NAME

awklex − lexically analyze awk program files

SYNOPSIS

awklex −f *awk-program-file >outfile*

DESCRIPTION

awklex lexically analyzes an **awk**(1) program file, and produces on *stdout* a token stream, one line per token. The output stream can then be further filtered by other programs, such as the one embedded in **awkpretty**(1).

The companion program, **awkunlex**(1), turns its input lexical token stream back into an **awk**(1) program.

Typical uses of these tools look like this:

```
awklex file(s) | cmds
cmds | awklex -f - | cmds
awklex file(s) | cmds | awkunlex >new-awk-file
cmds | awklex -f - | cmds | awkunlex >new-awk-file
```

OPTIONS

Although **awklex** accepts all of the command-line options recognized by **awk**(1), the only relevant option is:

−f *awk-program-file* Specify the name of the file containing the **awk**(1) program.

Multiple **−f** options can be used, if required; their files are then treated as if logically concatenated in the order that they were specified.

The filename can be a hyphen, −, meaning that input should be taken from *stdin*, allowing **awklex** to be used in the middle of a command pipeline.

LEXICAL ANALYSIS

The output stream produced by **awklex** has lines of the form

```
# line nnn "filename"
# arbitrary comment text
<token-number><tab>STRING<tab>"<token-value>"
<token-number><tab><token-type-name><tab><token-value>
```

Each output line is either a line number directive identifying the source of the original tokens, or a comment, or else contains a single complete token, identified by an integer number for use by a computer program, a token type name for human readers (and programs), and a token value.

Errors detected during the lexical and grammatical analyses produce messages on *stderr*, but unless a catastrophic error has occurred, processing will normally continue until all input has been processed, and it will not be evident from the token stream that errors have occurred. Therefore, in order to permit subsequent applications to detect such errors, and take suitable recovery action, a special error comment directive

```
# ERROR
```

is produced immediately following the line for the token at which the error was first reported.

If the type name is STRING, the value *includes* the surrounding quotes. Special characters in the token value string are represented with ANSI/ISO Standard C character and octal escape sequences, so all characters other than NUL are representable, and multi-line values can be represented in a single line. Tab characters will not appear literally: they will always be represented by the escape sequence `\t`.

It is possible for a non-STRING value to contain literal tab characters, so programs that process the **awklex** output stream must be prepared to handle that.

The special case of a newline (NL) token has an empty value string, rather than a literal newline character.

Except for token types NL and STRING, all other token values are represented literally; they are *not* converted to use Standard C escape sequences, and cannot contain newline characters. They extend from immediately after the second tab character on the line, up to, but not including, the following newline character.

Except for **gawk**(1), most **awk**(1) implementations do not properly handle NUL characters, and thus, **awklex** does not either. This is a consequence of their implementation in the C programming language using NUL-terminated C character strings to hold **awk**(1) strings. Text files, by definition, do not contain NUL characters, so this restriction is seldom of any consequence.

Here are the token numbers and token type names that can appear in the output, ordered first by increasing token number:

258 PROGRAM	281 EQ	304 NEXT	327 POSTDECR
259 PASTAT	282 GE	305 NEXTFILE	328 PREDECR
260 PASTAT2	283 GT	306 ADD	329 VAR
261 XBEGIN	284 LE	307 MINUS	330 IVAR
262 XEND	285 LT	308 MULT	331 VARNF
263 NL	286 NE	309 DIVIDE	332 CALL
264 ARRAY	287 IN	310 MOD	333 NUMBER
265 MATCH	288 ARG	311 ASSIGN	334 STRING
266 NOTMATCH	289 BLTIN	312 ASGNOP	335 REGEXPR
267 MATCHOP	290 BREAK	313 ADDEQ	336 COMMENT
268 FINAL	291 CLOSE	314 SUBEQ	337 WHITESPACE
269 DOT	292 CONTINUE	315 MULTEQ	338 GETLINE
270 ALL	293 DELETE	316 DIVEQ	339 RETURN
271 CCL	294 DO	317 MODEQ	340 SPLIT
272 NCCL	295 EXIT	318 POWEQ	341 SUBSTR
273 CHAR	296 FOR	319 PRINT	342 WHILE
274 OR	297 FUNC	320 PRINTF	343 CAT
275 STAR	298 SUB	321 SPRINTF	344 NOT
276 QUEST	299 GSUB	322 ELSE	345 UMINUS
277 PLUS	300 IF	323 INTEST	346 POWER
278 AND	301 INDEX	324 CONDEXPR	347 DECR
279 BOR	302 LSUBSTR	325 POSTINCR	348 INCR
280 APPEND	303 MATCHFCN	326 PREINCR	349 INDIRECT

and then alphabetically by token type name:

306 ADD	309 DIVIDE	265 MATCH	346 POWER
313 ADDEQ	294 DO	303 MATCHFCN	328 PREDECR
270 ALL	269 DOT	267 MATCHOP	326 PREINCR
278 AND	322 ELSE	307 MINUS	319 PRINT
280 APPEND	281 EQ	310 MOD	320 PRINTF
288 ARG	295 EXIT	317 MODEQ	258 PROGRAM
264 ARRAY	268 FINAL	308 MULT	276 QUEST
312 ASGNOP	296 FOR	315 MULTEQ	335 REGEXPR
311 ASSIGN	297 FUNC	272 NCCL	339 RETURN
289 BLTIN	282 GE	286 NE	340 SPLIT
279 BOR	338 GETLINE	304 NEXT	321 SPRINTF
290 BREAK	299 GSUB	305 NEXTFILE	275 STAR
332 CALL	283 GT	263 NL	334 STRING
343 CAT	300 IF	344 NOT	298 SUB
271 CCL	287 IN	266 NOTMATCH	314 SUBEQ
273 CHAR	348 INCR	333 NUMBER	341 SUBSTR
291 CLOSE	301 INDEX	274 OR	345 UMINUS
336 COMMENT	349 INDIRECT	259 PASTAT	329 VAR
324 CONDEXPR	323 INTEST	260 PASTAT2	331 VARNF
292 CONTINUE	330 IVAR	277 PLUS	342 WHILE
347 DECR	284 LE	327 POSTDECR	337 WHITESPACE
293 DELETE	302 LSUBSTR	325 POSTINCR	261 XBEGIN
316 DIVEQ	285 LT	318 POWEQ	262 XEND

In addition to these 92 uppercase names, token type names can also be of the form `token nnn`, where `nnn` is the decimal ASCII value of a single-character token, separated by a single space from the preceding word. The token number is also `nnn`, which is why token numbers 0 . . . 255 were unavailable for the named tokens listed above.

The token stream ends with a null token:

```
    0        token 0
```

The token numbers are generated automatically, and although **awk**(1) is quite a stable program, it is possible that they could change in future **awk**(1) versions, so programmers, beware. The token *type names* should never change, but the set of token types may grow if the **awk**(1) language is ever changed.

The following short example gives a flavor of what the lexical token stream looks like for this simple **awk**(1) program:

```
BEGIN   { initialize() }
        { print FNR ":" $0 }
END     { terminate() }

function initialize()
{
    print "Hello, world"
}

function terminate()
{
    print "Goodbye, world"
}
```

Here is the output of `awklex -f simple.awk`, displayed in two columns for compactness:

```
# line 1 "simple.awk"         337     WHITESPACE
261   XBEGIN      BEGIN       332     CALL    initialize
337   WHITESPACE        40     token 40        (
123   token 123   {      41     token 41        )
337   WHITESPACE       263     NL
332   CALL  initialize    # line 6 "simple.awk"
 40   token 40    (     123     token 123       {
 41   token 41    )     263     NL
337   WHITESPACE        # line 7 "simple.awk"
 59   token 59    }     337     WHITESPACE
125   token 125   }     319     PRINT   print
263   NL                337     WHITESPACE
# line 2 "simple.awk"         334     STRING  "Hello, world
337   WHITESPACE        337     WHITESPACE
123   token 123   {     263     NL
337   WHITESPACE        # line 8 "simple.awk"
319   PRINT print        59     token 59        }
337   WHITESPACE        125     token 125       }
329   VAR   FNR         263     NL
337   WHITESPACE        # line 10 "simple.awk"
334   STRING      ": "        263     NL
337   WHITESPACE        # line 10 "simple.awk"
349   INDIRECT    0     297     FUNC    function
333   NUMBER      0           337     WHITESPACE
337   WHITESPACE        332     CALL    terminate
 59   token 59    }      40     token 40        (
125   token 125   }      41     token 41        )
263   NL                263     NL
```

```
# line 3 "simple.awk"          # line 11 "simple.awk"
262   XEND  END         123     token 123       {
337   WHITESPACE        263     NL
123   token 123   {     # line 12 "simple.awk"
337   WHITESPACE        337     WHITESPACE
332   CALL  terminate     319     PRINT   print
 40   token 40    (     337     WHITESPACE
 41   token 41    )     334     STRING  "Goodbye, wor
337   WHITESPACE        337     WHITESPACE
 59   token 59    }     263     NL
125   token 125   }     # line 13 "simple.awk"
263   NL                 59     token 59        }
# line 5 "simple.awk"         125     token 125       }
263   NL                263     NL
# line 5 "simple.awk"               0     token 0
297   FUNC  function
```

SAMPLE APPLICATIONS

Here are some possibly useful things you can do simply with **awklex**:

check spelling:

```
awklex -f myfile.awk | grep STRING | spell
```

check for doubled words:

```
awklex -f myfile.awk | grep STRING | dw
```

count static references to each function:

```
awklex -f myfile.awk | grep CALL | sort | uniq -c
```

find functions that are defined, but never called:

```
awklex -f myfile.awk | grep CALL | sort | uniq -c | \
        awk '$1 == 1'
```

capitalize function names:

```
awklex -f myfile.awk | \
awk '$2 == "CALL"  {printf("%s\t%s\t%s\n", $1, $2, \
        toupper(substr($3,1,1)) tolower(substr($3,2)))}' | \
        awkunlex > newfile.awk
```

Because **awk** was designed for writing simple programs simply, it does not require variable declarations, and except for function arguments, variables are normally global throughout the entire program.

The standard idiom for declaring a local variable in a function is to give it as an extra function argument. Failure to do this in a large program could result in a local variable overwriting a global variable of the same name, producing a nasty bug that can be quite hard to detect by reading the code. **awk**(1) itself cannot diagnose the problem, since it is legal to do this in the language.

Here is a how you can find such problems, using just fifteen lines of **awklex** and **awk**(1):

find global variables in functions:

```
awklex -f myfile.awk | \
awk '
BEGIN       { FS = "\t"; fcn = "-OUTER-" }

($2 == "FUNC"),($3 == ")") { if ($2 == "CALL") fcn = $3; next }

$2 == "VAR" { GlobalVars[fcn,$3]++ }

END         {
                for (fv in GlobalVars)
```

```
        {
            split(fv,vars,SUBSEP)
            printf("%-31s\t%s\n", vars[1], vars[2]) | "sort"
        }
    }'
```

This program collects the names of all functions, and global variables referenced inside them, and uses them as index pairs in `GlobalVars[]` to count the number of such pairs. After all of the input stream has been processed, a sorted list is printed to display the pairs.

The `BEGIN` pattern handles initialization and defines a name for the outer level 'function'. The following range pattern matches the function header, and extracts from it the function name. The third pattern records references to variables which are not arguments, and thus, are global. The final `END` pattern handles the output.

Straightforward removal of unneeded newlines and whitespaces could reduce the above code to just five lines, but with a loss in readability.

While this example is somewhat more complex than the one-liners for which **awk**(1) is justly famous, it is surprising how little code is needed to produce a perfectly-reliable result for what would otherwise be a rather difficult job in most other programming languages, and one *impossible* to do reliably with simple pattern matching on the **awk**(1) program source text.

This example is so generally useful that it is provided via the **--globals** option in **awkpretty**(1).

SEE ALSO

awk(1), **awkpretty**(1), **awkunlex**(1), **gawk**(1), **mawk**(1), **nawk**(1).

AUTHORS

The **awk**(1) implementation on which **awklex** is based was written by, and is maintained by, Brian Kernighan of Lucent Technologies (formerly, AT&T) Bell Laboratories. The source code is freely available on the World-Wide Web at:

`http://cm.bell-labs.com/cm/cs/awkbook/index.html`

and the **awk**(1) programming language is completely described in the excellent book:

Alfred V. Aho, Brian W. Kernighan and Peter J. Weinberger
The AWK Programming Language
Addison-Wesley, 1988
ISBN 0-201-07981-X

The minor modifications (6 lines of changes, plus about 165 new lines) of the original **awk**(1) code (about 12,000 lines) to produce **awklex**, and this documentation, were done by:

Nelson H. F. Beebe
Center for Scientific Computing
University of Utah
Department of Mathematics, 322 INSCC
155 S 1400 E RM 233
Salt Lake City, UT 84112-0090
USA
Email: `beebe@math.utah.edu`, `beebe@acm.org`, `beebe@ieee.org`
WWW URL: `http://www.math.utah.edu/~beebe`
Telephone: +1 801 581 5254
FAX: +1 801 585 1640, +1 801 581 4148

It is a tribute to the great skill and care with which **awk**(1) has been implemented that **awklex** could be created with such tiny changes.

COPYRIGHT

Almost all of the code in **awklex** is derived from the **awk**(1) distribution, for which a copyright applies, and as required by that copyright, it is reproduced verbatim here in this documentation: