GAWK Language Reference Manual

Albert Cui, Karen Nan, Mei-Vern Then, & Michael Raimi

“So good, you’re gonna GAWK.”

1.0 Introduction

This manual describes the GAWK language and is meant to be used as a reliable guide to the language.

For the most part, this document follows the outline of the C Language Reference Manual, as described in Appendix A of *The C Programming Language*, by Brian W. Kernighan and Dennis M. Ritchie.

2.0 Lexical Conventions

A program consists of one or more translation units stored in files. It is translated in several phases. The first phases do low-level lexical transformations. When the preprocessing is complete, the program has been reduced to a sequence of tokens.

2.1 Tokens

Tokens will have C-like syntax. There are five classes of tokens: identifiers, keywords, string literals, operators and separators. Blanks, tabs, and newlines will be ignored, except for white space that is required to separate two consecutive tokens.

2.2 Comments

The characters /* introduce a comment and */ terminates them. Comments do not nest.

2.3 Identifiers

An identifier is a sequence of letters and digits of any length. The sequence must start with a letter; all following characters can be any combination of letters, numbers, or the underscore _ (which counts as a letter). Upper and lower case letters are different.

2.4 Keywords

The following are identifiers are reserved as keywords and cannot be used otherwise:

```
if              return          struct          true
else            this            int             false
while           void            string
for             null            bool
```

2.5 Constants

Constants are not supported.

2.6 String Literals

A string literal is a sequence of one or more escape characters or a non-double quote character, surrounded by double quotes, as in “...”. A string has type ‘string’ and is initialized with the given characters.

3.0 Syntax Notation

In the syntax notation used in this manual, syntactic characters are indicated by *italics*, and literal strings in `typewriter` style. An optional terminal or nonterminal symbol carries the subscript "*opt*", so that

{ $expression_{opt}$ }

means an optional expression, enclosed in braces.

4.0 Meaning of Identifiers

Identities or names refer to many things: functions, tags of structures, members within the structures, and variables. Interpretations of variables depend on two main attributes: *scope* and *type*. The scope is the region of the program where it is known and type determines the meaning of the values in the variable.

4.1 Basic Types

There are four fundamental types: strings; integers; booleans; and void.

Variables declared as strings (`string`) are large enough to store any sequence of combinations from the character set.

Integer (`int`) variables have the natural size suggested by the host machine architecture. It represents all signed values unless otherwise specified.

Booleans (`bool`) variables only hold "true" or "false" values.

The `void` type is an empty set of values and is the return type of functions that generate no value. It nonexistent value of a `void` cannot be used in any way. The expression can only be used where the value is not required.

4.2 Derived Types

Beside the basic types, there are derived types constructed from the fundamental types in the following ways:

arrays of objects of a given type;
functions returning objects of a given type;
structures containing a sequence of objects of various types, with optional *asserts* of conditional checks on objects of various types.

5.0 Expressions

In general, the GAWK language follows the same conventions as C in terms of grouping reading the expression. Expressions include primary expressions, postfix expressions, array references, function calls, structure references, and operators.

Primary Expressions

Constants, strings, and identifiers are primary expressions, along with expressions in parentheses.

Arithmetic operator: + calculates the sum of the operands.

Difference operator: - is the difference of the operands.

Multiplicative operators: These include *, /, and %, which denote multiplication, division, and modulo operators. Multiplicative operators are grouped from left to right.

Postfix Expressions: Postfix expressions are grouped from left to right.

Array References: Indexes are indicated between brackets, with its name before it processed in postfix manner. Elements of an index can be accessed in the form

```
foo[idx]
```

where `foo` is an array identifier, and `idx` is the index of the element to be accessed. The type returned is the type of the array.

Function Calls: Function calls are postfix expressions constructed with a designator (the name of function followed by a pair of parentheses () . Expressions within the parentheses serve as placeholders for arguments of each function, separated by commas. Example: function(arg1, arg2)

Structure References: Structure references are accessed using dot, in the form

```
foo.bar
```

where `foo` is an identifier of a struct and `bar` is a member of `foo`. The type returned is the type of the member.

5.1 Logical Negation Operator

The operand of the ! operator is unary operator and must be applied to a boolean expression. The result is true if the value of its operand is false, and false otherwise. The type of the result is `int`.

5.2 Relational Operators

The relation operators group left-to-right and evaluates to either true or false.

relation-expression:
relational-expression < *shift-expression*
relational-expression > *shift-expression*
relational-expression <= *shift-expression*
relational-expression >= *shift-expression*

The operators < (less), > (greater), <= (less or equal), >= (greater or equal), all yield false if the relation is false, and yield true if the relation is true. The type of the result is `bool`.

5.3 Equality Operators

equality-expression:
relational-expression
equality-expression == *relational-expression*
equality-expression != *relational-expression*

The == (equal to) and the != (not equal to) operators are analogous to the relational operators except for their lower precedence (e.g. `a < b == c < d` is true whenever `a < b` and `c < d` have the same truth-value).

5.4 Logical AND Operator
There is only one form of the logical AND operator.
logical-AND-expression:
expression & expression

Logical AND groups left-to-right. It returns true if the left and right boolean expressions both evaluate to true. Otherwise it returns false.The operands need to be of boolean type. The result is `bool`.

5.5 Logical OR Operator
There is only one form of the logical OR operator.

logical-OR-expression:
expression | *expression***:**

Logical OR groups left-to-right. It returns true if either of the left and right boolean expressions evaluate to true. Otherwise it returns false.The operands need to be of arithmetic or boolean type. The result is `bool`.

5.6 Assignment Expressions

There are several assignment operators; all group from left-to-right.

assignment-expression:
conditional-expression
unary-expression = *assignment-expression*

All require an lvalue as a left operand, where the lvalue must be modifiable. The type of an assignment expression is the type of its left operand, and the value is the value stored in the left operand after the assignment is executed.

In the assignment =, the value of the left operand is replaced by the expression to the right of the assignment operator. Both operands must have the same arithmetic type.

5.7 Comma Operator

The comma's function is to separate elements of a formal list of arguments (in a function declaration or call) and in an list of actual arguments.

formal args:

```
type id(type formal_arg1, type formal_arg2) {return NULL;}
id(formal_arg1, formal_arg2);
```

actual args:

```
struct type id = {actual_arg1, actual_arg2, actual_arg3}
```

6.0 Declarations

Declarations specify the interpretation give to each identifier. Declarations have the form

declaration:
type-specifier identifier;

Empty declarations are not permitted.

6.1 Type Specifiers

The type-specifiers are

type-specifier:
`void`
`null`
`string`
`int`
`bool`
`struct`

Each declaration must have one type-specifier.

6.2 Declarators

Declarators have the generic form:

```
T D;
```

where `T` is a "*type-modifier*" and `D` is a "*idenitifer*."

6.3 Meaning of Declarators

A list of declarators appears after a sequence of type and storage class specifiers. Each declarator declares a unique main identifier. The storage class applies directly to this identifier, but its type depends on the form of its declarator. When an identifier appears in an expression of the same for as the declarator, is will give an object of the specified type.

6.3.1 Array Declarators

Arrays can be declared in two ways.

In a declaration `T D` where `D` has the form

`D1 [`*constant-expression*$_{opt}$`]`

or

`T[] D = {`*actuals_list*$_{opt}$`}`

where the type of identifier in the declaration `T D1` is "*type-modifier* `T`," the type of the identifier `D` of is "*type-modifier* array of `T`." The constant-expression must have an integral type and a value greater than 0. An array can be constructed of an integer, array, bool, strings, or structs.

6.3.2 Function Declarators

Function declarations take the form:

return_type ID (formal_args_list){ return_stmt };

The value returned much match the return_type and is optional in the event that the function is declared with return type 'void'.

6.3.3 Assert Declarations

Every time there is a change in value or property of an struct, then the struct's properties will be checked against a series of assertions optionally declared in the struct. Failure to complete these assertions will result in the execution of a predefined block of code.

Assert statements and blocks are denoted by an @ character in the beginning of the line followed by a block {}.

```
struct Player {
      int hp = 100;
      int size = 100;
      int weight
      /* this will print "Not enough HP" at runtime */
      @(hp > 1000) { print("Not enough HP"); }
      /* this should pass since size 100 > 10 */
      @(size > 10) { print("Not big enough"); }
}
```

6.3.4 Structure Declarations

A program maintains a list of declared structures. A `struct` is an object consisting of a sequence of named members with optional assertions on the aforementioned members.

struct_declaration:
STRUCT ID [struct_body]
{ { struct_name = $2;
sbody = List.rev $4 } }

struct_body:
/ nothing */ { [] }*
struct_body vdecl { S_Varialbe_Decl($2) :: $1 }
struct_body ASSERT (expr) stmt { Assert($4, $6) :: $1}

An expression can describe a struct initialization as follows:

*Struct_initialization string * string * expr list*

Members of a `struct` are accessed by a single accessor symbol '.' .
For example:

```
struct test {
      int mem;
};
struct test s;
s.mem = 10;       /* mem is now 10 */
```

Members can also have their values assigned at initialization time as follows:

```
struct test {
      int mem;
```

```
        int mem2;
};
struct test s = {10, 11}; /* mem is now 10, mem2 is now 11 */
```

6.4 Initialization

There are three types of initializations for variables, arrays and structures.

6.4.1 Variables

When an object is declared, its init may specify an initial value (in the form of an expression) for the identifier being declared.

variable:
the_type ID;
the_type ID expr;

6.4.2 Arrays

Arrays may be initialized by first stating the identifier followed by left and right brackets, an assign operator '=' and a block of comma separated expressions enclosed within left and right curly braces.

array:
ID [ ] = block;

6.4.3 Structures

Structures are initialized similarly to arrays.

structure:
struct ID ID_2 = block;

8.0 Statements

Unless otherwise specified, statement execution is sequential. Statements are executed for their effect and do not contain values. They fall into several groups.

statement:
expression-statement
compound-statement
selection-statement
iteration-statement
jump-statement

8.1 Expression Statements

Most states are expression states, and appear in the form

expression-statement:
    expression$_{opt}$;

Most expression statements are assignments or function calls. All side effects from the expression are completed and evaluated before the next statement is executed. If the expression is missing, the construction is a null statement.

8.2 Compound Statements

To allow for use of several statements where only one is expected, the compound statement, or "block," is provided. The body of a function definition, as is the body of a structure definition, is a compound statement.

compound-statement:
    { *vdeclaration-list*$_{opt}$ statement-list$_{opt}$ }

vdeclaration-list:
    vdeclaration
    vdeclaration-list vdeclaration
statement-list:
    statement
    statement-list statement
    statement-list initialization

If an identifier in the declaration-list was in scope outside the block, the outer declaration is suspended inside the block. An identifier must be unique and declared once inside the block.

Initialization of objects is performed each time the block is entered at the top, and proceeds in the order of declarators. If a jump into the block is executed, the initializations are not performed.

8.3 Selection Statements

Selection statements have several flows of control.

selection-statement:
    `if` (*expression*) *statement*
    `if` (*expression*) *statement* `else` *statement*

In each form of the `if` statements, the expression must be of a boolean type. It is evaluated and includes all side effects, and if the expression evaluates to true, the first substatement is executed. If the `if` statement is followed by an `else`, the second substatement is executed if the expression evaluates to false. The `else` ambiguity is resolved by attaching the else with the last seen `else`-less `if` statement at the same block nesting level.

8.4 Iteration Statements

Iteration statements specify looping.

iteration-statement:
`while` (*expression*) *statement*
`for` (*expression*$_{opt}$; *expression*$_{opt}$; *expression*$_{opt}$) *statement*

In the `while` statement, the substatement is executed repeatedly provided the value of the expression evaluates to true. The boolean test, including all side effects from the expression, occurs before each execution statement body.

In the `for` statement, the first expression is evaluated once, which specifies initialization for the loop. The second expression must be either of boolean type or omitted. It is evaluated before each iteration, and if it evaluates to false, the `for` is terminated. The third expression is evaluated after each iteration, and thus specifies a reinitialization for the loop. There is no restriction on its type. Side-effects from each expression are completed immediately after its evaluation. A `for` statement must include all three expressions.

8.5 Jump Statements
Jump statements transfer control unconditionally.

jump-statement:
`return` *expression*;

A function recalls to its caller by the `return` statement. When `return` is followed by an expression, the value from the expression is returned to the caller of the function and must be of the same type specified by the function. If there is no expression after `return`, the value returned is undefined.

9.0 Lexical Scope
Identifiers fall into several name spaces that do not interfere with one another based on where they're declared.

Members of structures are uniquely namespaced if and only if their container structure bears a unique name in the global structure list. If a variable is declared globally (outside of a blocked section) it is not referred to from within a block of code when there's a local variable with the same name.

```
int a;
int func(){
      int a;
      a = 10; /* does not refer to global var */
}
```

10.0 Grammar

```
program:
        /* nothing */     { [], [], [] }
        | program vdecl { let (str, var, func) = $1 in str, var, $2::func } /* int world = 4; */
        | program fdecl { let (str, var, func) = $1 in str, $2::var, func }
        | program sdecl { let (str, var, func) = $1 in $2::str, var, func }

fdecl:
        the_type ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
        { { fname   = $2;
                formals = $4;
                locals  = List.rev $7;
                body    = List.rev $8 } }

formals_opt:
        /* nothing */                   { [] }
        | formal_list                   { List.rev $1 }

formal_list:
        ID                              { [$1] }
        | formal_list COMMA ID          { $3 :: $1 }

vdecl_list:
        /* nothing */                   { [] }
        | vdecl_list vdecl              { $2 :: $1 }

vdecl:
        the_type ID SEMI                { Variable($1, $2) }
        | the_type ID expr SEMI         { Variable_Initiation($1, $2, $3)}

sdecl:
        STRUCT ID LBRACK struct_body RBRACK
        { { sname = $2;
                sbody = List.rev $4 } }

struct_body:
        /* nothing */ { [] }
        | struct_body vdecl                                     { S_Varialbe_Decl($2) :: $1 }
        | struct_body ASSERT LPAREN expr RPAREN stmt            { Assert($4, $6) :: $1}

the_type:
        INT { Int }
        | STRING                        { String }
        | BOOL                          { Boolean }
```

```
        | STRUCT ID                                   { Struct($2) }
        | the_type LBRACK expr RBRACK                 { Array($1, $3) }

stmt_list:
        /* nothing */               { [] }
        | stmt_list stmt            { $2 :: $1 }

stmt:
        expr SEMI                                                   { Expr($1) }
        | RETURN expr SEMI                                          { Return($2) }
        | LBRACE stmt_list RBRACE                                   { Block(List.rev $2) }
        | IF LPAREN expr RPAREN stmt %prec NOELSE                   { If($3, $5, Block([])) }
        | IF LPAREN expr RPAREN stmt ELSE stmt                      { If($3, $5, $7) }
        | FOR LPAREN expr_opt SEMI expr_opt SEMI expr_opt RPAREN stmt
                                                                    { For($3, $5, $7, $9) }
        | WHILE LPAREN expr RPAREN stmt                             { While($3, $5) }

expr_opt:
        /* nothing */                                   { Noexpr }
        | expr                                          { $1 }


expr:
        ID                                              { Id($1) }
        | INT_LITERAL                                   { Int_literal($1) }
        | STRING_LITERAL                                { String_literal($1) }
        | BOOL_LITERAL                                  { Bool_literal($1) }
        | THIS                                          { This }
        | NULL                                          { Null }
        | expr PLUS expr                                { Binop($1, Add, $3) }
        | expr MINUS expr                               { Binop($1, Sub, $3) }
        | expr TIMES expr                               { Binop($1, Mult, $3) }
        | expr DIVIDE expr                              { Binop($1, Div, $3) }
        | expr MOD expr                                 { Binop($1, Mod, $3) }
        | expr EQ expr                                  { Binop($1, Equal, $3) }
        | expr NEQ expr                                 { Binop($1, Neq, $3) }
        | expr LT expr                                  { Binop($1, Less, $3) }
        | expr LEQ expr                                         { Binop($1, Leq, $3) }
        | expr GT expr                                          { Binop($1, Greater, $3) }
        | expr GEQ expr                                         { Binop($1, Geq, $3) }
        | expr OR expr                                          { Binop ($1, Or, $3) }
        | expr AND expr                                         { Binop ($1, And, $3) }
        | expr ACCESS expr                                      { Access ($1, $3) }
        | expr ASSERT expr                                      { Assert ($1, $3) }
```

```
        | ID ASSIGN expr                                          { Assign ($1, $3) }
        | ID LPAREN actuals_opt RPAREN                          { Call ($1, $3) }
        | STRUCT ID ID LBRACK actuals_opt RBRACK SEMI  { Struct_initialization($2, $3, $5) }
        | LPAREN expr RPAREN                                    { $2 }
        | ID LBRACK expr RBRACK                                 { Array_access($1, $3)}

actuals_opt:
        /* nothing */     { [] }
        | actuals_list    { List.rev $1 }

actuals_list:
        expr                            { [$1] }
        | actuals_list COMMA expr       { $3 :: $1 }
```