-params*⟩ '`,`' ⟨*expression*⟩

## 5.3 Multiplicative Operators

The multiplicative operators, `*` and `/`, are left associative. The result of `*` is multiplication of the two operands, and the result of `/` is the quotient of the two operands. The quotient with a second operator of zero yields an undefined result.

⟨*multiplicative-expression*⟩ ::= ⟨*primary-expression*⟩
| ⟨*multiplicative-expression*⟩ '`*`' ⟨*primary-expression*⟩
| ⟨*multiplicative-expression*⟩ '`/`' ⟨*primary-expression*⟩

## 5.4 Additive Operators

The two additive operators, `+` and `-`, are left associative. The expected arithmetic operation is performed on integers. The result of the `+` operator is the sum of the operands, and the result of the `-` operator is the difference of the operands.

⟨*additive-expression*⟩ ::= ⟨*multiplicative-expression*⟩
| ⟨*additive-expression*⟩ '`+`' ⟨*multiplicative-expression*⟩
| ⟨*additive-expression*⟩ '`-`' ⟨*multiplicative-expresssion*⟩

## 5.5 Relational Operators

The relational operators are all left associative. The operators all produce 0 if the specified relation is false and 1 if the specified relation is true. The operators are `<` (less than), `>` (greater than), `<=` (less than or equal to), `>=` (greater than or equal to).

⟨*relational-expression*⟩ ::= ⟨*additive-expression*⟩
| ⟨*relational-expression*⟩ '`<`' ⟨*additive-expression*⟩
| ⟨*relational-expression*⟩ '`>`' ⟨*additive-expression*⟩
| ⟨*relational-expression*⟩ '`<=`' ⟨*additive-expression*⟩
| ⟨*relational-expression*⟩ '`>=`' ⟨*additive-expression*⟩

## 5.6 Equality Operators

The `==` (equal to) and `!=` (not equal to) operators compare operands for equality. Like the relational operators, the equality operators produce 0 if the relationship is false and 1 if the relationship is true. The equality operators have a lower precedence than the relational operators, and are also left associative.

⟨*equality-expression*⟩ ::= ⟨*relational-expression*⟩
| ⟨*equality-expression*⟩ '`==`' ⟨*relational-expression*⟩
| ⟨*equality-expression*⟩ '`!=`' ⟨*relational-expression*⟩

### 5.7 Assignment Expressions

All assignment expressions require a modifiable identifier on the left hand side. The only type of modifiable identifier is a variable (see §3.2). The assignment operator, `=`, is right-associative and the return value of assignment is the value being assigned.

⟨*assignment-expression*⟩ ::= ⟨*equality-expression*⟩
| ⟨*identifier*⟩ '`=`' ⟨*assignment-expression*⟩

## 6 Pattern Expressions

A pattern expression matches content in the binary data file. Pattern expressions are distinct in syntax from the expressions described in §5.

⟨*pattern-expression*⟩ ::= $\epsilon$
| ⟨*pattern-token*⟩
| ⟨*pattern-expression*⟩ ⟨*pattern-token*⟩

⟨*pattern-token*⟩ ::= ⟨*pattern-constant*⟩
| ⟨*pattern-binding*⟩
| ⟨*pattern-variable*⟩
| ⟨*string-literal*⟩

Pattern expressions can contain constants or typed variable bindings. Each constant or binding is known as a pattern term, and pattern terms are whitespace delimited. Already bound pattern variables can also be used in pattern statements.

### 6.1 Constants

Inside pattern expressions, constants work differently than those described in §2.5. All constants in pattern terms are implicitly expressed in hexadecimal form, and the `0x` prefix required to express hexadecimal constants in §2.5 must be ommitted.

Constants are read in hexadecimal as bytes. For this reason leading zeros on a value have semantic meaning. For example `/0000abcd/` matches the four byte pattern `00 00 ab cd` in the binary file, while `/abcd/` matches the two byte pattern `ab cd`. Spaces in the constants, however, do not have any semantic meaning as long as the spaces are on byte boundaries. That is, `/0000abcd/` is a semantically identical pattern to `/00 00 ab cd/`. It is important to note, however that `/01ab/` is *not* identical to `/0 1 a b/`.

If a constant has an odd number of hexadecimal digits, an implied `0` is added to the left-most side of the value. That is `/123/` is the same as `/0123/` (is the same as `/01 23/`). For this reason the example of `/0 1 a b/` is identical to `/00 01 0a 0b/` (and `/00010a0b/`).

### 6.2 Bindings

A bind pattern term consists of an identifier, a colon (`:`), and a bind type. Valid bind types are: `int1`, `int2`, `int4`, `uint2`, `uint4`. The `int1` type matches a one-byte integer, the `uint4` type matches an unsigned 4-byte integer. Identifiers have the same naming rules as §2.3.

⟨*pattern-binding*⟩ ::= ⟨*identifier*⟩ '`:`' ⟨*bind-type*⟩

⟨*bind-type*⟩ ::= '`int1`'
| '`int2`'

| 'int4'
| 'uint2'
| 'uint4'

### 6.3 Pattern Variables

Previously bound pattern variables can be used in a pattern expression to match the previously bound value.

⟨*pattern-variable*⟩ ::= ⟨*identifier*⟩

### 6.4 String Literals

String literals are also supported in pattern expressions. The syntax is exactly the same as strings in §2.6. The string literal is converted to and interpreted as a sequence of ASCII-encoded bytes, exactly how constants are (see §6.1).

## 7 Interpretation and processing

Bawk is a bytecode interpeted language, meaning the native bawk code compiles to intermediate form bytecode. A bytecode interpreter is also provided, which interprets and executes the bytecode over the input file. The process of compiling bawk to bytecode is done on the the fly, so the use of bawk is seamless. From the user's perspective, the bawk interpreter takes two inputs for execution: (1) text in the bawk langauge, and (2) a binary input file to run the bawk program on. This workflow is similar to that of a basic awk workflow, and the language itself is inspired by awk, with similar program constructs. In bawk, however, the interpreter compiles the bawk language program to bawk bytecode, which is then executed by the bawk bytecode interpreter.