NAME

lineprocx - (extended) text line processing

SYNOPSIS

lineprocx *ftempl fsource* [[[*fsep*] *outrsep*] *extra_args*] ...
lineprocx -*templ fsource* [[[*fsep*] *outrsep*] *extra_args*] ...

DESCRIPTION

lineprocx sequentially reads text lines from file *fsour ce* and process them using the filter (template) specified in file *ftempl*, or in the string *templ* in the second form.

Each source line is separated into fields using field separator *fsep* (def ault is ’,’). Then, the fields are replaced into the template text given in file *ftempl*, and the resulting text is written to the standard output.

If *sourcefile* is ’-’ then **lineprocx** uses the standard input as source file.

In the second form, when the first argument starts with ’-’, **lineprocx** uses the string *templ* (ignoring the leading ’-’) as the template description. The usual escape chars ’\n’, ’\r’ and ’\t’ are interpreted in *templ*. In addition, ’\b’ is interpreted as a whitespace.

In the optional command line arguments *fsep* and *outrsep* escape chars are also interpreted in this way.

The field and record separator are also accesible via the special variables **__IFSEP__**, **__OFSEP__** and **__ORSEP__** (see Section **VARIABLES**) and setting them via the optional command line arguments is not recommended.

Technically, there exist two field separators: the input field separator *ifsep* is used for splitting the input lines into fields, while the output field separator *ofsep* is used by the commands **#r** and **#*** (see Section **TEMPLATE COMMANDS**). Initially, both field separators are set to the same value *fsep*, but they can be individually modified with the above special variables.

The optional argument *outrsep* is deprecated and it is only used by the command **#r** (see Section **TEMPLATE COMMANDS**). There is no way to change the input record separator, that is always the end of line.

Source lines starting with a **#** have a special meaning (see Section **SPECIAL SOURCE LINES**).

TEMPLATE SYNTAX

All commands in a template specification start with the character **#**.

A template consists of three (optional) parts: a head, the body and a tail.

The head is some text that is written to stdout before processing the first source line. Similarly, the tail is written after processing the last source text line.

The sequence **#@** is used to separate the tree parts. Then, a complete template has the form:

*head***#@***body***#@***tail*

If only one separator **#@** is present, then the template is interpreted as

*head***#@***body*

(so, *tail* is the empty string).

If no separator exists at all the whole template definition is taken as *body*, and *head* and *tail* are both empty.

The output produced by **lineprocx** is in general

head
body_1
body_2
...
body_n
...
tail

where *body_i* is the result of replacing the fields parsed from the i-th source line into the template body.

Only the first 9 fields in a text source line can be processed separately. They are respectively refered as **#1**,**#2**,...,**#9**.

There are many other **#** commands that **lineprocx** understands in a template specification. Most of them can be also used in the template head and tail, as well. See in the next section a description of the recognized commands and their restrictions.

TEMPLATE COMMANDS

#1,...,**#9**
: They are replaced by the corresponding fields in the source text line. Non existing fields are assumed to be the empty string. They are not allowed in the template tail.

#*
: It is replaced by all fields in the source line, separated with the output field separator *ofsep*. This command is not allowed in the template tail.

##
: It is replaced by the '#' character.

#q
: It is replaced by a '"' character.

#]
: It is replaced by a ']' character. This command is necessary because ']' is used by **lineprocx** as a delimiter in some contexts, as explained in the following sections.

#f
: It is replaced by the output field separator, given as the *ofsep* command line argument (or its default value ','). It can be changed by assigning new values to the reserved variable **__OFSEP__** (see Section **VARIABLES**).

#r
: It is replaced by the output record separator, given as the *outrsep* command line argument (or the default value NEWLINE). It can be changed by assigning new values to the reserved variable **__ORSEP__** (see Section **VARIABLES**).

#n
: If used in the template body, it is replaced by the source line number. The source line counter is initialized to 0 and incremented before parsing a source line. It can be changed by assigning new values to the reserved variable **__NR__** (see Section **VARIABLES**).

#n *number*
: If used in the template head (but only in the template provided in the command line, see Section **INLINE TEMPLATES**), it initializes the source line counter to *number*, which otherwise defaults to 0. Thus, in the first source line processed the command **#n** in body will be replaced by *number*+1.

#t
: It is replaced by a string giving the current time (the string returned by *ctime*(3)).

#c *number*
: Only allowed in template head of the command line template (see Section **INLINE TEMPLATES**). It sets the replication counter to*number* (def ault is 1, that is, no replication). Each source text line is processed *number* times as if it was repeated *number* times in the source file (and then each repetition counts as a new line for the command **#n**). The replication counter can be changed by assigning new values to the reserved variable **__RREP__** (see Section **VARIABLES**).

#.0,...,**#.9**
: Are replaced by the corresponding command line argument $0,...,$9. Non existing command line arguments are assumed to be the empty string. Observe that **#.0** is the program name, **#.1** is the template file name, **#.2** is the source file name, etc. Escape chars are not interpreted (thus, **#.3** and **#f** can produce different results).

A single **#** occuring at the end of a line and the following newline character are ignored. This allows splitting the definition of a long output line into several lines in the template file.

Any other use of the character **#** is considered a syntax error in the template specification.

Template examples:

1) **BEGIN LIST**
   #@#

ITEM: #2
#@#
END LIST

This template example causes that the second fields of the source lines are listed, each in a separate line, with a header line "BEGIN LIST" and a tail line "END LIST".

2) **lineprocx '-[#@(#n,#3) #@#]\n' '-' '\t'**

Produces an inline list of the form

[(1,*f1*) (2,*f2*) ... (n,*fn*)]

where *f1,..., fn* are the third fields in each line read from standard input, using TAB as the field separator.

3) **lineprocx '-#@#*#f#+#*\n' -**

Joins every odd and even lines into a single line. But an input file with an odd number of lines produces an error.

CONDITIONAL EXPANSION

Some parts of the template can be enabled or disabled depending on the result of some comparisons. The syntax is the following:

#?[*expr1***]***op***[***expr2***]?[***expr3***]:[***expr4***]**

The above sentence is replaced by *expr3* if *expr1 op expr2* is true, otherwise it is replaced by *expr4*.

op is one of

= evaluates to true if *expr1* and *expr2* are identical strings

< evaluates to true if *expr1* is a substring of *expr2*

> evaluates to true if *expr1* is a superstring of *expr2*

^ evaluates to true is *expr1* is a prefix of *expr2*

$ evaluates to true is *expr1* is a postfix of *expr2*

/ evaluates to true is *expr1* matches the **grep**-style regular expression given in *expr2* (see section **REGULAR EXPRESSION MATCHING**).

Conditional expansion operators can be nested in any meaningful way (i.e., they can appear in any of *expr1*, *expr2*, *expr3* and *expr4*), and they are also allowed in the head and tail parts of the template.

Example:

#@#?[#*]=[]?[]:[*some_template_material***]**

This template only processes non-empty lines (except that empty lines are counted in **#n**).

REGULAR EXPRESSION MATCHING

In addition to the conditional expansion operator described in the previous section, there is a specific command, as shown below, that expands to the matching substring (so it does not just check its existence). Almost all features of grep extended regular expressions are implemented, including repetition operators and back references (see **grep** documentation). Notice that you may need to escape the active characters **#** and **]**, as in the following example of conditional expansion using the last regular expression:

#?[#*]/[([^a#]*a)+]?[it matches!]:[it does not match!]

In order to find the matching substring, the following command is introduced:

#/[*string***]/[***regexpr***]**

that expands to the first longest substring of *string* that matches the regular expression *regexpr*. Using the same example again, the following code

#/[abcdefabcdaaazyx]/[([^a#]*a)+]

will expand to the string **abcdefabcdaaa**. If there is no such matching substring, the command expands to

the empty string. Thus, the only way to tell apart the two cases of "no mathing found" or "the empty string is the longest matching substring" is via the conditional expansion operator.

Combining back references and repetition operators in regular expressions must be done with some care, since most implementations are buggy (for some weird regular expressions). In this implementation the value of a back reference is the last match of the corresponding parenthesized subexpression. In some weird situations, two nested back references can contain values that are not contained as substrings in the expected way (this also happens for instance in some implementations of **grep**).

As an extension, the result of the expansion of the regular expression matching command can be specified by a special template appended to the regular expression. A/ is used as a separator (thus, a literal / inside a regular expression must be escaped as \/.)

The template is copied as the expansion result, except that back references are replaced by their values, and the escaped characters \\, **\n** and **\t** are interpreted as usual. Invalid or unmatched back references are just ignored. The default template is just the global back reference **\0**. As an example, the following template lists the values of the first three back references:

References:\n\t\\1=\1\n\t\\2=\2\n\t\\3=\3\n

The extension (i.e., the template specification) is ignored by the regular expression matching conditional expansion command. Indeed, anything following a / (including itself) in the regular expression is ignored. Similarly, a second / and all subsequent characters are ignored in the regular expression specified in the command **#/**, with one exception: a character | following the second /. This special case makes the matching operator to dump the back references into the input fields, where **#1** takes the value of the whole matching substring, i.e., the back reference **\0**, **#2** is the back reference **\1**, and so on. **#*** is the usual concatenation of input fields and output field separators **#1#f#2#f#3...**. The system variable **__NR__** is updated accordingly. Unmatched back references are simply taken as the empty string.

As soon as an input fields updating command like **#+**, **#-** or **#|** is executed, the previous association is lost, and the same occurs after any call to a regexpr matching command, with or without the | character. Notice that the regexpr matching conditional expansion operator does not offer the ability to associate the input fields with back references.

The following line is an example of the full regular expression matching command:

#/[23/11/1920]/[([0-9]+)\/([0-9]+)\/([0-9]+)/year=\3\n/ignored]

SPECIAL SOURCE LINES

Source lines starting with **#** are considered as comments and are ignored except when it is followed by one of **<**, **[**, **]** or **!**. These special source lines do not increase the record counter, used by **#n**.

Special source lines implement two extra functionalities of **lineprocx**: nested input files and inline templates, as described in the next sections.

INLINE TEMPLATES

One of the interesting features of **lineprocx** is the possibility of specifying the templates in the same source file, then allowing to specify which portion of the source file is processed with a specific template.

lineprocx implements a stack of inline templates. The special source line

#[[*inline_template_definition*

parses the template definition, pushes it into the stack and activates it (i.e., prints the template head) and uses it to process the subsequent source lines, eventually until the special input line

#]]

is reached. This last line prints the inline template tail and pops it form the stack, so that a previously active inline template is reactivated. Any characters following **#]]** are ignored.

The old inline templates are retained in memory. The special line

#[[

reactivates a previously defined inline template (and the head is printed again). Undefined templates are

assumed to be empty (i.e., with a trivial body), and they produce no output.

Example:

Here, the template given in the command line is used
#[[*templ1*
templ1 head is printed
...
Here, *templ1* is used
...
#[[*templ2*
templ2 head is printed
...
Here, *templ2* is used
...
#]]
templ2 tail is printed
...
Again, using *templ1*
...
#[[
templ2 head is printed again
...
Using now *templ2*
...
#]] This text is ignored!!!
templ2 tail is printed
#]]
templ1 tail is printed
...
The command line template is used here

There are some commands that behave slightly different in inline templates. Namely, **#c** cannot be used in any place of an inline template, while **#n** retains its normal meaning even in the head of the inline template (it prints the record counter).

MULTILINE INLINE TEMPLATES

To increase readability, the definition of an inline template can be split into several lines by ending each line with a **#** and starting the following one with another **#**. Any space after the starting **#** is ignored (you can use **\b** to specify a not ignored space).

Lines starting with **##** are considered as comments and they are skipped. This feature can be used to comment out a single line or to document a multiline inline template. A comment line does not need to end with a **#** character. To comment out the entire multiline template it suffices to prepend an extra **#** character to its first line (i.e., **##[[#**).

Example:

```
#[[#
#   The previous four spaces are ignored#
# #@#
#   The first field in the source line is: #
#   #1\n#
##  (this is a comment line, and it is ignored)
# #@#
#   \b The tail starts with two spaces#
#
```

Within inline templates, escape chars are interpreted as in the command line arguments.

NAMED INLINE TEMPLATES

Inline templates can be referred by name, using the special source line syntax

#[*templname***]***templdef*
#[-*templname***]***templdef*
#[*templname***]**
#[] this text is ignored!

The first line defines and activates (i.e., prints its head) the template specified in *templdef* and names it *templname*. The second line defines but does not activate (i.e., does not print its head) the template, for further activation. The third line activates the previously defined template named *templname*. The fourth line closes the current named inline template (thus printing its tail).

Named inline templates can be nested, and also be nested with the unnamed ones. However, named templates must be closed with **#[]** while unnamed ones with **#]]**. Failing to do so can cause an unpredictable behavior or even an error.

Names in named inline templates are taken verbosely (i.e., neither command nor escape chars are interpreted). Note that **]** cannot be part of the name of a named inline template. In addition, the name must not start with **-**, because otherwise the template will never be activated.

A named template can be redefined to be the empty template (but not activated) with

#[-*templname***]**

You can also clear and activate the named inline template with

#[*templname***]#@**

Observe that any redefinition of a named inline template has immediate effect in all its active nested instances.

VARIABLES

lineprocx allows variables named with arbitrary strings, taking as values also arbitrary strings. The syntax for setting a variable is:

#![*varname***]=[***value***]**

and any subsequent command

#&[*varname***]**

is replaced by *value*.

#‘[*varname***]**

interprets the variable value as a direct template, and expands it.

#^[*templ***]**

directly uses the direct template specified in *templ* without the need of storing the text into a variable.

To avoid unnecessary errors, undefined variables are assumed to have empty values.

Variable names and values are interpreted (i.e., command substitution and conditional expansion is performed and escape chars are interpreted) at special source line parsing time. As a side effect, this allows unusual definitions in which the variable name depends on other variable values, and it also enables recursive variable definitions (like appending or prepending some text to a given variable content).

To delay the command substitution and conditional expansion you would need to escape all **#** and all **]** corresponding to the delayed constructions. For instance

#![*varname***]=[##&[***othervar***#]***stuff* **]**

will set the variable to the value

#&[*othervar***]***stuff*

where the commands in *stuff* have been executed. A further execution of the variable *varname* with

#'[*varname***]**

will expand *othervar* to its current value. There is an alternative way to specify code with delayed command execution (see Section **DELAYED EXPANSION CODE**).

Some special variables are predefined at program startup: **__IFSEP__**, **__OFSEP__**and**__ORSEP__**, which contain the input and output field and output record separators. Input field separator is used to split data lines into fields, while the output field separator is used by commands **#f** and **#***. Both field separators are initially set to argv[3] (or to the default value '**,**'. The output record separator is only used by**#r**, and it is initially set to argv[4] (or to the default value '\n'). All of them are used as normal variables, and can be modified at any time. Therefore, the use of argv[3] and argv[4] to set the separator values is deprecated (but still supported).

__NR__ and **__RREP__** contain the record counter and the replication counter. related to the commands **#n** and **#c**. When setting the last two variables, only the prefix of the value that can be parsed as an integer is considered, and the remaining part is discarded without causing any error.

__NF__ is another system variable that contains the number of fields in the current record. It is a read-only variable, that is, any attempt to modify its value is simply ignored. Another read-only variable, with an obvious value, is **__VERSION__**.

FLEXIBLE AND COMPACT FIELD SEQUENCES

There is a way to write in a compact and flexible way a sequence of input fields, separated by output field separators. The syntax is

#[*seq_spec***]**

where *seq_spec* is a comma separated list of field ranges. Any field range has the form *x***:***y***:***z* where *x* is the initial field number, *y* is the (nonzero) increment and *z* is the final value. The range includes the values *x*, *x+y*, *x+2y*, ... until *z* is reached. The value *z* is only included if it is a member of the previous sequence. If *x* and *z* are different but *z-x* and *y* have different signs, the range is empty.

*x***:***z* is an abbreviation of *x***:1:***z* while a single value *x* corresponds to a range consisting of a single element (that is, it is an abbreviation of *x***:1:***x***.** Non positive values have a special meaning. Namely, 0 denotes the last field, -1 is the second last, and so on. Therefore,

#[2:2:0,1:2:0]

generates the sequence of all fields at even positions, followed by the ones at odd positions, or

#[-1:-1:2]

corresponds to all fields except for the first and last ones, in reversed order.

The field sequence specification is first processed as other bracketed expressions. This means for instance that it can contain variables, conditional expansion ot math expressions (see Section **MATH EXPRESSIONS AND VARIABLES**).

In a similar way, command line arguments can also be handled with range specifications with the syntax

#.[*seq_spec***]**

with the only difference that**#.[0]**, **#.[-1]**, **#.[-2]**... no w respectively refer to the first, the last, the second last command line arguments etc., i.e., **argv[0]**, **argv[argc-1]**, **argv[argc-2]**...

DIRECT TEMPLATES

The last type of inline templates are called direct, because they are only used implicitly and they do not process any subsequent source lines (they act as if they are closed immediately after being opened). The syntax

#!*templdef*

is esentially equivalent to

#[[*templdef*
#]]

except that it does not affect the inline templates stack. The commands **#1**,...,**#9**,**#*** are meaningless, and they produce no output, and as in other inline templates, they are not allowed in the direct template tail.

Direct inline templates can be useful to set some variables or to print their values.

Example:

```
#!#
#  #![program name]=[#.0]#
#  #![command line templ]=[#.1]#
#  #?[#.5]=[]?[#
#    #![opt arg 5]=[default value]#
#  ]:[#
#    #![opt arg 5]=[#.5]#
#  ]#
#  This is #&[program name] running at #t #
#  on a template #
#  #?[-]^[#&[command line templ]]?[#
#    given in the command line#
#  ]:[#
#    from file #q#&[command line templ]#q#
#  ]\n#
#
```

produces the output

This is lineprocx running at Mar 27 14:11:40 2018 on a template given in the command line

NESTED INPUT FILES

The special source line

#< filename

causes the inclusion of the source file *filename* at the point it is located. A source file stack is implemented in **lineprocx**, and every #< special source line pushes a new source file on it. Input is always read from the top of the stack. At end of file, the exhausted source file is poped out from the stack. The program ends when the source file stack is empty.

filename can include any command allowed in a direct inline template, including variable substitutions and conditional expansions, and also the file name definition can be split into different lines. The string obtained after these expansions is used as the actual source file name.

Example:

```
#<#
#   #?[#&[name]]=[]?[#
#     default#
#   ]:[#
#     #&[name]#
#   ].dat#
#
```

opens a source file with extension ".dat" and name determined by the contents of the variable **name**, or "default" if the variable is empty.

MULTIPLE INPUT CHANNELS

The input file stack (implementing the "nested input files" feature) is replicated 10 times, providing 10 independent input channels. The default input channel is 0. The following template commands provide input channels management:

#<0,...,**#<9**

Select the corresponding input channels 0,...,9. 0 is the default channel (used at program startup).

#<? Checks whether the current input channel is ready or exhausted. It expands to 0 if there is no pending input in the channel, or to 1 otherwise.

#<- Closes the current input file on the current input channel.

#<[*filename***]**
Pushes the file *filename* into the input file stack corresponding to the current channel.

The readonly variable **__INPUT__** holds the number of the current input channel.

Example:

lineprocx '-#@#*:\n#<1#<[#*]#+#*\n\n#<-#<0' *filelist*

outputs every filename listed in *filelist* follo wed by its first line, then a blank line. See below the explanation of the special command **#+**.

IMMEDIATE OUTPUT

The special command **#>[**<*text*>**]** immediately outputs <*text*> to the standard output, without storing it into the expansion buffer. This can affect the order in which the resulting text is printed. Indeed, in the following example of a direct template

#!text1 #>[text2] text3

the output will be **text2 text1 text3** because **text2** is not stored in the expansion buffer, while **text1** and **text3** are, and the entire buffer is flushed when the template expansion ends.

This command is useful to directly output the text that does not need further processing, thus saving space in the expansion buffer (e.g., in a direct template that contains a loop).

MATH EXPRESSIONS AND VARIABLES

The following dedicated syntax constructions is reserved for mathematical expressions and variables:

#$[*assignment expression***]**
#$[@*assignment expression***]**

The first form evaluates the expression and prints the numerical result, while in the second form only evaluation is performed. The assignment expression can have two forms:

<*assignment expression*> := [<*variable name*> <*assignment operator*>] <*assignment expression*>
<*assignment expression*> := <*simple expression*> [**;** < *format specification*>]

The simple expression is just a mathematical expression involving the usual math operators and functions as well as variable names. The optional format specification is any of the allowed format specifiers for integers or floating point numbers supported by the C printf function.

The length specification characters 'l' and 'L' are not allowed (the format will be considered as invalid). If no format is specified or it is invalid, the default format "%g" is used. Integer types like 'i', 'd', 'u', 'x' causes a conversion of the evaluation result from floating point to integer. For example

#$[pi;%d]

produces the output "3".

The assignment operators can be either **=** or a binary operator followed by **=**, like **+=**, **-=**, etc.

Supported binary arithmetic operators are **+**, **-**, *****, /, **%** and ^, where the last two correspond to remainder and power. Only two unary arithmetic operators are defined: **+** and **-**, and two constants: **e** and **pi**. The supported functions with one argument are **abs** (for fabs), **acos**, **asin**, **atan**, **ceil**, **ch** (for cosh) , **cos**, **exp**, **floor**, **frac** (for fmod(x,1)), **log**, **log10**, **sh** (for sinh), **sin**, **sqr** (for squaring), **sqrt**, **sgn** (for the sign function), **tan**, **th**(for tanh) and **u** (for the unit step function). With two arguments, the program supports **atan2**, **hypot**, **pow**, **max**, **min** and **mod** (for fmod). Standard C-like comparison and logical operators are also supported. Namely, the binary **<**, **>**, **<=**, **>=**, **==**, **!=**, **&&** and ||, and the unary negation **!**. All these operators return 0 (false) or 1 (true) as a boolean result.

Only the function with zero arguments **rand** is provided. It must be called as **rand()**.

Variable names can be formed with alphanumeric characters and the characters **@** and **_**, but the starting character must not be a digit. The variable name length is limited to 64 characters.

An examples of supported assignment expressions is **x+=y=z+=eˆ3*sin(3*pi/2)**, where **z** is incremented by **eˆ3*sin(3*pi/2)**, then **z** is copied into **y** and finally **x** is incremented with the resulting value of **y**.

Conditional expansion and variable substitution are performed before evaluating the mathematical expression (as in a direct template). This feature allows the use of nested mathematical evaluations. In particular, the name of a mathematical variable can depend on the result of a mathematical expression (if it is a non-negative integer), like in **#$[x_#$[23*2]=#$[-2*5]5]**, which is equivalent to **#$[x_46=-105]**.

There are some additional abbreviated syntax constructions: **#$[&***name***]** and **#$[@&***name***]** are the same as **#$[#&[***name***]]** and **#$[@#&[***name* **]]**, and, similarly with **#$[‘***name***]** and **#$[@‘***name***]** which mean **#$[#‘[***name***]]** and **#$[@#‘[***name* **]]**.

The variables created by an assignment expression are kept for subsequent expression evaluations. The mathematical variable namespace is independent of the normal **lineprocx** variable namespace. Attepting to use an undefined math variable causes an error.

CONDITIONAL MATH EXPANSION

The following syntax combines the conditional expansion and mathematical evaluation features:

#?$[*math expression***]?[***true expression***]:[** *false expression***]**

It expands *true expression* if *math expression* evaluates to a nonzero number, and *false e xpression* otherwise. All formats of a mathematical expression defined in the previous section are allowed, but the "mute character" **@** and the format specification are simply ignored.

The mathematical expression can perfectly be an assignement expression, and constructions like

#?$[&*varname***]?[***true expression***]:[** *false expression***]**
#?$[‘*varname***]?[***true expression***]:[** *false expression***]**

take the natural meaning, according to the previous section.

SPECIAL INPUT HANDLING COMMANDS

There are some specific commands that change the normal behaviour of the source text parsing mechanism.

#+ Parses a new source text line (without incrementing the line counter associated to command **#n**) and overwrites the fields, affecting the meaning of the commands **#1**,...,**#9** and **#***. The command itself produces no output. As a side effect, **#+** can cause a special source input line be dealt as a normal input line.

#- Parses again (perhaps using a different input field separator) the same source text line (without incrementing the line counter associated to command **#n**) and overwrites the fields, affecting the meaning of the commands **#1**,...,**#9** and **#***. The command itself produces no output.

#:[*regexpr***]**
Parses again the same source text line, but the field separators are the substrings matching the regular expression *regexpr*. The matching substrings are also stored as input fields, corresponding to the even indexes **#2**, **#4**..., while the proper fields (substrings limited by *regexpr* matchings) are stored into the odd indexes **#1**, **#3**.... Any literal occurrence of a / in *regexpr* must be escaped as \/ and any character in *regexpr* following a /, including itself, is ignored. If both a proper field and the matching are zero length strings, then the proper field is assumed to be a single character, in order to avoid endless loops. Notice that the anchor ˆ in *regexpr* does not refer to the beginning of the source text line but to the end of the previous matching. Therefore, a regular expression starting with a ˆ can only be matched if the previous proper field is the empty string.

#|[*data***]**
Sets the current source text line to *data* and parses it (without incrementing the line counter associated to command **#n**) and overwrites the fields, affecting the meaning of the commands **#1**,...,**#9** and **#***. The command itself produces no output. The data contained in*data* is processed with the same rules as other bracketed text (i.e., it can contain nested conditional execution commands,

variables, math expressions, etc.).

Actually, **#-** is an abbreviation of **#|[#*]**, that is also equivalent to **#|[#[1:0]]**.

One can combine some of the above commands, for instance to ignore the regular expression matching field separators and keep only the proper input fields, with

#:[*regexpr***]#|[#[1:2:0]]**

Setting **__IFSEP__** to an empty string makes the parser to interpret characters as separate fields. Using this feature in combination to the flexible and compact field sequence specification, one can perform some useful processing in a simple way. For instance, the following named template

```
#!#![temp_IFSEP]=[#&[__IFSEP__]]#![__IFSEP__]=[]
#!#![temp_OFSEP]=[#&[__OFSEP__]]#![__OFSEP__]=[]
#[gobble]#@#[2:0]\n
 ...
 data lines
 ...
#[]
#!#![__IFSEP__]=[#&[temp_IFSET]]
#!#![__OFSEP__]=[#&[temp_OFSET]]
```

omits the first character in each data line, or

#[reverse]#@#[0:-1:1]\n

reverses the characters in each data line.

SPECIAL TEMPLATE EXPANSION CONTROL COMMANDS

There are two additional commands that alter the normal template expansion behavior:

#_ immediately stops the outer template head, body or tail expansion (either in the command line template or an inline template, and also in any direct template), and then proceeds with the following source line.

#% stops the outer template expansion if it occurs during its body expansion (i.e., the command is ignored when it occurs either in the head or in the tail expansion of the outer template) and restarts the expansion of the template body causing a (possibly endless) loop. Direct templates also implement loops in the same way.

Direct templates include the special lines starting with **#<** or with **#!** and also all executions of the contents of a variable with **#‘** or related commands, like **#?[‘** or **#?$[‘**, and the direct template expansion command **#^**.

Loop command **#%** should be used with extreme care. The loop termination condition must be ensured by the programmer, typically implying both a conditional expansion command and a redefinition of the input source line via **#|[...]** command.

For instance, the command line template

#$[@x=0]#@#?[#*]=[]?[#$[x]\n#$[@x=0]]:[#$[@x+=#1]#|[#[2:0]]#%]

adds all the fields in each input record and prints the result, and the named template

```
#[-polyval]#
# #$[@y=0]#
# #@#
# #?[#$[u(#&[__NF__]-2)]]=[0]?[#
# #$[y]\n#$[@y=0]#
# ]:[#
# #$[@y=y*(#1)+(#[0])]#|[#[1:-1]]#%#
# ]#
#
```

performs a polynomial evaluation for every record, where the polynomial coefficients are the fields **#2**, **#3**, **#4**... and the evaluation point is the first field.

DELAYED EXPANSION CODE

The easiest way to delay command expansion is by using **##** and **#]**, that produce the literal characters **#** and **]**. Then, you can write something like

#![setvar]=[##![var#]=[##$[value+=1#]#]]

that will increment and assign the value to the variable at the point where you execute

#‘[setvar]

Alternatively, you can surround the delayed execution code with **#{**, **#}**. All commands within the delimiters are not executed but they are left unchanged. However, escape chars are still interpreted. Nested **#{**, **#}** groups are supported. The inner delimiters are left unchanged.

Moreover, multiline direct templates can be specified as above: Lines ending in a single # followed by a line starting with a single # are joined into a single line (removing the ending **#**, the newline character, the starting **#**, and any subsequent whitespaces).

The previous example will now be written as

#![setvar]=[#{#![var]=[#$[value+=1]]#}]

MORE EXAMPLES

An inline template that parses every line starting with ’\’ and "executes" it as a command:

#[expand]#?[\\]^[#1]?[#‘[#1]]:[#*\n]

A ’\define’ macro that defines other commands to be used within the previous template:

#!#![\\define]=[##![##2#]=[##3#]]

or with the alternative syntax:

#!#![\\define]=[#{#![#2]=[#3]#}]

An example of use of the previous objects:

```
## Set the field separator to SPACE
#!#![__FSEP__]=[ ]
\define \HTMLremark \n<!--\b#2\b-->\n
## Set the field separator to TAB to make spaces ordinary chars.
#!#![__FSEP__]=[\t]
```

Now you can use:

\HTMLremark This is a comment in the HTML file

to insert a remark in a HTML generation procedure.

Another example:

```
#!#![__FSEP__]=[ ]
## Teach how to increment small numbers:
\define 0+1 1
\define 1+1 2
\define 2+1 3
\define 3+1 4
\define 4+1 5
\define 5+1 6
\define 6+1 7
\define 7+1 8
\define 8+1 9
\define 9+1 10
## Teach how to operate a counter i:
```

```
\define \i= #![i]=[#2]
\define i++ #![i]=[#‘[#&[i]+1]]
\define \i++ #‘[i++]
\define \i? #&[i]
```

Now you can run, for instance:

```
\i= 0
\i++
\i?
```

and the program answers 1. The program actually knows how to decrement a number:

```
\define try #?[#‘[#&[i]+1]]=[#2]?[]:[#‘[i++]#‘[try]]
\define \prev #![i]=[0]#‘[try]#&[i]
```

Now you can run:

```
\prev 6
```

and the program answers 5.

A calculator example:

```
lineprocx ’-calc$ #@#![line]=[#*]#$[‘line]\ncalc$ ’ -
```

turns **lineprocx** into a math calculator.

A more detailed example: a filter that transforms a data file into a SVG file with a scatter data plot.

```
### CREATION OF A SIMPLE SCATTER PLOT SVG FILE
### Arguments: #.3=width #.4=height #.5=datafile
###
#!#![__RSEP__]=[\n]
#!#![__IFSEP__]=[ ]
#!#![__OFSEP__]=[ ]
###
### Define a named inline template for
### Latex type commands syntax
###
#[-expand]#
# #@#
# #?[\\]^[#1]?[#
## COMMAND EXECUTION
# #‘[#1]#
# ]:[#
## NORMAL LINE
# #*\n#
# ]#
# #@#
#
###
### The following macro sets the variable #2 to #3
### without execution the commands in #3.
### E.g., the line
### \define \foo today\bis\b#t
### defines the variable named \foo to today\bis\b#t
### but #‘[\\foo] results in today is Mar 15 12:13:49 2019
###
#!#![\\define]=[##![##2#]=[##3#]]
###
### Activate the expand template
```

```
###
#[expand]
###
### Some auxiliary definitions:
### To force the execution of the commands contained
### in the arguments, replace #1 by #‘[arg1], etc.
### Technically, #‘[arg1] copies #1 into the variable
### _arg and then the content of _arg is executed.
###
\define arg1 #![_arg]=[#1]#‘[_arg]
\define arg2 #![_arg]=[#2]#‘[_arg]
\define arg3 #![_arg]=[#3]#‘[_arg]
\define arg4 #![_arg]=[#4]#‘[_arg]
###
### Define the marker generation. Literal blanks are escaped with \b,
### to avoid interpreting them as field separators.
###
### Each record in the data file consists in the fields: x y r c,
### The coordinates x and y must be in the interval [-1,1]
### but the actual picture coordinates are transformed to [0,w]x[0,h]
### The radius is set to 0.01*r*w, and r defaults to 1.
### The color defaults to black, and it is a string.
### If the color string starts with rgb, then the commands in the
### string are executed.
###
\define x #$[x=(1+(#‘[arg1]))/2*w]
\define y #$[y=(1-(#‘[arg2]))/2*h]
\define r #$[r=(#?[#3]=[]?[1]:[#‘[arg3]])*0.01*w]
\define strokecolor black
\define fillcolor #?[#4]=[]?[black]:[#?[rgb]^[#4]?[#‘[arg4]]:[#4]]
###
\define \marker <circle\bstroke="#‘[strokecolor]"\bfill="#‘[fillcolor]"\bcx="#‘[x]"\bcy="#‘[y]"\br="#‘[r]"
###
### Text direct output for the SVG header
###
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE svg PUBLIC -//W3C//DTD SVG 20010904//EN http://www.w3.org/TR/2001/REC-SVG-2001
###
### The next text lines require command execution.
### Then, we use direct templates.
### (The final \n is necessary to produce a newline in the output.)
###
#!<svg width="#$[w=#.3]" height="#$[h=#.4]">\n
#!<path stroke="none" fill="white" d="M0 0l#$[w] 0l0 #$[h]l#$[-w] 0x" />\n
###
### Define and activate a marker generator named template
###
#[draw]#‘[\marker]
###
### Read data file
###
#<#.5
###
### Deactivate named template
```

```
###
#[]
###
### More direct text output (without commands)
###
</svg>
#[]
### End
```

If the previous filter file is named svgfilter, it can be used to convert a file called data into the SVG file data.svg with size 400x300 as follows:

```
lineprocx - svgfilter 400 300 data > data.svg
```

Actually, the data file can have **lineprocx** executable code, like for instance

```
#!#$[@n=100]#![__IFSEP__]=[:]#![__RREP__]=[#$[2*n+1]]
#!#![__NR__]=[#$[-n-1]]
### Green plot of the sin() function, default radius
### The color is specified in rgb(R,G,B) format
### Next record is repeated 2n+1=201 times. Then #n takes
### the values -100,-99,...,100.
(#n)/n:sin(.1*(#n))::rgb(16,250,54)
#!#![__NR__]=[#$[-n-1]]
### Red plot of the cos() function, default radius
### The color is specified in #RRGGBB format
(#n)/n:cos(.1*(#n))::#ff0000
#!#![__NR__]=[#$[-n-1]]
### Black (dotted) plot of the x-axis, smaller radius
### The color is just a named color
(#n)/n:0:.2:black
#!#![__RREP__]=[1]#![__IFSEP__]=[ ]
```

produces plots for the sin and the cos functions and also the x-axis, in different colors. The use of **__RREP__** allows the automatic generation of 100 points with a single data line.

OPTIONS

No command line options are currently provided in **lineprocx**.

BUGS AND MISSING FUNCTIONALITIES

The behavior of the command **#+** with the special input lines can be considered either as a bug or as an unexpected functionality of **lineprocx**.

The syntactic limitations in the names of inline templates (use of **-** and **]** chars) are rather artificial.

Named and unnamed inline templates are not properly mixed, specially when dealing with the deactivation commands **#[]** and **#]]**.

Escape chars could be also interpreted in the optional command line arguments, when used through commands **#.0**,...,**#.9**.

There are many missing functionalities that could be easily added to **lineprocx**, while keeping a degree of compatibility with the existing ones:

Redirecting output to different files would enable splitting a file into several ones.

CURRENT SPACE LIMITATIONS

Some of the data structures used by **lineprocx** are static and have a fixed predefined size:

Max. field and record separator length: 255
Source file name length: 512
Source line buffer: 4096
Command line template definition buffer: 65535

Template expansion buffer: 65535
Command line template (command) fields: 256
Math variable name length: 64
Number of nested unnamed templates: 32
Number of recorded back references in a regexpr: 32

HISTORY

An early version of the program with a very limited functionality existed with the name **lineproc** (from "line processing"), until May 2006. The 'x' in **lineprocx** comes from "extended". The initial aim of the program was to build an easy-to-use tool for extracting, shuffling, replicating fields in data files organized as one-record-per-line text file.

Different functionalities were gradually added to the application: Inline templates and contitional expansion were added in 2007. Variable definitions, access to command line arguments, and the special variables **__ORSEP__**, **__IFSEP__** and **__OFSEP__** were added in 2018. One important improvement, introduced by the end of 2018, was the **#‘** operator, which substantially increased the expressiveness of the scripting language. In the early 2019 most syntax restrictions of the head and the tail portions of the templates were removed. Also most objects and containers are made dynamic, removing most restrictions about the number of objects simultaneously defined or the maximum length of some buffers. It also helps to improve the program performance, since variable name lookup is done in hash tables of binary trees. Another essential improvement was the integration of a math expression evaluator into **lineprocx** inspired in a previouly existing (from 1995) RPN calculator called **vexpr**. In 2021, experimental support for regular expression matching was added, but there would be some room for performance improvement.

This application have been mainly used to generate webpages like the ones of Eurocrypt 2007 and PKC 2008 conferences, or the author's personal webpage, and also to generate some classnotes documents both in html and latex formats from the same source. Other applications are selectively extracting information from xml files, automatic generation of svg files or formatting text information from non-standard sources (like copying and pasting from pdf files).

awk(1) is far more powerful, flexible and reliable. However, the extra functionalities added to **lineprocx** makes it useful when a data file contains different sections to be filtered in different ways, since both the data and the filter definitions are integrated into a single file.

AUTHOR

Jorge L. Villar (jorge.villar@upc.edu).

SEE ALSO

grep(1), **awk**(1), **colrm**(1), **paste**(1), **column**(1), **sed**(1).