Gema —
a general purpose macro processor

David N. Gray <dgray@acm.org>

March 17, 1995
Revised May 21 and August 20, 1995, and December 26, 2003

1 Introduction

This document is the user manual for **gema**, a program whose name stands for **ge**neral purpose **ma**cro processor. This is a utility program which is run as a shell command under Unix, MS-DOS, or Windows and can be used for performing conversions or translations of data files or extracting information from files.

The `man` page for **gema** provides a tutorial introduction and a brief reference summary, while this document provides a more detailed specification. It would probably be best to read the tutorial section first before reading the rest of this document. The command line options are fully documented in the `man` page, so will not be repeated here.

This version of the document is consistent with versions 1.2 through 1.4 of the program. (Note that the `-version` option can be used to check which version you are running.)

2 Operational Overview

The program operates on files which (as in the C programming language) are a stream of bytes, with lines separated by a new line character, which is denoted as "`\n`".

The general model of operation is that the program reads an input file and writes an output file which consists of the input data transformed in accordance with a set of transformation rules provided by the user. A rule consists of a *template* and an *action*. The template is a pattern which the program will attempt to match with the input data. Any input text that matches a template pattern will be replaced by the result of evaluating the rule's action. There may be multiple sets of rules, where each set of rules is called a *domain*. At any given time, translation is controlled by the rules of one particular domain, but both templates and actions are able to switch to a different domain for processing particular portions of the data. A domain can inherit from another domain, meaning that if no match is found for the current input text in any of the rules for the current domain, then the rules of the inherited domain will be tried.

Processing of a file begins using the default domain, whose name is the empty string. First, if there is a rule with template "`\B`" (beginning of file) or "`\A`" (beginning of data), then its action is performed. Then the program begins reading the file. For each character position in the file, the program attempts to find a rule in the current domain whose template pattern matches the input text beginning at that point. If a match is found, then the input stream is advanced to the end of the matched text and the rule's action is executed. When no template matches the current position, the current character is copied to the output file (unless the `-match` option is being used), the input stream is advanced to the next character, and it tries to find a template matching the text starting at that position. When the end of the input file is reached, if there are any rules with template "`\E`" (end of file) or "`\Z`" (end of data), their actions will be executed, and then the files will be closed.

However, if a template matches without advancing the input stream (for example, if it begins with "`\P`"), then after executing its action, the search continues as though it had not matched. This is necessary to avoid hanging in a loop repeating the same match forever.

A rule may have an empty action, with the effect that the matching text is simply discarded. In each domain there may be at most one rule with an empty template, which signifies a default action to be taken when no other rule matches. However, since an empty template does not cause the input stream to be advanced and there are no more rules to try, this is only meaningful if the corresponding action exits the current context by using one of `@end`, `@terminate`, `@fail`, or `@abort`.

Generally speaking, while looking for a match, the rules within a domain will conceptually be tried in the same order in which they were defined, so wherever there might be ambiguities, the user should define the rules for preferred special cases before the rules for default general cases. However, there are some important exceptions:

- Rules beginning with a literal character (including space or "`\S`") will be tried before rules beginning with an argument. (Operators that don't advance the input stream, such as "`\N`", "`\I`", or "`\L`", are ignored for the purpose of this determination.) The way this actually works is that the current input character is used as an array index to find the list of rules beginning with that character. If none are found, or none match, then the rules beginning with arguments are tried in sequence.
- If two rules begin with the same sequence of literal characters, the one with the longer literal string will be tried first. This is because if this were not done, a more specific rule appearing later would never match, which would surely not be what the user intended.
- If two rules have identical templates, the second rule will replace the first on the assumption

that it is a redefinition.[1]

Rules can be defined either as arguments of the `-p` command line option or in pattern files loaded by the `-f` option. Each line in a pattern file can be one of the following:

- A comment line, indicated by a "`!`" in the first column.
- Any blank lines are ignored.
- An immediate action – a line beginning with "`@`" can contain one or more function calls which will be evaluated immediately before reading the next line. Normally these should be functions that are used for their side effects and do not return a value. The most common case is using "`@set`" to initialize a variable.
- Any other line is expected to be a rule, which should contain a "`=`" separating the template and action. A line may contain multiple rules, separated by semicolons. A rule may be continued on another line by ending the line with a backslash, which causes the following newline to be ignored and skips over any leading spaces on the following line.

  Rules may be preceded by a domain name followed by a colon, which causes all of the rules following on the same line to be defined in the designated domain. The domain name may optionally be enclosed in angle brackets, and any leading or trailing blanks will be ignored. If the domain is to be referenced in an action following "`@`", then the name should be limited to letters, digits, hyphen, and underscore.
- A line of the form "*name1*`::`*name2*" specifies that the domain named on the left inherits from the domain named on the right. Each domain can inherit from at most one other, but multiple levels of inheritance is allowed.

A template may contain any of the following:

- Literal characters, which are to be exactly matched. Literal characters include letters, digits, special characters quoted by a backslash, control characters designated by backslash followed by a lower case letter or digit, control characters designated by "`^`" followed by a letter, and any other characters that don't have any special meaning. Any of the 256 possible characters can be used.
- Arguments, which match some variable portion of text, and remember the matched text so that it can be referenced later. There are several different kinds of arguments, which are denoted by "`*`", "`?`", "`#`", "`<...>`". and "`/.../`". The current implementation allows a template to have a maximum of twenty arguments.
- Template operators, denoted by a backslash followed by an upper case letter, or by the space character. These may set local options, impose additional requirements for a match, or allow skipping of redundant spaces in the input.
- A dollar sign may be used followed by a single letter or digit to insert the value of a variable or a previous argument into the text that is to be matched. (This does not currently work for "`*`" arguments.)

An action may contain any of the following:

[1] This policy was adopted before `@undef` had been invented; it might be better to warn about duplicates and require explicit undefinition before redefining.

- Literal characters, which when evaluated simply copy themselves to the output stream.
- The value of a template argument can be output by using a dollar sign followed by the argument number (enclosed in braces if more than one digit), or by using the characters "`*`", "`?`", or "`#`" to denote the corresponding argument.
- The notation "`${`*name*`}`" can be used to output the value of a variable.
- The notation "`@`*name*`{`*args*`}`" can be used to call a built-in function or to translate the argument with a user-defined domain.
- Some of the operators denoted by a space or backslash followed by an upper case letter also apply in actions, typically for conditional output of spaces or newlines.

A notable difference from traditional macro processors is that the text resulting from an action is not automatically re-scanned to look for matches on the result. This has seemed to be more useful since many of the typical uses of `gema` involve translating from one language or representation to another, so the rules that apply to the input language are not relevant to the output language. Where desired, rescanning can be explicitly invoked as a part of the action by using the notation "`@`*domain*`{`*text*`}`" to re-process the constructed text with the rules of the specified domain.

3 Notation

In the documentation, quotation marks are often used to visually delimit examples that are embedded in the text. The quotation marks are never part of the example. Letters in italics are descriptive place-holders, not literal characters.

3.1 Special characters

By default, the following characters have special meaning in patterns. Note that all of these can be changed through the use of the `@set-syntax` function. See also the `-literal` and `-ml` options.

`*` In a template, this denotes a wild-card argument that matches any number of characters, from zero up to a maximum of 4096, or as specified by the `-arglen` option. (Some limit is needed for efficiency to avoid reading all the way to the end of the file before concluding that the match has failed.) Characters are copied from the input stream into the argument value until a match is found for the entire remainder of the template. Thus, when a template has two or more wild card arguments, the input text is divided among them as necessary for the complete template to be matched. (By contrast, a "`<u>`" argument is similar except that it terminates when a match is found for whatever sequence of literal characters follows it, up until the next argument.) If the `-line` option is in effect or if "`\L`" appeared earlier in the template, then it will not accept a newline character.

In an action, it denotes the value of the corresponding template argument.

`?` Wild-card argument that matches any one character. If the `-line` option is in effect or if "`\L`" appeared earlier in the template, then it will not accept a newline character.

`#` Recursive argument. In a template, this denotes an argument whose value is obtained by translating the input text in the same domain as the current rule until a match is found for whatever sequence of literal characters follows the argument (up to the next argument, or the end of the template, or "`\G`").

In an action, it denotes the value of the corresponding template argument.

`<`*name*`>` Recursive argument, translated according the named domain, or a pre-defined recognizer argument. The name may be empty to denote the default domain. The name does not have to have been defined before it is referenced. This can be used only in a template. If the `-ml` option (new in version 1.4) is in effect, the syntax is instead: `[`*name*`]`

`/`*regexp*`/` In a template, this denotes an argument where the characters between the slashes are used as a regular expression, and the argument value is however much text it matches. If the `-ml` option is in effect, a vertical bar is used instead of a slash. Regular expressions have been documented many other places, so will not be detailed here. Suffice it to say that the following characters and combinations have special meaning:

```
. \ [ ] * + ^ $ \( \) \< \>
```

A slash that is to be part of the regular expression needs to be preceded by a backslash. Regular expression arguments never cross line boundaries. Unlike other kinds of arguments, they will match as many characters as they can, without regard to whatever follows in the template. For example, the template "`a/[a-z]*/x`" will never match anything because if there is an ending "`x`", it will be swallowed by the argument; however, in the template "`a<l>x`" the argument will match on any letter except "`x`".

`=` This designates the end of a template and the beginning of the corresponding action.

`$0` This can be used in an action to copy the matched text to the output. The template is evaluated as though it were an action, with each argument designator being replaced by the actual argument value. Note that this does not necessarily exactly duplicate the input text since any ignored whitespace will be lost and recursive arguments are shown in their translated form.

`$`*digit* or `${`*digits*`}` In either a template or action, this represents the value of the numbered argument. The argument number must be enclosed in braces if it needs more than one digit. In a template, this obviously can only refer to a preceding argument, and in the current implementation, the value of a "`*`" argument cannot be accessed within the same template.

`$`*letter* In either a template or action, this inserts the value of a variable, which is limited to having a name which is a single letter. An error is reported if the variable is not defined.

`${`*name*`}` In an action, this outputs the value of variable. The name is limited to not begin with a digit. An error is reported if the variable is not defined.

`${`*name*`;`*default*`}` In an action, this outputs the value of the named variable, if it is defined, or evaluates the default action if the variable is not defined.

**** Escape character; see the section on "escape sequences" below.

^ Control key. Together with the following character, this represents the control character formed by combining the Control key with the character. For example, either "`^J`" or "`^j`" could be used to denote the ASCII Line Feed character. This notation is not meaningful if the character set being used is not based on ASCII.

Space In a template, a space character matches one or more whitespace characters in the input, the same as "`\S`". (In the less likely event that you really want to match exactly one space character, you can use "`\ `" or "`\s`".) In an action, a space character causes one space to be output if the last character output was not a whitespace character, except that if there are multiple adjacent spaces, all but the first are taken literally. However, if the `-w` option is used, then spaces are ignored except where they server to separate two identifiers.

NewLine The end of a line denotes the end of a rule or immediate action.

; The semicolon is used to separate multiple rules on the same line, and to separate arguments of function calls.

`@`*name*`{`*args*`}` In an action, this notation is used to either call a built-in function or to translate the argument using the rules of the named domain. The name may be empty to denote the default domain. It is permissible to reference a domain name that is defined later in the file. The braces may be optionally omitted for functions that take no arguments.

`@`*spchar* When followed by a special character (i.e. not a letter or digit), the "`@`" indicates that the following character has its default meaning, as documented in this list. This can be used to access the original functionality of a character that has been changed by the `-literal` option or `@set-syntax` function. For example, if you had done "`-literal /`" and then discovered that you do need to use a regular expression, you could write it as "`@/`*regexp*`@/`".

: The characters to the left of the colon (with any leading and trailing spaces and surrounding angle brackets removed) constitute the name of the domain in which the rules that follow on the same line will be defined.

`::` A double colon specifies that the domain whose name appears to the left, inherits from the domain whose name appears to the right.

`!` Comment – the rest of the line is ignored. This can either appear at the beginning of a line to cause the whole line to be ignored, or it can be used at the end of a rule so that the remainder of the line is a comment.

3.2 Escape Sequences

The backslash character denotes special handling for the character that follows it.

- When followed by a lower-case letter or a digit, it represents a particular control character.
- When followed by an upper-case letter, it is a pattern match operator.
- A backslash at the end of a line designates continuation by causing the newline to be ignored along with any leading white space on the following line.
- Before any other character, the backslash quotes the character so that it simply represents itself. In particular, a literal backslash is represented by two backslashes.

Following are the defined escape sequences:

`\a` Alert (a.k.a. bell) character

`\b` Backspace character

`\c`*x* Control key combined with the following character. For example, "`\ci`", "`\cI`", "`^i`", "`^I`", and "`\t`" all have the same effect, namely to represent the ASCII Tab character.

`\d` Delete character

`\e` Escape character (i.e. ESC, not backslash)

`\f` Form feed character

`\n` New line character

`\r` carriage Return character

`\s` Space character

`\t` horizontal Tab character

`\v` Vertical tab character

`\x`*xx* character specified by its heXadecimal code

`\`*digits* character specified by its octal code

`\A` Matches the beginning of the input data, either the beginning of a file or the beginning of the argument for a domain used as a function.

`\B` Matches the beginning of file. This can be used either by itself to specify actions to be taken before beginning to read the file, or it can be used at the beginning of a template that is to match only on the first line of the file.

`\C` This causes case-insensitive comparison for letters in the rest of the template. (See also the `-i` option which selects case-insensitive mode globally.)

`\E` Matches the end of file.

`\G` Goal point. This can be used in a template to indicate the end of the literal string that is used to recognize the end of the preceding argument. For example, if the template "`a(<T>) done`" is applied to the input data "`a(x) b(y) done`", the argument "`<T>`" will match on the text "`x) b(y`", which is probably not what was desired. If the template is written as "`a(<T>)\G done`" then the argument will be terminated by the first right parenthesis, and then the match will fail if the text following the parenthesis doesn't match "` done`". This does not yet work for "`*`" arguments.

If "`\G`" immediately follows a recursive argument, then there is no delimiter, and the argument will continue to accept characters until it stops itself by executing `@end` or `@terminate`.

`\I` Identifier separator. In a template, this matches an empty string if it is not within an identifier. In other words, it requires either of the adjacent characters to not be an identifier constituent in order for the template to match. In an action, this outputs a space character if the last character output is an identifier constituent. By default, an identifier constituent is a letter, digit, or underscore, but this can be extended by the `-idchars` option.

`\J` Join – locally counteracts the `-w` and/or `-t` option by saying that spaces in the input will not be ignored at this position, and an identifier delimiter is not required here. If neither of these options is being used, then it has no effect. Not meaningful in an action.

`\L` Line mode – arguments that follow in the same template are not allowed to cross line boundaries. This also means that "`\S`" and "`\W`" will not accept newline characters. However, a line boundary can still be crossed by an explicit "`\n`" or "`\N`".

`\N` New line boundary. In a template, this matches an empty string if it is at either the beginning of a line or the end of a line (either before or after a new line character, or at the beginning or end of the file or data stream). In an action, it outputs a new line character if the last character output is not a new line.

`\P` Position – if the template matches, the input stream will be left at this position. Thus everything following this is a look-ahead, and will be re-read for subsequent pattern matches.

`\S` Space. In a template, this matches one or more whitespace characters. (See also "`<S>`" which has the same effect except that the spaces are remembered as an argument value.) In an action, it outputs one space character if the last character output is not a whitespace character.

`\W` Optional whitespace. In a template, this specifies that any whitespace characters in the input stream at this point will be skipped over. (See also "`<s>`" which has the same effect except that the spaces are remembered as an argument value.) However, if this is followed in the template by a literal whitespace character, then that character will not be skipped. For example, in "`\W\n`", the "`\W`" will skip any whitespace other than a newline. This has no effect in an action. See also the `-w` option which ignores spaces everywhere.

`\X` Word separator. In a template, this matches an empty string if it is not within a word. In this context, a word consists of letters and digits.

`\Z` Matches the end of the input data, either the end of a file or the end of the argument for a domain used as a function, or a look-ahead match of the terminating string for a recursive argument.

3.3 Recognizer arguments

The following argument designators, consisting of a single letter between angle brackets, can be used in templates to match on various kinds of characters. Preceding the letter with "`-`" inverts the test. The argument requires at least one matching character if the letter is uppercase, or is optional if the letter is lowercase. The letter may be followed by a number to match on that many characters, or up to that maximum for an optional argument. If the number is `0`, the argument matches if the next character is of the indicated kind, but the input stream is not advanced past it; in other words, this acts as a one-character look-ahead.

If the argument is followed in the template by literal characters, then the argument will be terminated when that literal string is matched, even if those characters would otherwise qualify for inclusion in the argument.

`<A>` Alphanumeric (letters and digits)

`<C>` Control characters

`<D>` Digits

`<F>` File pathname. See the `-filechars` option.

`<G>` Graphic characters, i.e. any non-space printable character

`<I>` Identifier. By default, an identifier consists of letters, digits, and underscores. See the `-idchars` option.

`<J>` lower case letters (in version 1.2 or later)

`<K>` upper case letters (in version 1.2 or later)

`<L>` Letters (either upper or lower case)

`<N>` Number, i.e. digits with optional sign and decimal point

`<O>` Octal digits

`<P>` Printing characters, including space

`<S>` white Space characters (space, tab, newline, FF, VT)

`<T>` Text characters, including all printing characters and white space

`<U>` Universal (matches anything except end-of-file)

`<W>` Word (letters, apostrophe, and hyphen)

`<X>` hexadecimal digits

`<Y>` punctuation (graphic characters that are not identifiers)

4 Built-in Functions

There are a large number of built-in functions that can be used in actions. Function calls have the form "`@`*name*`{`*args*`}`", with arguments separated by "`;`". For functions without arguments, the argument delimiters "`{}`" may be omitted if they are not needed to separate the name from the following character.[2] Each argument is itself an action which can use any of the special characters defined in actions, including nested function calls. The argument value is the result of evaluating the argument. In a few cases, arguments that are not used are skipped instead of being evaluated, but arguments are never used literally. All functions take a fixed number of arguments, although in a couple of cases the last argument is optional.

The descriptions of the functions use the terminology of a value being returned by the function, but it would be more accurate to speak of the result as being the series of characters that will be written to the current output stream, since in general the result value is not actually materialized as a separate string. Usually while evaluating a function argument, the current output stream is an internal buffer that collects the argument value for the function, and the function argument is actually an input stream that reads from that buffer. But in most cases these distinctions are not important for understanding how to use the functions.

The following sections document groups of related functions.

4.1 Numbers

Since **gema** is a text processor, it is not intended to be convenient or efficient for performing numeric operations, but it does have a set of arithmetic functions that should be sufficient to make it possible to do whatever calculations are necessary.

While all values are character strings, a string can be treated as a number if it consists of decimal digits optionally preceded by `+` or `-` and optionally preceded or followed by spaces. Where a numeric argument is required, such a string will be converted internally to a 32-bit signed integer. An error will be reported if the string is not a valid number. Functions that return a number will return a string of decimal digits possibly preceded by a minus sign.

Following are the arithmetic functions:

`@add{`*number*; *number*`}` Addition – returns the sum of the two numbers.

`@sub{`*number*; *number*`}` Subtraction – returns the first argument minus the second.

`@mul{`*number*; *number*`}` Multiplication – returns the product of the two numbers.

`@div{`*number*; *number*`}` Division – returns the quotient of dividing the first argument by the second.

`@mod{`*number*; *number*`}` Modulus – returns the first argument modulo the second, as implemented by the C operator "`%`".

Also, the following group of functions can be used to operate on numbers as bit strings:

`@and{`*number*; *number*`}` Returns the bit-wise *and* of the two numbers.

`@or{`*number*; *number*`}` Returns the bit-wise *or* of the two numbers.

`@not{`*number*`}` Returns the bit-wise inverse of the argument.

[2]Empty braces might mean either no argument, or a single argument which is an empty string. This potential ambiguity is not a problem because there aren't any functions that take a single optional argument.

Finally, some other assorted functions that deal with numbers:

`@cmpn{`*number*`;`*number*`;` *less-value*`;`*equal-value*`;`*greater-value*`}` Compare numbers – returns the result of evaluating either the third, fourth, or fifth argument depending on whether the first argument is less than, equal to, or greater than the second, when compared as 32-bit signed numbers. The two arguments that are not used are not evaluated. For example, the following rule defines a function that will return the larger of two comma-separated numbers:

```
maxn:<N>,<N>=@cmpn{$1;$2;$2;$1;$1}
```

while the following rule sets a variable to the largest number seen:

```
notemax:<N>=@cmpn{$1;${max};;;@set{max;$1}}
```

`@int-char{`*number*`}` Returns the character whose internal code is given by the argument.

`@char-int{`*character*`}` Returns the decimal number representation of the internal character code of the argument, which should be a single-character string.

`@radix{`*from*`;`*to*`;`*value*`}` Radix conversion. The first two arguments must be decimal integers. The third argument is interpreted as a number whose base is specified by the first argument. The result value is that number represented in the base specified by the second argument. As currently implemented, *from* may be any number from 2 to 32, but *to* can only be one of 8, 10, or 16. For example, octal constants in a C program could be converted to hexadecimal form by the following rule:

```
\I0<O>\I=0x@radix{8;16;$1}
```

For hexadecimal output, upper case letters are used for the digits greater than 9. If lower case letters are desired, the `@downcase` function can be used on the value returned by `@radix`.

4.2 String functions

The following built-in functions perform various manipulations on character strings.

4.2.1 Output formatting — padding, filling, and wrapping

The following group of functions take two arguments; the first must be a number and the second is an arbitrary string. If the length of the string is greater than the number, then it is returned unchanged. Otherwise, the returned value will consist of the string padded with spaces to be of the designated length. The choice of function determines how the padding is done:

`@left{`*length*`;`*string*`}` Left-justify the string, padding with spaces to the designated length. For example, "`@left{8;ab}`" returns "`ab`" followed by 6 spaces, while "`@left{8;hippopotamus}`" returns "hippopotamus" with no spaces. If you want long values to be truncated, you can use:

`@left{`*length*`;@substring{0;`*length*`;`*string*`}}`

or write something like "`@left{8;@cut8{`*arg*`}}`", accompanied by the rule: "`cut8:<U8>=$1@end`".

`@right{`*length*`;`*string*`}` Right-justify the string, padding with spaces to the designated length.

`@center{`*length*`;`*string*`}` Center the string within a field of the designated length.

Note that any of these functions can also be used with an empty second argument as a convenient way to generate a particular number of spaces.

The following group of functions serve a similar purpose, except that padding can be done using any arbitrary string instead of spaces. Here the first argument is the string representing an empty field, and the second argument will be justified within that field.

`@fill-left{`*background*`;` *value*`}` Left-justify the value on top of the background string. For example, "`@fill-left{......;foo}`" returns "`foo...`".

`@fill-right{`*background*`;` *value*`}` Right-justify the value on top of the background string. For example, "`@fill-right{00000;12}`" returns "`00012`".

`@fill-center{`*background*`;` *value*`}` Center the value on top of the background string. For example, "`@fill-center{((()));xy}`" returns "`((xy))`".

The following functions perform formatting based in the current context in the output stream:

`@tab{`*number*`}` The return value consists of however many space characters it takes to advance the output stream to the specified column number. If the output stream is already at or beyond the specified column, the return value is empty. Column 1 means the first character position following a newline character or the beginning of the data stream. Thus, for example, if the last character output was a newline, then `@tab{10}` will return 9 space characters so that the next character written will go in column 10.

`@wrap{`*string*`}` Output with line wrapping. If there is room for the string on the current line of output, then it will be returned unchanged. Otherwise, when the string is longer than the remaining space on the current line, the return value consists of a newline character followed by an optional indentation string followed by the string argument with any leading whitespace removed. However, if the output stream is already at the beginning of a line, then the return value is the indentation string followed by the argument string with leading whitespace removed. By default, the lines are up to 80 characters long and the indentation string is empty. These parameters can be changed by the `@set-wrap` function below. Typically the argument string will be a word preceded by a space character, so that the space will separate it from the previous word if it fits on the current line, or will be discarded if a new line is started.

For example, you could reformat a text file with the shell command:

```
gema -p '<G>=@wrap{ $1};\n\W\n=\n\n;\S=;' in.text out.text
```

where the first rule causes the groups of graphic (non-space) characters to be written separated by a single space in 80-character lines, the second rule preserves blank lines as paragraph separators, and the third rule discards other whitespace characters.

`@set-wrap{`*number*`;` *string*`}` For subsequent calls to `@wrap`, the first argument specifies the maximum number of characters in a line, and the second argument is the indentation string. No value is returned. For example, for output with a four character left margin followed by a maximum of 70 characters of text, do: "`@set-wrap{74;\s\s\s\s}`"

4.2.2 String Comparison

The following functions compare two strings, and then returns the value of one of three arguments depending on the result of the comparison. The two arguments that are not used are not evaluated, so these functions can be used for conditional evaluation of side-effects as well as for returning a value.

`@cmps{`*string*`;` *string*`;` *less-value*`;` *equal-value*`;` *greater-value*`}` Compare strings, case-sensitive. The comparison is performed by the C function `strcmp`. The returned value is either the third, fourth, or fifth argument depending on whether the first argument is less than, equal to, or greater than the second.

`@cmpi{`*string*`;`*string*`;`*less-value*`;`*equal-value*`;`*greater-value*`}` Compare strings, case-insensitive. The comparison is performed by the C function `stricmp`.

4.2.3 Case conversion

The following functions return a copy of their argument, converting the case of any letters:

`@upcase{`*string*`}` Convert any lower case letters to upper case.

`@downcase{`*string*`}` Convert any upper case letters to lower case.

For example, the following rule will capitalize each word in the input data:

```
<L1><w>=@upcase{$1}@downcase{$2}
```

4.2.4 Miscellaneous string functions

`@length{`*string*`}` Returns the length of the argument as a decimal number. For example,"`@length{abcdefghijkl}`" returns the string "12".

`@reverse{`*string*`}` Returns the characters of the argument in reversed order. For example, "`@reverse{abcd}`" returns "`dcba`". This may be useful for performing processing that needs to be done from right to left. For example, the following set of rules will insert commas in the proper position in all numbers of four or more digits, grouping the digits by threes from the right-hand end:

```
<D3><D>=@reverse{@comma{@reverse{$1$2}}}
comma:<D3><D0>=$1,
```

`@substring{`*skip*`;`*length*`;`*string*`}` Returns a substring of the third argument formed by skipping the number of characters indicated by the first argument and then taking the number of characters indicated by the second argument. For example, "`@substring{3,4,elephant}`" returns "`phan`". If the first argument is negative, the effect is the same as zero. If the first argument is greater than the length of the string, then the result value is empty. The result may actually be shorter than *length* if there are not enough characters in the string: "`@substring{3;99;tiger}`" returns "`er`".

Note that splitting input data into fields is usually more conveniently done by using a template such as "`\L<U2><U3><u>\n`". The `@substring` function is more likely to be useful in cases where the numbers are computed instead of being constants.

`@repeat{`*number*`;`*action*`}` The second argument is repeated the number of times specified by the first argument. For example, a string of eighty hyphens can be constructed by "`@repeat{80;-}`".

While this is being listed under string functions because it doesn't seem to fit any other category, it is useful for much more than just repeating strings. Rather than just repeating the value, the second argument is an action which is evaluated the specified number of times, so it can have side-effects which are also performed repeatedly. If the number is less than or equal to zero, the second argument is not evaluated at all. For example, the following action will output the numbers from 1 to 100:

```
@set{n;0}@repeat{100;@incr{n} $n}
```

Note that there is no operator or function needed for concatenation of strings since concatenation of elements is implied simply by juxtoposition.

4.3 Variables

A *variable* consists of a *name* and an associated *value*, both of which are character strings. Variable names are case-sensitive (regardless of the `-i` option). The value can contain any of the 256 possible characters, and the name can contain any characters except for NUL. Names consisting of a period followed by upper case letters are by convention reserved for internal use. Both strings may be of any length, limited only by the amount of memory available. Variables are manipulated by using the following action functions. Except for `@var`, they have no return value.

`@set{`*name*`;`*value*`}` Set the named variable to the designated value. If the variable was already defined, the previous value is discarded. For example, "`@set{count;0}`" initializes variable "`count`" to 0.

`@var{`*name*`}` Returns the current value of the named variable. If the variable is not defined, an error is reported and the return value is unspecified.[3]

`@var{`*name*`;`*default*`}` If the named variable is defined, then its current value is returned and the second argument is skipped without being evaluated. Otherwise, when the name is not defined, the return value is the result of evaluating the second argument.

`@append{`*name*`;`*string*`}` The string is appended to the end of the value of the named variable. If the variable was not previously defined, then this acts the same as `@set`. For example, "`@append{buf;$1}`" has the same effect as "`@set{buf;@var{buf;}$1}`", but using `@append` is considerably more efficient.

`@incr{`*name*`}` The value of the named variable is incremented by one. You might think of "`@incr{n}`" as being an abbreviation for
" `@set{n;@add{@var{n};1}}`"
but it is actually more general than that. The value may contain arbitrary characters before or after the number, and the number will be incremented while leaving the other characters unchanged. For example, if the value is "`B9a`", it will be incremented to "`B10a`". The value can also be just one or more letters, in which case the last letter will be incremented to the following letter; for example, "`a`" increments to "`b`". and "`z`" increments to "`aa`".

`@decr{`*name*`}` The value of the named variable is decremented by one. This works like `@incr` except that the increment is -1 instead of $+1$, and decrementing a value of "`a`" is an error.

`@bind{`*name*`;`*string*`}` Sets the value of the named variable to the string. If the variable was already defined, the previous value is remembered so that it can be restored by a subsequent call to `@unbind`. If `@bind` is called in the context of a recursive argument for a template match that subsequently fails, then the binding will be undone automatically.

`@unbind{`*name*`}` The named variable is restored to the value it had before the most recent `@bind`. It it had not been defined before the `@bind`, then it becomes undefined again. An error is reported if the variable is undefined or if there is no pending binding.

`@push{`*name*`;`*string*`}` A variable may be thought of as a stack of values, where `@var` accesses the top-of-stack value, `@push` pushes a new value onto the top of the stack, `@set` modifies the top value, and `@pop` pops the top value off the stack. `@push` is actually just another name for `@bind`.

[3]Currently, the name is returned, but that should not be relied on.

`@pop{`*name*`}` The variable is restored to the value it had before the most recent `@push`. This is actually just another name for `@unbind`. The top-of-stack value is simply discarded; there is no return value.

Also, `@var` (both the one and two argument forms) may be abbreviated as `$`, providing that the name does not begin with a digit. Thus, for example, "`${foo}`" has the same meaning as "`@var{foo}`". Furthermore, if the variable name is a single constant letter and there is no default value argument, then the braces may be omitted. Thus, "`@var{i}`" can be abbreviated as "`$i`". This last form (dollar letter) also has the special property that it can be used in a template to insert the current value of a variable into the template to be matched. All of the other variable operations can only be used in actions.

Lisp programmers may find it helpful to note that `@set` is like the Lisp `set` form, `@var` is like `symbol-value`, and the combination of `@bind` and `@unbind` is like what happens in a `let` for a variable with dynamic scope.

While there is no support for arrays as such, note that since the name of a variable can contain any characters, and the name is an evaluated argument, it is possible to do things like "`@set{A[$i];$1}`" which looks like an array and can be used like an array, even though the brackets and subscript are really just part of the variable name.

Variables can also be used as an associative look-up table, where the name is the key. However, the current implementation assumes that the number of variables will be small, so it may become slow if used as a table with a large number of entries.

4.4 Files

4.4.1 Pathname manipulation

This group of functions allow constructing pathnames in a manner that allows a pattern file to be independent of the pathname syntax for a particular operating system.

`@makepath{`*directory*`;`*name*`;`*suffix*`}` Returns the file pathname formed by merging the file name in the second argument with the default directory in the first argument and replacing the suffix from the third argument, if not empty. If the second argument is an absolute pathname, then it retains the same directory and the first argument is not used. For example (assuming running on Unix):

```
@makepath{/home/dir;bar.c;.o} ⇒ /home/dir/bar.o
@makepath{/home/dir;/scr/bar.c;.o} ⇒ /scr/bar.o
@makepath{/home/dir;bar.c;} ⇒ /home/dir/bar.c
```

`@mergepath{`*pathname*`;`*name*`;`*suffix*`}` Returns the file pathname formed by merging the second argument with a default directory extracted from the first argument and replacing the suffix from the third argument, if not empty. This differs from `@makepath` in that the first argument is a complete file pathname whose name portion is ignored. This would be used to create a new file in the same directory as another file. For example (assuming running on Unix):

```
@mergepath{/a/foo.i;bar.c;/a/baz.o} ⇒ /a/bar.o
@mergepath{/a/foo.i;/b/bar.c;.o} ⇒ /b/bar.o
@mergepath{/a/foo.i;bar.c;} ⇒ /a/bar.c
```

`@relative-path{`*pathname*`;`*pathname*`}` If the two pathnames have the same directory portion, return the second argument with the common directory removed; else return the whole second argument. Note that if the two arguments are the same, this has the effect of separating the file name from the directory. For example:

`@relative-path{/a/x/cat.x;/a/x/dog.c}` ⇒ `dog.c`
`@relative-path{/a/x/cat.x;/a/y/dog.c}` ⇒ `/a/y/dog.c`

`@expand-wild{`*pathname*`}` Usually this function just returns its argument followed by a newline. When running on MS-DOS or Windows and the *pathname* is a wild card (i.e. contains "`*`" or "?"), the return value consists of all files that match the pattern, with a newline following each one. If there are no matches, a warning is written to the error output and the return value is empty. This wild card expansion is done by a system call, so it is consistent with other MS-DOS or Windows command-line utilities, but the meaning of "`*`" is not completely the same as in **gema** patterns. On Unix, wild card arguments are presumed to have already been expanded by the shell, so expansion is not done here.

4.4.2 Using alternate input and output files

`@err{`*string*`}` The argument is evaluated with its output being directed to the standard error output stream (`stderr` in C terminology). There is no return value. This can be used to write error messages or status messages. Don't forget that newlines must be explicitly provided, so the argument typically needs to end with "`\n`".

`@out{`*string*`}` The argument is evaluated with its output being sent directly to the current output file instead of to the current output stream. The distinction arises during translation of a recursive argument, where `@out` can be used to write directly to the output file instead of appending to the value of the argument being translated. Usually this is not what you want to do, but it may be useful in some circumstances.

For example, suppose some algebraic language is to be translated into an assembly-like language. A typical rule might look something like:

```
expr:<term>+<term>=@incr{t}@out{\N  ADD $1,$2,R$t\n}R$t
```

where an expression "`x+y`" would be processed by outputting "`  ADD x,y,R1`" and returning "`R1`" as the result value to be used as an operand of the next instruction.

`@write{`*pathname*`;`*string*`}` First the first argument is evaluated and the file that it names is opened for writing. If the pathname is "`-`", then standard output will be used. If the same identical pathname has previously been used in a `@write` call, then it will continue writing to the end of the same file without re-opening or rewinding it.

Then the second argument is evaluated, with its output being directed to the designated file. Within that evaluation, the function `@outpath` will return the first argument of the `@write`.

The file remains open until either the program terminates or the same pathname is referenced in a call to `@close` or `@read`.

`@close{`*pathname*`}` If the argument is identical to one previously appearing as the pathname argument in a call to `@write`, then that output file will be closed. Otherwise, nothing happens.

`@read{`*pathname*`}` The file named by the argument is opened for reading. If the *pathname* is "`-`", then standard input is used. If the same identical pathname was previously used in a `@write`, the output file will be closed before re-opening the file for reading. The result is an input stream that will read from the file as needed, and close it when the end is reached. This is commonly used in the context: "`@`*domain*`{@read{`*pathname*`}}`" which says to translate using the alternate file as input. Within this translation, the functions `@file`, `@inpath`, `@line`, `@column`, and `@file-time` will all refer to the file named in the argument of `@read`. However, if the `@read` function has its result concatenated with something else instead of appearing by itself as the argument to another function, then the effect will be to copy the entire contents of the file to the current output and close the file.

`@probe{`*pathname*`}` This can be used to test a pathname to see whether it can be opened. The result value is "`F`" if the argument names an existing file, "`D`" if it names a directory, "`V`" if it names a device, "`U`" if it is undefined, or "`X`" if it is defined in some unexpected way.

4.4.3 File context queries

`@outpath{}` Returns the pathname of the current output file, or as much of the pathname as is known. This would be the same as the output file argument on the command line or the pathname argument to the `@write` function if within that context.

`@inpath{}` Returns the pathname of the current input file, or as much of the pathname as is known. This would be the same as the input file argument on the command line or the pathname argument to the `@read` function.

`@file{}` Returns the name of the current input file, with any directories removed.

`@line{}` Returns the current line number in the input file. More precisely, this is the line number of the last character matched by the template. If that character is a newline character, this is the number of the line preceding the newline.

`@column{}` Returns the column number of the current position in the input stream, i.e., the column of the last character in the text matched by the template. For example, the following default rule could be used to write an error message for unexpected input:

```
?=@err{\NIllegal character "$1" in line @line, column @column.\n}
```

`@out-column{}` Returns the column number of the current position in the current output file.

`@file-time{}` Returns the date and time when the current input file was last modified. The information is presented as formatted by the C function `ctime`, except without any newline.

4.5 Control flow functions

`@end{}` Signals the successful completion of the current translation.

If this appears in the action for a pattern match at the top level of a file, the remainder of the input file will not be read, and if there are no more input files to be processed, the program will terminate with an exit status of 0 (assuming there were no errors before). For example, the following shell command will print the first line that matches and then stop:

```
gema -match -p 'Title\:*\n=$0@end' foo
```

If this appears within the context of a recursive argument, then it ends the argument and returns control to the enclosing template to continue processing the input. For example, with the following rules:

```
sign:+=+@end;-=-@end;=@end
```

the template argument “`<sign>`” will accept an optional plus or minus sign and nothing more.

`@fail{}` Signals failure of the current translation. At top-level, this will terminate processing of the input file like `@end`, except that the program will have a non-zero exit status. For example, the following command will indicate by the exit status whether the file being tested contains a particular string:

```
gema -match -p 'Success=@end;\E=@fail' foo.text
```

If the string is found, the program exits with 0; if the end of the file is reached, then a non-zero exit status is returned.

Within the context of a recursive argument, this causes the enclosing template to report a failed match.

`@terminate{}` This ends the translation of a recursive argument. If the argument value is empty, then the template match fails, like for `@fail`. Otherwise, when some characters have been accepted, processing of the template continues like for `@end`. This is typically used instead of `@end` in a delimiter rule when an empty string is not be be considered a match. For example, with the following rules:

```
vowel:a=a;e=e;i=i;o=o;u=u;=@terminate
```

the argument “`<vowel>`” will match one or more vowels.

`@abort{}` Immediately terminates execution of the program with a non-zero exit status.

`@exit-status{`*number*`}` This function can be used to specify that a particular exit status value will be returned when the program exits, providing that there is no error condition that specified a higher value first. This could be called before `@fail` or `@abort` to cause some particular non-zero value to be returned for the sake of a shell script that wants to test for what kind of failure occurred.

4.6 Other operating system interfaces

`@date{}` Returns the current date, in the form: *mm*/*dd*/*yyyy*

`@datime{}` Returns the current date and time, as formatted by the C function `ctime`, except without any newline.

`@time{}` Returns the current time, in the form: *hh*:*mm*:*ss*

`@getenv{`*name*;*default*`}` Returns the value of an environment variable, as from the C function `getenv`. The first argument is the name of the environment variable. The second argument is optional, and will be returned as the default value if the environment variable is not defined. For example, on a Unix system, the action “`@getenv{USER}`” will output the current user ID.

`@shell{`*string*`}` The argument value is executed as a shell command by passing it to the C function `system`. Although it would be desirable for this to return the text written by execution of the command, that is not currently implemented. Instead, any output from the command goes directly to standard output, and there is no value returned to the current output stream. For example, the following action could be used to sort a temporary file:

```
@shell{sort \< '${tmpfil1}' \> '${tmpfil2}'}
```

4.7 Definitions

The following functions can be used to add or remove definitions of rules at run time.

`@define{`*patterns*`}` Define new rules. The evaluated argument value is read as a pattern file, defining rules and performing immediate actions as specified. There is no return value. For example, you can have one pattern file include another by using an immediate action like this:

```
@define{@read{foo.pat}}
```

For another example, to emulate a C pre-processor, the `#define` directive could be implemented by the following rule (assuming, for simplicity of the example, no arguments, no continuation lines, and no comments):

```
\N\#define <I> *\n=@define{\\I$1\\I\=@quote{$2}}
```

Note that the tricky part here is to get the right level of quoting so that things are evaluated at the proper time. The `@quote` function is explained below. Given the input line "`#define NUM 34`", the argument of `@define` will evaluate to the string "`\INUM\I=34`" which will then be defined as a new rule.

`@quote{`*string*`}` Returns a copy of the argument value with backslashes inserted where necessary so that `@define` and `@undefine` will treat all of the characters as literals.[4] For example, given an argument which evaluates to the string "`a * 3`", the return value will be "`a\ \*\ 3`".

`@undefine{`*patterns*`}` This can be used to undefine rules. The argument is processed like for `@define`, except that instead of defining rules, the effect is to cancel any existing rule that exactly matches. The argument may also be just a template, without any "`=`" or action, in which case any rule with the same template will be cancelled, without regard to its action. For example, the C `#undef` directive could be emulated (with the same simplifying assumptions as the `#define` example above) by the rule:

```
\N\#undef <I>=@undefine{\\I$1\\I}
```

`@subst{`*patterns*`;`*operand*`}` Substitution. Return the result of translating the operand according to the patterns specified by the first argument. The first argument is processed the same as by `@define`, except that the rules are implicitly defined in a temporary domain which is deleted after being used to translate the operand. An explicit domain name (i.e. before a colon) is not allowed. For example, "`@subst{\\Iis\\I\=was;this is it}`" will return "`this was it`". Usually this sort of substitution is more conveniently and efficiently done by using a domain

[4]Lisp programmers should not confuse `@quote` with the Lisp `quote` form; rather this is like the notion of *printing with slashification.*

function (for example, "`@frob{this is it}`" with rule "`frob:\Iis\I=was`"), but the `@subst` function can be used in cases where the substitution needs to be computed at run time. For example, to emulate a C `#define` directive with one argument:

```
\N\#define <I>\W(\W<I>\W) *\n=\
  @define{\\I$1\\W(\#)\=@subst{\\I$2\\I\=\\\$1;@quote{*}}}
```

Here `@subst` is used to replace references to the C argument name with the "`$1`" notation used by **gema**.

4.8 Setting Options

The following group of functions can be used to set various program options. These are typically used as immediate actions in a pattern file to set the options that the file needs, instead of requiring separate command line options. None of these functions return any result value.

`@set-switch{`*name*`;`*value*`}` Sets the value of any of several option switches that have numeric values. In most cases the value should be either 1 for true or 0 for false. The defined switch names are:

`arglen` – maximum length for "`*`" operands. This is the only switch that takes a number rather than being just true or false.

`b` – binary mode

`i` – case-insensitive mode

`k` – keep going after errors

`line` – line mode

`match` – match only mode

`t` – token mode (However, this is only part of the "`-t`" command line option, which is implemented by "`@set-switch{t;1}@set-switch{w;1}`".)

`trace` – write pattern match diagnostic messages to `stderr`. (Only recognized if the program was compiled with "`-DTRACE`".)

`w` – ignore whitespace in the input (This is only part of the "`-w`" command line option, which is implemented by "`@set-switch{w;1}@set-syntax{S;\s\t}`".)

In each case, the switch corresponds to the command line option with the same name, and further explanation of the meaning can be found there.

`@get-switch{`*name*`}` Returns the current value of the named switch.

`@set-parm{`*name*`;`*value*`}` Sets the value of any of several options that have string values. The defined names are "`idchars`", "`filechars`", and "`backup`". These are used to implement the command line options with the same names, and the meaning is documented there.

`@set-syntax{`*type*`;`*charset*`}` This function can be used to change the meaning of characters in patterns. When used as an immediate action in a pattern file, it takes effect beginning with the next line read. The first argument designates one or more syntactic categories, and the second argument is a set of characters that will now have that meaning. For each character in the first argument, the corresponding character in the second argument acquires the designated syntactic class; when there is only one remaining character in the first argument, it applies to all of the remaining characters in the second argument. The syntactic class may be identified by either the special character which currently has that class (or which has that class by default if it is currently a literal), or by one of the following letters:

`A` – argument separator. This is one of the two uses of the semicolon in the default syntax. For example, if you wanted to be able to use comma to separate arguments, do: “`@set-syntax{A;,}`”

`C` – comment. Causes the rest of the line to be ignored as a comment.

`D` – domain argument. Characters of this class can be used as an abbreviation for a domain argument with the same name. There are no characters that have this class by default. For example, if you say “`@set-syntax{D;%}`”, then the character “`%`” represents a recursive argument in the domain defined by rules prefixed by “`%:`”. In other words, “`%`” becomes an abbreviation for “`<%>`”, and it also can be used in an action to represent the value of the corresponding argument, like with “`*`”, “`?`”, and “`#`”.

`E` – escape. Together with the following character, it specifies a control character or template operator. This is half of what the backslash does in the default syntax.

`F` – function prefix. This introduces the name of a function to be called. This is one of two uses of “`@`” in the default syntax.

`I` – ignore. Characters with this class will be completely ignored. There are no characters that have this class by default.

`K` – character operator. Causes the following character to have its default meaning. This is one of the two things that “`@`” is used for in the default syntax.

`L` – literal. For example, the command line option “`-literal '/?^'`” is implemented by: “`@set-syntax{L;\/\?\^}`”

`M` – quote until match. Causes the following characters to be taken literally until a second occurrence of the character is found. For the sake of compatibility with earlier versions, there are currently no characters that have this class by default. For example, “`@set-syntax{M;\'}`” causes all characters between matching apostrophes to be taken literally (even backslash).

`Q` – quote one character. Causes the following character to be taken literally. This is half of what the backslash does in the default syntax.

`S` – ignored space. Characters with this class will be ignored unless they separate two identifiers, in which case they will be treated like “`\S`”. There are no characters that have this class by default, but part of what the `-w` option does is: “`@set-syntax{S;\s\t}`”

`T` – terminator. A character with this class marks the end of a rule. By default, newline is used for this purpose.

For example, “`@set-syntax{\*;\~}`” would cause tilde to represent a wildcard argument. This doesn't change the meaning of the asterisk, it just means that now either of the characters can be used for that purpose. If you wanted to delimit recursive arguments with square brackets, and let angle brackets be literals, do: “`@set-syntax{\<\>LL;\[\]\<\>}`”

`@reset-syntax{}` Re-initializes the syntax tables to their default state, thus undoing the effects of any calls to `@set-syntax`, including any use of the `-literal` option.

`@set-locale{`*name*`}` Set the internationalization locale, using the C function `setlocale`. A typical usage would be:

```
@set-locale{@getenv{LANG;C}}
```

This may affect which characters are considered to be letters. Currently this has no effect on MS-DOS.

4.9 Informational functions

`@show-help{}` Displays on the standard error output a brief explanation of how to use the program. No value is returned. This is used internally to implement the `-help` option, and is not likely to be of use otherwise. The message is constructed based on the current syntax tables, so use of the `@set-syntax` function will be reflected here.

`@version{}` Returns the program version identification string. This is used internally to implement the `-version` option, and is not likely to be of use otherwise.

5 Customized command-line processing

Part of the intent of **gema** is that it can be used as a means of implementing more specialized tools. A utility program is defined by the command line arguments that it uses as well as by how it processes its input files. Therefore, **gema** provides a way to customize the handling of command line arguments.

The main program of **gema** just does some initialization, and then processes the command line arguments by translating them with a set of built-in patterns. These rules that define the command line arguments are defined in a domain named "`ARGV`". The user is free to add additional rules to this domain, thereby implementing new command line options, or even to undefine existing rules. In the input stream that is translated by the `ARGV` domain, the command line arguments are separated by newline characters.[5] The actions for the `ARGV` rules are expected to do all their work with side-effects and to not return any value. Any value that is returned by the translation (except for the delimiting newlines) will be reported by the main program as undefined arguments.

The complete set of built-in `ARGV` rules can be seen by looking at the source file "`gema.c`" in the variable `argv_rules`. Here are a few representative examples:

```
ARGV:\N-idchars\n*\n=@set-parm{idchars;$1}
ARGV:\N-literal\n*\n=@set-syntax{L;$1}
ARGV:\N-p\n*\n=@define{*}
ARGV:\N\L*\=*\n=@define{$0}
ARGV:\N-odir\n*\n=@set{.ODIR;*}
ARGV:\N-<L1>\n=@set-switch{$1;1}
ARGV:\N-*\n=@err{Unrecognized option\:\ "-*"\n}@exit-status{3}
```

For an example of extending the command line options, suppose you wanted to emulate a C pre-processor by accepting "`-D`" options to define macros. That could be done by defining rules such as:

```
ARGV:\N-D<I>\=*\n=@define{\\I$1\\I\=@quote{$2}}
ARGV:\N-D<I>\n=@define{\\I$1\\I\=1}
```

Instead of adding to the built-in rules, it is also possible to suppress the built-in rules and define your own rules from scratch. To do this, start the program with a command line like:
`gema -prim` *pattern-file ...*
The `-prim` ("primitive mode") option suppresses loading of the built-in rules and reads patterns from the specified file. Then the remainder of the command line is processed according to whatever `ARGV` rules were defined in that file. Note that even the default behavior of reading from standard input and writing to standard output is implemented by the `ARGV` rules. (The `-prim` option is the only one that is hard-coded instead of being implemented by patterns.)

[5]The newline was chosen for convenience, but it would more exactly emulate the C argument semantics if the NUL character was used as the separator, and perhaps that ought to be done in the future.

6 Exit codes

When the program terminates, it will return one of the following status codes to the operating system (unless overridden by the use of function `@exit-status`):

0 nothing wrong

1 (reserved for user via `@exit-status{1}`)

2 failed match signaled by `@fail` or `@abort`

3 undefined command line argument

4 syntax error in pattern definitions

5 use of undefined name during translation (domain, variable, switch, parameter, syntax type, or locale)

6 invalid numeric operand

7 can't execute shell command for `@shell` function

8 I/O error on input file

9 I/O error on output file

10 out of memory

7 Status and Future development

This program was essentially functionally complete by the end of 1995. There have been only minor enhancements and bug fixes since then, both because it had reached a point where it was sufficient for my own needs and because I have had very little time to spend on further development in recent years.

I consider this to have been a succesful experiment since this program continues to prove very useful for a wide variety of tasks that are not as well served by other tools.

In an ideal world, the current program would be regarded as a completed prototype, and it would be appropriate to start designing the real program to replace it. However, as usually happens in the real world, we ship the prototype because there isn't time to do any more. There is room for improvement in the areas of consistency, ease of use, and performance at least. One particular design issue is that the pattern matching should build a multi-level decision tree instead of using just a two-level dispatch with linear search after that.

Since this was developed by one person as a spare time hobby, it has not had as extensive or systematic testing as could be done, but I continue to use it frequently and it has been used by a number of other people over the years, with only a small number of bugs being discovered.

I don't know whether I will be spending any more effort on further development, but I am interested in hearing about any bugs found or other suggestions.

Following, in no particular order, are some assorted ideas for enhancements which remain for the future:

- Should warn about a domain that is defined but not referenced, since it it easy to mistakenly neglect to quote a colon.
- It might be useful to have a way to switch (or *push* and *pop*) the output file – e.g. to write each chapter of a document to a separate file even though the input might be a single file.
- A function to construct a unique pathname for a temporary file.
- A default notation for quoting a long section of literal text, in addition to using the backslash for quoting individual characters.
- A function to return the pathname of the current directory.
- Record the file and line that each rule came from, to be used in run-time error messages.
- Improved trace mode as an aid for debugging pattern files.
- A template operator for specifying an action to be taken after all input files have been processed.
- A variation of the `@shell` function that returns the output of the command.
- Support for Unicode characters.

8 Acknowledgments

This program was conceived as an extension of the concepts embodied in W. M. Waite's "STAGE2" processor[6], as implemented by Roger Hall.[7]

[6] *Communications of the ACM*, July, 1970, page 415ff

[7] Under the name "TILT" ("Texas Instruments Language Translator"), it was used on various TI computers during the 70s and 80s.

This program has some similarities to `awk`, but they are generally due more to similarity of purpose than to any deliberate copying. I did copy the `$0` notation and adopt the term *action*.

This program was designed and coded by myself, David N. Gray, except for the regular expression processor, which utilizes public domain code written by Ozan S. Yigit and updated by Craig Durland and Harlan Sexton. David A. Mundie[8] supplied modifications to enable use on the Macintosh, and offered helpful comments and encouragement.

Thanks to Remo Dentato for the integration with Lua and for setting up a SourceForge project[9] and to Cecil Bayona for providing a mirror distribution site[10].

[8] `mundie@anthus.com`

[9] `http://sourceforge.net/projects/gema/`

[10] `http://www.cbayona.com/pub/Macro%20Languages/Gema/`