IEEE P1003.2 Draft 11.2 – September 1991

Copyright © 1991 by the
Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street
New York, NY 10017, USA
All rights reserved as an unpublished work.

This is an unapproved and unpublished IEEE Standards Draft, subject to change. The publication, distribution, or copying of this draft, as well as all derivative works based on this draft, is expressly prohibited except as set forth below.

Permission is hereby granted for IEEE Standards Committee participants to reproduce this document for purposes of IEEE standardization activities only, and subject to the restrictions contained herein.

Permission is hereby also granted for member bodies and technical committees of ISO and IEC to reproduce this document for purposes of developing a national position, subject to the restrictions contained herein.

Permission is hereby also granted to the preceding entities to make limited copies of this document in an electronic form only for the stated activities.

The following restrictions apply to reproducing or transmitting the document in any form: 1) all copies or portions thereof must identify the document's IEEE project number and draft number, and must be accompanied by this entire notice in a prominent location; 2) no portion of this document may be redistributed in any modified or abridged form without the prior approval of the IEEE Standards Department.

Other entities seeking permission to reproduce this document, or any portion thereof, for standardization or other activities, must contact the IEEE Standards Department for the appropriate license.

Use of information contained in this unapproved draft is at your own risk.

IEEE Standards Department
Copyright and Permissions
445 Hoes Lane, P.O. Box 1331
Piscataway, NJ 08855-1331, USA
+1 (908) 562-3800
+1 (908) 562-1571 [FAX]

the exit status of the command.

All forms of `getline` shall return 1 for successful input, zero for end of file, and −1 for an error.

4.1.7.6.2.4 User-Defined Functions

The `awk` language also shall provide user-defined functions. Such functions can be defined as:

`function` *name* `(`*args*`,...) {` *statements* `}`

A function can be referred to anywhere in an `awk` program; in particular, its use can precede its definition. The scope of a function shall be global.

Function arguments can be either scalars or arrays; the behavior is undefined if an array name is passed as an argument that the function uses as a scalar, or if a scalar expression is passed as an argument that the function uses as an array. Function arguments shall be passed by value if scalar and by reference if array name. Argument names shall be local to the function; all other variable names shall be global. The same name shall not be used as both an argument name and as the name of a function or a special `awk` variable. The same name shall not be used both as a variable name with global scope and as the name of a function. The same name shall not be used within the same scope both as a scalar variable and as an array.

The number of parameters in the function definition need not match the number of parameters in the function call. Excess formal parameters can be used as local variables. If fewer arguments are supplied in a function call than are in the function definition, the extra parameters that are used in the function body as scalars shall be initialized with a string value of the null string and a numeric value of zero, and the extra parameters that are used in the function body as arrays shall be initialized as empty arrays. If more arguments are supplied in a function call than are in the function definition, the behavior is undefined.

When invoking a function, no white space can be placed between the function name and the opening parenthesis. The implementation shall permit function calls to be nested, and for recursive calls to be made upon functions. Upon return from any nested or recursive function call, the values of all of the calling function's parameters shall be unchanged, except for array parameters passed by reference. The `return` statement can be used to return a value. If a `return` statement appears outside of a function definition, the behavior is undefined.

In the function definition, `<newline>`s shall be optional before the opening brace and after the closing brace. Function definitions can appear anywhere in the program where a *pattern-action* pair is allowed.

4.1.7.7 `awk` Grammar

The grammar in this subclause and the lexical conventions in the following subclause shall together describe the syntax for `awk` programs. The general conventions for this style of grammar are described in 2.1.2. A valid program can be

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

represented as the nonterminal symbol *program* in the grammar. Any discrepancies found between this grammar and other descriptions in this clause shall be resolved in favor of this grammar.

```
%token  NAME NUMBER STRING ERE NEWLINE
%token  FUNC_NAME     /* name followed by '(' without white space */

        /* Keywords */
%token   Begin   End
/*      'BEGIN' 'END' */

%token   Break   Continue   Delete   Do   Else
/*      'break' 'continue' 'delete' 'do' 'else' */

%token   Exit   For   Function   If   In
/*      'exit' 'for' 'function' 'if' 'in' */

%token   Next   Print   Printf   Return   While
/*      'next' 'print' 'printf' 'return' 'while' */

        /* Reserved function names */
%token  BUILTIN_FUNC_NAME      /* one token for the following:
        * atan2 cos sin exp log sqrt int rand srand
        * gsub index length match split sprintf sub substr
        * tolower toupper close system
        */
%token  GETLINE      /* Syntactically different from other built-ins */

        /* Two-character tokens */
%token  ADD_ASSIGN SUB_ASSIGN MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN POW_ASSIGN
/*        '+='       '-='       '*='       '/='       '%='       '^=' */

%token   OR  AND  NO_MATCH EQ   LE   GE   NE  INCR DECR APPEND
/*      '||' '&&' '!~'    '==' '<=' '>=' '!=' '++' '--' '>>' */

        /* One-character tokens */
%token  '{' '}' '(' ')' '[' ']' ',' ';'
%token  '+' '-' '*' '%' '^' '!' '>' '<' '|' '?' ':' '~' '$' '='

%start  program
%%


program:
        item_list
      | actionless_item_list
      ;

item_list:
        newline_opt
      | actionless_item_list item terminator
      | item_list item terminator
      | item_list action terminator
      ;

actionless_item_list:
        item_list pattern terminator
      | actionless_item_list pattern terminator
      ;

item:
        pattern action
```

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```
796         | Function NAME '(' param_list_opt ')' newline_opt action
797         | Function FUNC_NAME '(' param_list_opt ')' newline_opt action
798         ;
799     param_list_opt:
800           /* empty */
801         | param_list
802         ;
803     param_list:
804           NAME
805         | param_list ',' NAME
806         ;
807     pattern:
808           Begin
809         | End
810         | expr
811         | expr ',' newline_opt expr
812         ;
813     action:
814           '{' newline_opt '}'
815         | '{' newline_opt terminated_statement_list '}'
816         | '{' newline_opt unterminated_statement_list '}'
817         ;
818     terminator:
819           ';'
820         | NEWLINE
821         | terminator NEWLINE ';'                                                    2
822         ;
823     terminated_statement_list:
824           terminated_statement
825         | terminated_statement_list terminated_statement
826         ;
827     unterminated_statement_list:
828           unterminated_statement
829         | terminated_statement_list unterminated_statement
830         ;
831     terminated_statement:
832           action newline_opt
833         | If '(' expr ')' newline_opt terminated_statement
834           Else newline_opt terminated_statement
835         | While '(' expr ')' newline_opt terminated_statement
836         | For '(' simple_statement_opt ';' expr_opt ';' simple_statement_opt ')'
837           newline_opt terminated_statement
838         | For '(' NAME In NAME ')' newline_opt terminated_statement
839         | ';' newline_opt
840         | terminatable_statement NEWLINE newline_opt
841         | terminatable_statement ';' newline_opt
842         ;
843     unterminated_statement:
844           terminatable_statement
845         | If '(' expr ')' newline_opt unterminated_statement
846         | If '(' expr ')' newline_opt terminated_statement
847           Else newline_opt unterminated_statement
```

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```
848         | While '(' expr ')' newline_opt unterminated_statement
849         | For '(' simple_statement_opt ';' expr_opt ';' simple_statement_opt ')'
850           newline_opt unterminated_statement
851         | For '(' NAME In NAME ')' newline_opt unterminated_statement
852         ;
853     terminatable_statement:
854           simple_statement
855         | Break
856         | Continue
857         | Next
858         | Exit expr_opt
859         | Return expr_opt
860         | Do newline_opt terminated_statement While '(' expr ')'
861         ;
862     simple_statement_opt:
863           /* empty */
864         | simple_statement
865         ;
866     simple_statement:
867           Delete NAME '[' expr_list ']'
868         | expr
869         | print_statement
870         ;
871     print_statement:
872           simple_print_statement
873         | simple_print_statement output_redirection
874         ;
875     simple_print_statement:
876           Print print_expr_list_opt
877         | Print '(' multiple_expr_list ')'
878         | Printf print_expr_list
879         | Printf '(' multiple_expr_list ')'
880         ;
881     output_redirection:
882           '>' expr
883         | APPEND expr
884         | '|' expr
885         ;
886     expr_list_opt:
887           /* empty */
888         | expr_list
889         ;
890     expr_list:
891           expr
892         | multiple_expr_list
893         ;
894     multiple_expr_list:
895           expr ',' newline_opt expr
896         | multiple_expr_list ',' newline_opt expr
897         ;
898     expr_opt:
899           /* empty */
```

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```
900         | expr
901         ;
902     expr:
903           unary_expr
904         | non_unary_expr
905         ;
906     unary_expr:
907           '+' expr
908         | '-' expr
909         | unary_expr '^' expr
910         | unary_expr '*' expr
911         | unary_expr '/' expr
912         | unary_expr '%' expr
913         | unary_expr '+' expr
914         | unary_expr '-' expr
915         | unary_expr non_unary_expr
916         | unary_expr '<' expr
917         | unary_expr LE expr
918         | unary_expr NE expr
919         | unary_expr EQ expr
920         | unary_expr '>' expr
921         | unary_expr GE expr
922         | unary_expr '~' expr
923         | unary_expr NO_MATCH expr
924         | unary_expr In NAME
925         | unary_expr AND newline_opt expr
926         | unary_expr OR newline_opt expr
927         | unary_expr '?' expr ':' expr
928         | unary_input_function
929         ;
930     non_unary_expr:
931           '(' expr ')'
932         | '!' expr
933         | non_unary_expr '^' expr
934         | non_unary_expr '*' expr
935         | non_unary_expr '/' expr
936         | non_unary_expr '%' expr
937         | non_unary_expr '+' expr
938         | non_unary_expr '-' expr
939         | non_unary_expr non_unary_expr
940         | non_unary_expr '<' expr
941         | non_unary_expr LE expr
942         | non_unary_expr NE expr
943         | non_unary_expr EQ expr
944         | non_unary_expr '>' expr
945         | non_unary_expr GE expr
946         | non_unary_expr '~' expr
947         | non_unary_expr NO_MATCH expr
948         | non_unary_expr In NAME
949         | '(' multiple_expr_list ')' In NAME
950         | non_unary_expr AND newline_opt expr
951         | non_unary_expr OR newline_opt expr
952         | non_unary_expr '?' expr ':' expr
953         | NUMBER
954         | STRING
```

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```
955         | lvalue
956         | ERE
957         | lvalue INCR
958         | lvalue DECR
959         | INCR lvalue
960         | DECR lvalue
961         | lvalue POW_ASSIGN expr
962         | lvalue MOD_ASSIGN expr
963         | lvalue MUL_ASSIGN expr
964         | lvalue DIV_ASSIGN expr
965         | lvalue ADD_ASSIGN expr
966         | lvalue SUB_ASSIGN expr
967         | lvalue '=' expr
968         | FUNC_NAME '(' expr_list_opt ')'  /* no white space allowed */
969         | BUILTIN_FUNC_NAME '(' expr_list_opt ')'
970         | BUILTIN_FUNC_NAME
971         | non_unary_input_function
972         ;
973     print_expr_list_opt:
974           /* empty */
975         | print_expr_list
976         ;
977     print_expr_list:
978           print_expr
979         | print_expr_list ',' newline_opt print_expr
980         ;
981     print_expr:
982           unary_print_expr
983         | non_unary_print_expr
984         ;
985     unary_print_expr:
986           '+' print_expr
987         | '-' print_expr
988         | unary_print_expr '^' print_expr
989         | unary_print_expr '*' print_expr
990         | unary_print_expr '/' print_expr
991         | unary_print_expr '%' print_expr
992         | unary_print_expr '+' print_expr
993         | unary_print_expr '-' print_expr
994         | unary_print_expr non_unary_print_expr
995         | unary_print_expr '~' print_expr
996         | unary_print_expr NO_MATCH print_expr
997         | unary_print_expr In NAME
998         | unary_print_expr AND newline_opt print_expr
999         | unary_print_expr OR newline_opt print_expr
1000        | unary_print_expr '?' print_expr ':' print_expr
1001        ;
1002    non_unary_print_expr:
1003          '(' expr ')'
1004        | '!' print_expr
1005        | non_unary_print_expr '^' print_expr
1006        | non_unary_print_expr '*' print_expr
1007        | non_unary_print_expr '/' print_expr
1008        | non_unary_print_expr '%' print_expr
```

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```
      | non_unary_print_expr '+' print_expr
      | non_unary_print_expr '-' print_expr
      | non_unary_print_expr non_unary_print_expr
      | non_unary_print_expr '~' print_expr
      | non_unary_print_expr NO_MATCH print_expr
      | non_unary_print_expr In NAME
      | '(' multiple_expr_list ')' In NAME
      | non_unary_print_expr AND newline_opt print_expr
      | non_unary_print_expr OR newline_opt print_expr
      | non_unary_print_expr '?' print_expr ':' print_expr
      | NUMBER
      | STRING
      | lvalue
      | ERE
      | lvalue INCR
      | lvalue DECR
      | INCR lvalue
      | DECR lvalue
      | lvalue POW_ASSIGN print_expr
      | lvalue MOD_ASSIGN print_expr
      | lvalue MUL_ASSIGN print_expr
      | lvalue DIV_ASSIGN print_expr
      | lvalue ADD_ASSIGN print_expr
      | lvalue SUB_ASSIGN print_expr
      | lvalue '=' print_expr
      | FUNC_NAME '(' expr_list_opt ')' /* no white space allowed */
      | BUILTIN_FUNC_NAME '(' expr_list_opt ')'
      | BUILTIN_FUNC_NAME
      ;

lvalue:
        NAME
      | NAME '[' expr_list ']'
      | '$' expr
      ;

non_unary_input_function:
        simple_get
      | simple_get '<' expr
      | non_unary_expr '|' simple_get
      ;

unary_input_function:
        unary_expr '|' simple_get
      ;

simple_get:
        GETLINE
      | GETLINE lvalue
      ;

newline_opt:
        /* empty */
      | newline_opt NEWLINE
      ;
```

This grammar has several ambiguities that shall be resolved as follows:

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

— Operator precedence and associativity shall be as described in Table 4-1.

— In case of ambiguity, an `else` shall be associated with the most immediately preceding `if` that would satisfy the grammar.

4.1.7.8 `awk` Lexical Conventions

The lexical conventions for `awk` programs, with respect to the preceding grammar, shall be as follows:

(1) Except as noted, `awk` shall recognize the longest possible token or delimiter beginning at a given point.

(2) A comment shall consist of any characters beginning with the number sign character and terminated by, but excluding the next occurrence of, a `<newline>` character. Comments shall have no effect, except to delimit lexical tokens.

(3) The character `<newline>` shall be recognized as the token `NEWLINE`.

(4) A backslash character immediately followed by a `<newline>` character shall have no effect.

(5) The token `STRING` shall represent a string constant. A string constant shall begin with the character `"`. Within a string constant, a backslash character shall be considered to begin an escape sequence as specified in Table 2-15 (see 2.12). In addition, the escape sequences in Table 4-2 shall be recognized. A `<newline>` character shall not occur within a string constant. A string constant shall be terminated by the first unescaped occurrence of the character `"` after the one that begins the string constant. The value of the string shall be the sequence of all unescaped characters and values of escape sequences between, but not including, the two delimiting `"` characters.

(6) The token `ERE` represents an extended regular expression constant. An ERE constant shall begin with the slash character. Within an ERE constant, a `<backslash>` character shall be considered to begin an escape sequence as specified in Table 2-15 (see 2.12). In addition, the escape sequences in Table 4-2 shall be recognized. A `<newline>` character shall not occur within an ERE constant. An ERE constant shall be terminated by the first unescaped occurrence of the slash character after the one that begins the string constant. The extended regular expression represented by the ERE constant shall be the sequence of all unescaped characters and values of escape sequences between, but not including, the two delimiting slash characters.

(7) A `<blank>` shall have no effect, except to delimit lexical tokens or within `STRING` or `ERE` tokens.

(8) The token `NUMBER` shall represent a numeric constant. Its form and numeric value shall be equivalent to the either of the tokens `floating-constant` or `integer-constant` as specified by the C Standard {7}, with the following exceptions:

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Table 4-2 – awk Escape Sequences

Escape Sequence	Description	Meaning
\"	<backslash> <quotation-mark>	<quotation-mark> character
\/	<backslash> <slash>	<slash> character
ddd	<backslash> followed by the longest sequence of one, two, or three octal-digit characters (01234567). If all of the digits are 0, (i.e., representation of the NUL character), the behavior is undefined.	The character whose encoding is represented by the one-, two-, or three-digit octal integer. If the size of a byte on the system is greater than nine bits, the valid escape sequence used to represent a byte is implementation defined. Multibyte characters require multiple, concatenated escape sequences of this type, including the leading \ for each byte.
c	<backslash> followed by any character not described in this table or in Table 2-15	Undefined

(a) An integer constant cannot begin with `0x` or include the hexadecimal digits `a`, `b`, `c`, `d`, `e`, `f`, `A`, `B`, `C`, `D`, `E`, or `F`.

(b) The value of an integer constant beginning with `0` shall be taken in decimal rather than octal.

(c) An integer constant cannot include a suffix (`u`, `U`, `l`, or `L`).

(d) A floating constant cannot include a suffix (`f`, `F`, `l`, or `L`).

If the value is too large or too small to be representable (see 2.9.2.1), the behavior is undefined.

(9) A sequence of underscores, digits, and alphabetics from the portable character set (see 2.4), beginning with an underscore or alphabetic, shall be considered a word.

(10) The following words are keywords that shall be recognized as individual tokens; the name of the token is the same as the keyword:

```
BEGIN     delete   for      in     printf
END       do       function next   return
break     else     getline  print  while
continue  exit     if
```

(11) The following words are names of built-in functions and shall be recognized as the token `BUILTIN_FUNC_NAME`:

```
atan2   index   match   sprintf   substr
close   int     rand    sqrt      system
```

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```
cos     length  sin     srand     tolower
exp     log     split   sub       toupper
gsub
```

The above-listed keywords and names of built-in functions are considered reserved words.

(12) The token NAME shall consist of a word that is not a keyword or a name of a built-in function and is not followed immediately (without any delimiters) by the (character.

(13) The token FUNC_NAME shall consist of a word that is not a keyword or a name of a built-in function, followed immediately (without any delimiters) by the (character. The (character shall not be included as part of the token.

(14) The following two-character sequences shall be recognized as the named tokens:

Token Name	Sequence	Token Name	Sequence
ADD_ASSIGN	+=	NO_MATCH	!~
SUB_ASSIGN	-=	EQ	==
MUL_ASSIGN	*=	LE	<=
DIV_ASSIGN	/=	GE	>=
MOD_ASSIGN	%=	NE	!=
POW_ASSIGN	^=	INCR	++
OR	||	DECR	--
AND	&&	APPEND	>>

(15) The following single characters shall be recognized as tokens whose names are the character:

```
<newline> { } ( ) [ ] , ; + - * % ^ ! > < | ? : ~ $ =
```

There is a lexical ambiguity between the token ERE and the tokens / and DIV_ASSIGN. When an input sequence begins with a slash character in any syntactic context where the token / or DIV_ASSIGN could appear as the next token in a valid program, the longer of those two tokens that can be recognized shall be recognized. In any other syntactic context where the token ERE could appear as the next token in a valid program, the token ERE shall be recognized.

4.1.8 Exit Status

The awk utility shall exit with one of the following values:

0 All input files were processed successfully.

>0 An error occurred.

The exit status can be altered within the program by using an exit expression.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

4.1.9 Consequences of Errors

If any *file* operand is specified and the named file cannot be accessed, `awk` shall write a diagnostic message to standard error and terminate without any further action.

If the program specified by either the *program* operand or the *progfile* operand(s) is not a valid `awk` program (as specified in 4.1.7), the behavior is undefined.

4.1.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

The `awk` program specified in the command line is most easily specified within single-quotes (e.g., *'program '*) for applications using `sh`, because `awk` programs commonly contain characters that are special to the shell, including double-quotes. In the cases where an `awk` program contains single-quote characters, it is usually easiest to specify most of the program as strings within single-quotes concatenated by the shell with quoted single-quote characters. For example,

```
awk '/'\''/ { print "quote:", $0 }'
```

prints all lines from the standard input containing a single-quote character, prefixed with `quote:`.

The following are examples of simple `awk` programs:

(1) Write to the standard output all input lines for which field 3 is greater than 5.

```
$3 > 5
```

(2) Write every tenth line.

```
(NR % 10) == 0
```

(3) Write any line with a substring matching the regular expression.

```
/(G|D)(2[0-9][[:alpha:]]*)/
```

(4) Write any line in which the second field matches the regular expression and the fourth field does not.

```
$2 ~ /xyz/ && $4 !~ /xyz/
```

(5) Write any line in which the second field contains a backslash.

```
$2 ~ /\\/
```

(6) Write any line in which the second field contains a backslash. Note that backslash escapes are interpreted twice, once in lexical processing of the string and once in processing the regular expression.

```
$2 ~ "\\\\"
```

(7) Write the second to the last and the last field in each line. Separate the fields by a colon.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```
{OFS=":";print $(NF-1), $NF}
```

(8) Write the line number and number of fields in each line. The three strings representing the line number, the colon and the number of fields are concatenated and that string is written to standard output.

```
{print NR ":" NF}
```

(9) Write lines longer than 72 characters.

```
{length($0) > 72}
```

(10) Write first two fields in opposite order separated by the `OFS`:

```
{ print $2, $1 }
```

(11) Same, with input fields separated by comma and/or `<space>`s and `<tab>`s:

```
BEGIN   { FS = ",[ \t]*|[ \t]+" }
        { print $2, $1 }
```

(12) Add up first column, print sum and average.

```
        {s += $1 }
END     {print "sum is ", s, " average is", s/NR}
```

(13) Write fields in reverse order, one per line (many lines out for each line in):

```
{ for (i = NF; i > 0; --i) print $i }
```

(14) Write all lines between occurrences of the strings `start` and `stop`:

```
/start/, /stop/
```

(15) Write all lines whose first field is different from the previous one:

```
$1 != prev { print; prev = $1 }
```

(16) Simulate `echo`:

```
BEGIN   {
        for (i = 1; i < ARGC; ++i)
                printf "%s%s", ARGV[i], i==ARGC-1?"\n":""
}
```

(17) Write the path prefixes contained in the **PATH** environment variable, one per line:

```
BEGIN   {
        n = split (ENVIRON["PATH"], path, ":")
        for (i = 1; i <= n; ++i)
                print path[i]
}
```

(18) If there is a file named ‘‘input’’ containing page headers of the form:

```
Page #
```

and a file named ‘‘program’’ that contains:

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```
/Page/{ $2 = n++; }
{ print }
```

then the command line:

```
awk -f program n=5 input
```

will print the file ''input,'' filling in page numbers starting at 5.

The index, length, match, and substr should not be confused with similar functions in the C Standard {7}; the awk versions deal with characters, while the C Standard {7} deals with bytes.

To forestall any possible confusion, where strings are used as the name of a file or pipeline, the strings must be textually identical. The terminology ''same string value'' implies that ''equivalent strings,'' even those that differ only by <space>s, represent different files.

History of Decisions Made

This description is based on the new awk, 'nawk,'' (see *The AWK Programming Language* {B21}), which introduced a number of new features to the historical awk:

(1) New keywords: delete, do, function, return

(2) New built-in functions: atan2, cos, sin, rand, srand, gsub, sub, match, close, system

(3) New predefined variables: FNR, ARGC, ARGV, RSTART, RLENGTH, SUBSEP

(4) New expression operators: ?:, ^

(5) The FS variable and the third argument to split are now treated as extended regular expressions.

(6) The operator precedence has changed to more closely match C. Two examples of code that operate differently are:

```
while ( n /= 10 > 1) ...
if (!"wk" ~ /bwk/) ...
```

Several features have been added based on newer implementations of awk:

(1) Multiple instances of -f *progfile* are permitted.

(2) New option: -v *assignment*

(3) New predefined variable: ENVIRON

(4) New built-in functions: toupper, tolower

(5) More formatting capabilities added to printf to match the C Standard {7}.

Regular expressions have been extended somewhat from traditional implementations to make them a pure superset of Extended Regular Expressions as defined by this standard (see 2.8.4). The main extensions are internationalization

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

features and interval expressions. Traditional implementations of awk have long supported <backslash> escape sequences as an extension to regular expressions, and this extension has been retained despite inconsistency with other utilities. The number of escape sequences recognized in both regular expressions and strings has varied (generally increasing with time) among implementations. The set specified by the standard includes most sequences known to be supported by popular implementations and by the C Standard {7}. One sequence that is not supported is hexadecimal value escapes beginning with "\x". This would allow values expressed in more than 9 bits to be used within awk as in the C Standard {7}. However, because this syntax has a nondeterministic length, it does not permit the subsequent character to be a hexadecimal digit. This limitation can be worked around in the C language by the use of lexical string concatenation. In the awk language, concatenation could also be a solution for strings, but not for regular expressions (either lexical ERE tokens or strings used dynamically as regular expressions). Because of this limitation, the feature has not been added to POSIX. 2.

When a string variable is used in a context where an ERE normally appears (where the lexical token ERE is used in the grammar) the string does not contain the literal slashes.

Some versions of awk allow the form:

```
func name (args,...) { statements }
```

This has been deprecated by the language's authors, who have asked that it not be included in the standard.

Traditional implementations of awk produce an error if a next statement is executed in a BEGIN action, and cause awk to terminate if a next statement is executed in an END action. This behavior has not been documented, and it was not believed that it was necessary to standardize it.

The specification of conversions between string and numeric values is much more detailed than in the documentation of traditional implementations or in *The AWK Programming Language* {B21}. Although most of the behavior is designed to be intuitive, the details are necessary to ensure compatible behavior from different implementations. This is especially important in relational expressions, since the types of the operands determine whether a string or numeric comparison is performed. From the perspective of an application writer, it is usually sufficient to expect intuitive behavior and to force conversions (by adding zero or concatenating a null string) when the type of an expression does not obviously match what is needed. The intent has been to specify existing practice in almost all cases. The one exception is that, in traditional implementations, variables and constants maintain both string and numeric values after their original value is converted by any use. This means that referencing a variable or constant can have unexpected side effects. For example, with traditional implementations the following program:

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```
{
        a = "+2"
        b = 2
        if (NR % 2)
                c = a + b
        if (a == b)
                print "numeric comparison"
        else
                print "string comparison"
}
```

would perform a numeric comparison (and output **numeric comparison**) for each odd-numbered line, but perform a string comparison (and output **string comparison**) for each even-numbered line. POSIX.2 ensures that comparisons will be numeric if necessary. With traditional implementations, the following program:

```
BEGIN {
        OFMT = "%e"
        print 3.14
        OFMT = "%f"
        print 3.14
}
```

would output **3.140000e+00** twice, because in the second `print` statement the constant `3.14` would have a string value from the previous conversion. The standard requires that the output of the second `print` statement be **3.140000**. The behavior of traditional implementations was seen as too unintuitive and unpredictable.

However, a further modification was made in Draft 11. It was pointed out that with the Draft 10 rules, the following script would print nothing:

```
BEGIN {
        y[1.5] = 1
        OFMT = "%e"
        print y[1.5]
}
```

Therefore, a new variable, `CONVFMT`, was introduced. The `OFMT` variable is now restricted to affecting output conversions of numbers to strings and `CONVFMT` is used for internal conversions, such as comparisons or array indexing. The default value is the same as that for `OFMT`, so unless a program changes `CONVFMT` (which no historical program would do), it will receive the historical behavior associated with internal string conversions.

The POSIX `awk` lexical and syntactic conventions are specified more formally than in other sources. Again the intent has been to specify existing practice. One convention that may not be obvious from the formal grammar as in other verbal descriptions is where <newline>s are acceptable. There are several obvious placements such as terminating a statement, and a backslash can be used to escape <newline>s between any lexical tokens. In addition, <newline>s without backslashes can follow a comma, an open brace, logical AND operator

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

(&&), logical OR operator (||), the `do` keyword, the `else` keyword, and the closing parenthesis of an `if`, `for`, or `while` statement. For example:

```
{ print $1,
        $2 }
```

The requirement that `awk` add a trailing `<newline>` to the *program* argument text is to simplify the grammar, making it match a text file in form. There is no way for an application or test suite to determine whether a literal `<newline>` is added or whether `awk` simply acts as if it did.

Because the concatenation operation is represented by adjacent expressions rather than an explicit operator, it is often necessary to use parentheses to enforce the proper evaluation precedence.

The overall `awk` syntax has always been based on the C language, with a few features from the shell command language and other sources. Because of this, it is not completely compatible with any other language, which has caused confusion for some users. It is not the intent of this standard to address such issues. The standard has made a few relatively minor changes toward making the language more compatible with the C language as specified by the C Standard {7}; most of these changes are based on similar changes in recent implementations, as described above. There remain several C language conventions that are not in *awk* . One of the notable ones is the comma operator, which is commonly used to specify multiple expressions in the C language `for` statement. Also, there are various places where `awk` is more restrictive than the C language regarding the type of expression that can be used in a given context. These limitations are due to the different features that the `awk` language does provide.

This standard requires several changes from traditional implementations in order to support internationalization. Probably the most subtle of these is the use of the decimal-point character, defined by the LC_NUMERIC category of the locale, in representations of floating point numbers. This locale-specific character is used in recognizing numeric input, in converting between strings and numeric values, and in formatting output. However, regardless of locale, the period character (the decimal-point character of the POSIX Locale) is the decimal-point character recognized in processing `awk` programs (including assignments in command-line arguments). This is essentially the same convention as the one used in the C Standard {7}. The difference is that the C language includes the *setlocale* () function, which permits an application to modify its locale. Because of this capability, a C application begins executing with its locale set to the C locale, and only executes in the environment-specified locale after an explicit call to *setlocale* (). However, adding such an elaborate new feature to the `awk` language was seen as inappropriate for POSIX. 2. It is possible to explicitly execute an `awk` program in any desired locale by setting the environment in the shell.

The behavior in the case of invalid `awk` programs (including lexical, syntactic, and semantic errors) is undefined because it was considered overly limiting on implementations to specify. In most cases such errors can be expected to produce a diagnostic and a nonzero exit status. However, some implementations may choose to extend the language in ways that make use of certain invalid constructs.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Other invalid constructs might be deemed worthy of a warning but otherwise cause some reasonable behavior. Still other constructs may be very difficult to detect in some implementations. Also, different implementations might detect a given error during an initial parsing of the program (before reading any input files) while others might detect it when executing the program after reading some input. Implementors should be aware that diagnosing errors as early as possible and producing useful diagnostics can ease debugging of applications, and thus make an implementation more usable.

The unspecified behavior from using multicharacter `RS` values is to allow possible future extensions based on regular expressions used for record separators. Historical implementations take the first character of the string and ignore the others.

The undefined behavior resulting from NUL s in regular expressions allows future extensions for the GNU `gawk` program to process binary data.

Unspecified behavior when `split(string,array,<null>)` is used is to allow a proposed future extension that would split up a string into an array of individual characters.

4.2 `basename` — Return nondirectory portion of pathname

4.2.1 Synopsis

`basename` *string* [*suffix*]

4.2.2 Description

The *string* operand shall be treated as a pathname, as defined in 2.2.2.102. The string *string* shall be converted to the filename corresponding to the last pathname component in *string* and then the suffix string *suffix* , if present, shall be removed. This shall be done by performing actions equivalent to the following steps in order:

(1) If *string* is `//`, it is implementation defined whether steps (2) through (5) are skipped or processed.

(2) If *string* consists entirely of slash characters, *string* shall be set to a single slash character. In this case, skip steps (3) through (5).

(3) If there are any trailing slash characters in *string* , they shall be removed.

(4) If there are any slash characters remaining in *string* , the prefix of *string* up to and including the last slash character in *string* shall be removed.

(5) If the *suffix* operand is present, is not identical to the characters remaining in *string* , and is identical to a suffix of the characters remaining in *string* , the suffix *suffix* shall be removed from *string* . Otherwise, *string*

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

shall not be modified by this step. It shall not be considered an error if *suffix* is not found in *string* .

The resulting string shall be written to standard output.

4.2.3 Options

None.

4.2.4 Operands

The following operands shall be supported by the implementation:

string A string.

suffix A string.

4.2.5 External Influences

4.2.5.1 Standard Input

None.

4.2.5.2 Input Files

None.

4.2.5.3 Environment Variables

The following environment variables shall affect the execution of `basename`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

LC_MESSAGES This variable shall determine the language in which messages should be written.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

4.2.5.4 Asynchronous Events

Default.

4.2.6 External Effects

4.2.6.1 Standard Output

The `basename` utility shall write a line to the standard output in the following format:

"%s\n", <*resulting string*>

4.2.6.2 Standard Error

Used only for diagnostic messages.

4.2.6.3 Output Files

None.

4.2.7 Extended Description

None.

4.2.8 Exit Status

The `basename` utility shall exit with one of the following values:

0 Successful completion.

>0 An error occurred.

4.2.9 Consequences of Errors

Default.

4.2.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

If the string *string* is a valid pathname,

```
$(basename "string")
```

produces a filename that could be used to open the file named by *string* in the directory returned by

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```
$(dirname "string")
```

If the string *string* is not a valid pathname, the same algorithm is used, but the result need not be a valid filename. The `basename` utility is not expected to make any judgements about the validity of *string* as a pathname; it just follows the specified algorithm to produce a result string.

The following shell script compiles `/usr/src/cmd/cat.c` and moves the output to a file named `cat` in the current directory when invoked with the argument `/usr/src/cmd/cat` or with the argument `/usr/src/cmd/cat.c`:

```
c89 $(dirname "$1")/$(basename "$1" .c).c
mv a.out $(basename "$1" .c)
```

History of Decisions Made

The POSIX.1 {8} definition of pathname allows trailing slashes on a pathname naming a directory. Some historical implementations have not allowed trailing slashes and thus treated pathnames of this form in other ways. Existing implementations also differ in their handling of *suffix* when *suffix* matches the entire string left after removing the directory part of *string* .

The behaviors of `basename` and `dirname` in this standard have been coordinated so that when *string* is a valid pathname

```
$(basename "string")
```

would be a valid filename for the file in the directory

```
$(dirname "string")
```

This would not work for the versions of these utilities in earlier drafts due to the way it specified handling of trailing slashes.

Since the definition of *pathname* in 2.2.2.102 specifies implementation-defined behavior for pathnames starting with two slash characters, Draft 11 has been changed to specify similar implementation-defined behavior for the `basename` and `dirname` utilities. On implementations where the pathname `//` is always treated the same as the pathname `/`, the functionality required by Draft 10 meets all of the Draft 11 requirements.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.