NAME

sgrep - search a file for a structured pattern

SYNOPSIS

sgrep [**-aCcDdhilNnPqSsTtV**] [**-O** *filename*] [**-o** "*format*"] [**-p** *preprocessor*] [**-e**] *expression* [*filename* ...]

sgrep [**-aCcDdhilNnPqSsTtV**] [**-O** *filename*] [**-o** "*format*"] [**-p** *preprocessor*] **-f** *filename* [**-e** *expression*] [*filename* ...]

sgrep **-h**

DESCRIPTION

sgrep (structured grep) is a tool for searching *text files* and filtering *text streams* using structural criteria. The data model of sgrep is based on *regions*, which are non-empty substrings of text. *Regions* are typically occurrences of constant strings or meaningful text elements, which are recognizable through some delimiting strings. Regions can be arbitrarily long, arbitrarily overlapping, and arbitrarily nested.

sgrep uses patterns called *region expressions* to express which regions of the input text are output to standard output. The selection of regions is based on mutual *containment and ordering conditions* of the regions, expressed by the region expression.

Region expressions are read by default first from file **$HOME/.sgreprc,** or if it doesn't exist, from file **/usr/lib/sgreprc,** and then from the command line. Different behavior can be specified through command line options.

Input *files* are processed one by one (i.e., regions cannot extend over file boundaries), except if the **-S** flag is given, in which case **sgrep** takes the concatenation of the input files as its input text. If no input files are given, **sgrep** reads the standard input. Standard input can also be specified as an input file by giving hyphen '-' as a file name.

The selected regions are output in increasing order of their start positions. If several output regions overlap, a minimal region that covers them all is output, by default, instead of outputting each of them separately.

OPTIONS

-a Act as a filter: display the maching regions, possibly formatted according to the output format, interleaved with the rest of the text. (See the description of option -o below.)

-C Display copyright notice.

-c Display only the count of the regions that match the expression.

-D Display verbose progress output. **NOTE:** This is used for debugging purposes only and may not function in future versions of sgrep.

-d Display each matching region once, even if the regions overlap or nest.

-e *expression*
Search the input text for occurrences of *expression*.

-f *file* Read the region expression from the named file. Filename **-** refers to stdin.

-h Display a short help.

-i Ignore case distinctions in phrases.

-l Long output format: precede each output region by a line which indicates the ordinal number of the region, the name of the file where the region starts, the length of the region in bytes, the start and end positions of the region within the entire input text, the start position of the region within the file containing the start, and the end position of the region within the file containing the end.

-N Do not add a newline after the last output region.

-n Suppress reading **$HOME/.sgreprc** or **/usr/lib/sgreprc.**

-O file Read the output format from file. See the description of output formats below.

-o format
Set the output format. The format is displayed for each output region with any occurrences of the following place holders substituted:

%f	name of the file containing the start of the region
%s	start position of the region
%e	end position of the region
%l	length of the region in bytes (i.e., **%e-%s+1**)
%i	start position of the region in the file where the region begins
%j	end position of the region in the file where the region ends
%r	text of the region. "%r" is the default output format.
%n	gets the ordinal number of the region

-P Display the (preprocessed) region expression without executing it.

-p *preprocessor*
Apply *preprocessor* to the region expression before evaluating it.

-S Stream mode. With this option sgrep considers it's input files as a continuous stream, so that regions may extend across file boundaries.

```
sgrep -S file_1 ... file_n
```

is similar to

```
cat file_1 ... file_n | sgrep
```

except that the latter creates a temporary disk file of the input stream. Sgrep may use much more memory when run with the **-S** option, since then it cannot release its internal region lists between processing each file.

-s Short output format (default): do not format the text of the output regions, and display overlapping parts of regions only once.

-T Display statistics about the execution.

-t Display time usage.

-V Display version information.

-- No more options.

A list of options can be given also as the value of the environment variable **SGREPOPT**.

SYNTAX OF EXPRESSIONS

```
region_expr ->   basic_expr
               | operator_expr

operator_expr -> region_expr ['not'] 'in' basic_expr
               | region_expr ['not'] 'containing' basic_expr
               | region_expr ['not'] 'equal' basic_expr
               | region_expr 'or' basic_expr
               | region_expr 'extracting' basic_expr
               | region_expr '..' basic_expr
               | region_expr '_.' basic_expr
               | region_expr '._' basic_expr
               | region_expr '__' basic_expr
               | region_expr 'quote' basic_expr
               | region_expr '_quote' basic_expr
```

```
                     | region_expr 'quote_' basic_expr
                     | region_expr '_quote_' basic_expr
                     | 'concat' '(' region_expr ')'
                     | 'inner' '(' region_expr ')'
                     | 'outer' '(' region_expr ')'
                     | 'join' '(' integer ',' region_expr ')'

basic_expr ->   phrase
              | 'start'
              | 'end'
              | 'chars'
              | constant_list
              | '(' region_expr ')'

phrase -> '"' char [ char ... ] '"'

constant_list -> '[' ']' | '[' regions ']'

regions ->   region
           | region regions

region -> '(' integer ',' integer ')'
```

Note that region expressions are left-associative. This means, for example, that an expression

```
'"<a>".."</a>" or "</b>"'
```

evaluates to the regions starting with `"<a>"` and ending with `"</a>"`, or comprising only the string `"</b>"`. In order to obtain the regions that begin with `"<a>"` and end with either `"</a>"` or `"</b>"`, one should indicate the proper order of evaluation using parentheses:

```
"<a>".. ("</a>" or "</b>")
```

Expressions can also contain *comments*, which start with '#' and extend to the end of the line. However, a '#'-sign in a phrase does not begin a comment.

SEMANTICS OF EXPRESSIONS

The value of an expression is a set of regions of input text that satisfy the expression.

Value **v(basic_expr)** of a basic expression:

v(phrase):=
the set of regions of input text whose text equals the text of the phrase.

v('start'):=
a set consisting of single-character regions for the first position of each input file. If the -S option is given, the value is a set containing a single region that comprises the first character in the input stream.

v('end'):=
a set consisting of single-character regions for the last position of each input file. If the -S option is given, the value is a set containing a single region that comprises the last character in the input stream.

v('chars'):=
a set consisting of all single-character regions.

v([ ]):= an empty set.

v([(s_1,e_1) (s_1,e_2) ... (s_n,e_n)]):=
a set consisting of regions **r_i** for each **i = 1,...,n,** where the start position of region **r_i** is **s_i** and its end position is **e_i.** The positions have to be nonnegative integers, and the regions have to be given in increasing order of their start positions; regions with a common start positions have to be given in increasing order of their end positions. The positions are counted from the first character of each input file, unless the -S option is given, in which case the positions are counted starting from the beginning of the input stream. The number of the first position in a file or a stream is zero.

v('('region_expr')'):= v(region_expr).

Value **v(operator_expr)** of operator expressions:

v(region_expr 'in' basic_expr):=
the set of the regions in **v(region_expr)** that are contained in some region in **v(basic_expr)**. A region **x** is contained in another region **y** if and only if the start position of **x** is greater than the start position of **y** and the end position of **x** is not greater than the end position of **y**, or the end position of **x** is smaller than the end position of **y** and the start position of **x** is not smaller than the start position of **y**.

v(region_expr 'not' 'in' basic_expr):=
the set of the regions in **v(region_expr)** that are not contained in any region in **v(basic_expr).**

v(region_expr 'containing' basic_expr):=
the set of the regions in **v(region_expr)** that contain some region in **v(basic_expr).**

v(region_expr 'not' 'containing' basic_expr):=
the set of the regions in **v(region_expr)** that do not contain any region in **v(basic_expr).**

v(region_expr 'equal' basic_expr):=
The set of regions, which occur in both **v(region_expr)** and **v(basic_expr).**

v(region_expr 'not equal' basic_expr):=
The set of regions, which occur in **v(region_expr)** but do not occur in **v(basic_expr).**

v(region_expr 'or' basic_expr):=
the set of the regions that appear in **v(region_expr)** or in **v(basic_expr)** or in both.

v(region_expr 'extracting' basic_expr):=
the set of the non-empty regions that are formed of the regions in **v(region_expr)** by extracting an overlap with any region in **v(basic_expr).** For example, the value of

```
'[(1,4) (3,6) (7,9)] extracting [(2,5) (4,7)]'
```

consists of the regions (1,1) and (8,9).

v(region_expr '..' basic_expr):
The value of this expression consists of the regions that can be formed by *pairing* regions from **v(region_expr)** with regions from **v(basic_expr).** The pairing is defined as a generalization of the way how nested parentheses are paired together "from inside out". For this we need to be able to compare the order of regions, which may be overlapping and nested. This ordering is defined as follows.

Let **x** and **y** be two regions. We say that region **x** *precedes* region **y** if the end position of **x** is smaller than the start position of **y.** We say that region **x** is *later* than region **y** if the end position of **x** is greater than the end position of **y,** or if they end at the same position and the start of **x** is greater than the start of **y.** Region **x** is *earlier* than region **y** if the start position of **x** is smaller than the start position of **y,** or if they start at the same position and the end position of **x** is less than the end position of **y.** Now a region **x** from **v(region_expr)** and a region **y** from **v(basic_expr)** are paired in expression **v(region_expr '..' basic_expr)** if and only if

1. **x** precedes **y,**

2. **x** is not paired with any region from **v(basic_expr)** which is earlier than **y,** and
3. **y** is not paired with any region from **v(region_expr)** which is later than **x.**

The pairing of regions **x** and **y** forms a region that extends from the start position of **x** to the end position of **y.**

v(region_expr '._' basic_expr):
The pairing of the regions from **v(region_expr)** and the regions from **v(basic_expr)** is defined similarly to **v(region_expr '..' basic_expr) above,** except that the pairing of regions **x** and **y** now forms a region which extends from the start position of **x** to the position immediately preceding the start of **y.**

v(region_expr '_.' basic_expr):=
The pairing of the regions from **v(region_expr)** and the regions from **v(basic_expr)** is defined similarly to **v(region_expr '..' basic_expr) above, except that** the pairing of regions **x** and **y** now forms a region which extends from the position immediately following the end position of **x** to the end position of **y.**

v(region_expr '__' basic_expr):=
The pairing of the regions from **v(region_expr)** and the regions from **v(basic_expr)** is defined similarly to **v(region_expr '..' basic_expr)** above, except that now the pairing of regions **x** and **y** forms a region which extends from the text position immediately following the end of **x** to the text position immediately preceding the start of **y**. Possibly resulting empty regions are excluded from the result.

v(region_expr 'quote' basic_expr):
The value of this expression consists of the regions that extend from the start position of a "*left-quote region*" in **v(region_expr)** to the end position of a corresponding "*right-quote region*" in **v(basic_expr)**. The regions in the result are non-nesting and non-overlapping. The left-quote regions and the right-quote regions are defined as follows:

- The earliest region (see above) in **v(region_expr)** is a *possible left-quote region.*
- For each possible left-quote region **x**, the earliest region in **v(basic_expr)** preceeded by **x** is its right-quote region.
- For each right-quote region **y** in **v(basic_expr)**, the earliest region in **v(region_expr)** preceeded by **y** is a possible left-quote region.

The below example query finds C-style non-nesting comments:

```
"/*" quote "*/"
```

The below example query finds strings between quotation marks:

```
"\"" quote "\""
```

(Notice the difference to expression `"\""` **..** `"\""`, which would evaluate to any substring of input text that starts with a quotation mark and ends with the next quotation mark.)

The variants **_quote**, **quote_** and **_quote_** are analogical to the operators **_.**, **._** and **__**, in the sense that the "quote regions" originating from the expression on the side of the underscore _ are excluded from the result regions. (In the case of **_quote_** any possibly resulting empty regions are excluded from the result.)

v('concat' '(' region_expr ')'):=
the set of the longest regions of input text that are covered by the regions in **v(region_expr).**

v('inner' '(' region_expr ')'):=
the set of regions in **v(region_expr)** that do not contain any other region in **v(region_expr).** Note that for any region expression **A**, the expression **inner(A)** is equivalent to **(A not containing A)**.

v('outer' '(' region_expr ')'):=
the set of regions in **v(region_expr)** that are not contained in any other region in **v(region_expr).** Note that for any region expression **A**, the expression **outer(A)** is equivalent to **(A not in A)**.

v('join' '(' n ',' region_expr ')'):
The value of this expression is formed by processing the regions of v(region_expr) in increasing order of their start positions (and in increasing order of end positions for regions with a common start). Each region **r** produces a result region beginning at the start of r and extending to the end of the (n-1)th region after r. The operation is useful only with non-nesting regions. Especially, when applied to 'chars', it can be used to express nearness conditions. For example,

```
'"/*" quote "*/" in join(10,chars)'
```

selects comments "/* ... */" which are at most 10 characters long.

EXAMPLES OF REGION EXPRESSIONS

Count the number of occurrences of string "sort" in file eval.c:

```
sgrep -c '"sort"' eval.c
```

Show all blocks delimited by braces in file eval.c:

```
sgrep '"{" .. "}"' eval.c
```

Show the outermost blocks that contain "sort" or "nest":

```
sgrep 'outer("{" .. "}" containing ("sort" or "nest"))'\
        eval.c
```

Show all lines containing "sort" but no "nest" in files with an extension .c, preceded by the name of the file:

```
sgrep -o "%f:%r" '"\n" _. "\n" containing "sort" \
                  not containing "nest"' *.c
```

(Notice that this query would omit the first line, since it has no preceding new-line character '\n', and also the last one, if not terminated by a new-line. For a correct way to express text lines, see the definition of the LINE macro below.)

Show the beginning of conditional statements, consisting of "if" followed by a condition in parentheses, in files *.c. The query has to disregard "if"s appearing within comments "/* ... */" or on compiler control lines beginning with '#':

```
sgrep '"if" not in ("/*" quote "*/" or ("\n#" .. "\n"))  \
                    .. ("(" ..  ")")' *.c
```

Show the if-statements containing string "access" in their condition part appearing in the main function of the program in source files *.c:

```
sgrep '"if" not in ("/*" quote "*/" or ("\n#" .. "\n"))  \
         .. ("(" ..  ")") containing "access" \
                          in ("main(" .. ("{" .. "}")) \
        .. ("{" .. "}" or ";")'  *.c
```

We see that complicated conditions can become rather illegible. The use of carefully designed *macros* can make expressing queries much easier. For example, one could give the below m4 macro processor definitions in a file, say, c.macros:

```
define(BLOCK,( "{" .. "}" ))
define(COMMENT,( "/*" quote "*/" ))
changecom(%)
define(CTRLINE,( "#" in start or "\n#"
                  _. ("\n" or end) ))
define(IF_COND,( "if" not in (COMMENT or CTRLINE)
                  .. ("(" .. ")")))
```

Then the above query could be written more intuitively as

```
sgrep -p m4 -f c.macros -e 'IF_COND containing "access"\
       in ( "main(" ..  BLOCK ) .. (BLOCK or  ";")' *.c
```

OPTIMIZATION

sgrep performs common subexpression elimination on the query expression, so that recurring sub-expressions are evaluated only once. For example, in expression

```
'(" " or "\n" or "\t") .. (" " or "\n" or "\t")'
```

the sub-expression

```
'(" " or "\n" or "\t")'
```

is evaluated only one.

DIAGNOSTICS

Exit status is 0 if any matching regions are found, 1 if none, 2 for syntax errors or inaccessible files (even if matching regions were found).

ENVIRONMENT

One's own default options for sgrep can be given as a value of the environment variable *SGREPOPT*. For example, executing

```
setenv  SGREPOPT  '-p m4 -o %r\n'
```

makes sgrep to apply m4 preprocessor to the expression and display each output region as such followed by a line feed.

FILES

Sgrep tries to read the contents of the files *$HOME/.sgreprc* and */usr/lib/sgreprc*. Generally useful macro definitions may be placed in these files. Using m4 (or some other) macro processor, for example the following definitions could go in one of these files:

```
define(BLANK,( " " or "\t" or "\n"))
define(LEND,( "\n" or end ))
define(LINE,( start .. LEND or ("\n" _. LEND) ))
define(NUMERAL,( "1" or "2" or "3" or "4" or "5" or
                 "6" or "7" or "8" or "9" or "0" ))
```

FUTURE EXTENSIONS

- Regular expressions (The most important missing feature)
- Built-in macro preprocessor
- More operations
- Indexing for large static texts

AUTHORS

Jani Jaakkola and Pekka Kilpelainen, University of Helsinki, Department of Computer Science, 1995.

BUGS

Sgrep may use lots of memory, when evaluating complex queries on big files. When sgrep reads its input text from a pipe, it copies it to a temporary file. sgrep does not have regular expressions in search patters.

SEE ALSO

awk(1), ed(1), grep(1)

sgrep home page at `http://www.cs.helsinki.fi/˜jjaakkol/sgrep.html`