Printed September 12, 2024

APREPRO: An Algebraic Preprocessor for Parameterizing Finite Element Analyses

Gregory D. Sjaardema
Sandia National Laboratories
Albuquerque, NM 87185-0380

Abstract

Aprepro is an algebraic preprocessor that reads a file containing both general text and algebraic, string, or conditional expressions. It interprets the expressions and outputs them to the output file along with the general text. The syntax used in Aprepro is such that all expressions between the delimiters **{** and **}** are evaluated and all other text is simply echoed to the output file. Aprepro contains several mathematical functions, string functions, and flow control constructs. In addition, functions are included that implement a units conversion system. Aprepro was written primarily to simplify the preparation of parameterized input files for finite element analyses at Sandia National Laboratories; however, it can process any text file that does not use the characters **{ }**.

Contents

List of Tables

1 Introduction

Aprepro is an algebraic preprocessor that reads a file containing both general text and algebraic expressions. It echoes the general text to the output file, along with the results of the algebraic expressions. The syntax used in Aprepro is such that all expressions between the delimiters **{** and **}** are evaluated and all other text is simply echoed to the output file. For example, if the following lines are input to Aprepro:

```
$ Rad  = {Rad  =  12.0}
Point 1 {x1 = Rad * sind(30.)} {y1 = Rad * cosd(30.)}
Point 2 {x1 + 10.0}            {y1}
```

The output would look like:

```
$ Rad = 12
Point  1 6   10.39230485
Point  2 16  10.39230485
```

In this example, the algebraic expressions are specified by surrounding them with **{** and **}**, and the functions `sind()` and `cosd()` calculate the sine and cosine of an angle given in degrees.

Aprepro has been used extensively for several years to prepare parameterized files for finite element analyses using the Sandia National Laboratories SEACAS system [1]. The units conversion capability has greatly increased the usability of Aprepro. Aprepro can also be used for non-finite element applications such as a powerful calculator and a general text processor for any file that does not use the delimiters **{** and **}**.

The remainder of this document is organized as follows:

- Chapter 2 documents the command line options for Aprepro and the text input, editing, and recall capabilities.
- Chapter 3 documents the syntax recognized by *Aprepro*,
- Chapters 4, 5, and 6 describe the operators, predefined variables, and functions,
- Chapter 7 describes the units conversion system,
- Chapter 8 describes the error messages output from Aprepro, and
- Chapter 9 presents some examples of Aprepro usage.

2 Execution

2.1 Aprepro Execution and Program Options

Aprepro is executed with the command:

```
aprepro [--parameters] [-dsviehMWCq] [-I path] [-c char] [var=val] filein fileout
```

The effects of the parameters are:

–debug (-d)	Dump all variables, debug loops/if/endif
–dumpvars (-D)	Dump all variables at end of run
–dumpvars_json (-J)	Dump all variables at end of run in json format
–version (-v)	Print version number to stderr
–comment char (-c char)	Change comment character to 'char'
–immutable (-X)	All variables are immutable–cannot be modified
–errors_fatal (-f)	Exit program with nonzero status if errors are encountered
–errors_and_warnings_fatal (-F)	Exit program with nonzero status if warnings are encountered
–require_defined (-R)	Treat undefined variable warnings as fatal
–interactive (-i)	Interactive use, no buffering of output.
–include path (-I path)	Include file or include path. Any variables defined in the include file will be immutable.
–one_based_index (-1)	Array indexing is one-based (default = zero-based)
–exit_on (-e)	If this is enabled, Aprepro will exit when any of the strings EXIT, Exit, exit, QUIT, Quit, or quit are entered. Otherwise, Aprepro will exit at end of file.
–message (-M)	Print INFO messages. (See Chapter 8 for a list of INFO messages.)
–info=file	Output INFO messages (e.g. DUMP() output) to file.
–nowarning (-W)	Do not print warning messages. (See Chapter 8 for a list of warning messages.)
–copyright (-c)	Print copyright message
–quiet (-q)	Do not anything extra to stdout
–help (-h)	Print this list
var=val	Assign value *val* to variable *var*. This lets you dynamically set the value of a variable and change it between runs without editing the input file. Multiple *var=val* pairs can be specified on the command line. A variable that is defined on the command line will be an immutable variable whose value cannot be changed[1]. If *var* is a string variable, then *val* needs to be surrounded by escaped double quotes. For example `name=\"My\Name\"` will define the string variable *name*.[2]
input_file	specifies the file that contains the Aprepro input. If this parameter is omitted, Aprepro will run interactively.
output_file	specifies the file Aprepro will write the processed data to. If this parameter is omitted, Aprepro will write the data to the terminal. (stdout)

The - followed by a single letter shown in the parameter descriptions above are optional short-options that can be specified instead of the long options. For example, the following two lines are equivalent:

```
aprepro --debug --nowarning --statistics --comment #
aprepro -dWsc#
```

[1] Unless the variable name begins with an underscore.
[2] Note that any spaces in the string variables value must be escaped also.

Note that the short options can be concatenated.

2.2 Interactive Input

If no input file is specified when Aprepro is executed, then all input will be read from standard input; or in other words, typed in by the user. In this mode, there are a few command-line editing and recall capabilities provided.

The command-line editing provides Emacs style key bindings and history functionality. The key bindings are shown in the following table. The syntax `^X` indicates that the user should press and hold the "control" key and then press the `X` key. The syntax `M-X` indicates pressing the "meta" key followed by the `X` key. The meta key is sometimes escape, or sometimes "alt", or some other key depending on the users keymap.

Table 2.2: Key Bindings used in the interactive input to Aprepro

Key	Function
^A/^E	Move cursor to beginning/end of the line.
^F/^B	Move cursor forward/backward one character.
^D	Delete the character under the cursor.
^H	Delete the character to the left of the cursor.
^K	Kill from the cursor to the end of line.
^L	Redraw current line.
^O	Toggle overwrite/insert mode. Initially in insert mode. Text added in overwrite mode (including yanks) overwrite existing text, while insert mode does not overwrite.
^P/^N	Move to previous/next item on history list.
^R/^S	Perform incremental reverse/forward search for string on the history list. Typing normal characters adds to the current search string and searches for a match. Typing ^R/^S marks the start of a new search, and moves on to the next match. Typing ^H deletes the last character from the search string, and searches from the starting location of the last search. Therefore, repeated ^H's appear to unwind to the match nearest the point at which the last ^R or ^S was typed. If ^H is repeated until the search string is empty the search location begins from the start of the history list. Typing ESC or any other editing character accepts the current match and loads it into the buffer, terminating the search.
^T	Toggle the characters under and to the left of the cursor.
^U	Kill from beginning to the end of the line.
^Y	Yank previously killed text back at current location. Note that this will overwrite or insert, depending on the current mode.
M-F/M-B	Move cursor forward/backward one word.
^SPC	Set mark.
^W	Kill from mark to point.
^X	Exchange mark and point.
RETURN	returns current buffer to the program.

3 Syntax

Aprepro is in one of two states while it is processing an input file, either echoing or parsing. In the *echoing* state, Aprepro echoes every character that it reads to the output file. If it reads the character {, it enters the *parsing* state. In the parsing state, Aprepro reads characters from the input file and identifies the characters as tokens which can be *function names*, *variables*, *numbers*, *operators*, or *delimiters*. When Aprepro encounters the character }, it tries to interpret the tokens as an algebraic, string, or conditional expression; if it is successful, it prints the value to the output file; if it cannot evaluate the expression, it prints the message:

```
Aprepro:  ERROR: parse error {filename}, line {line#}
```

to the terminal[1] prints the value 0 to the output file.

The rules that Aprepro uses when identifying functions, variables, numbers, operators, delimiters, and expressions are described below:

Functions Function names are sequences of letters and digits and underscores (`_`) that begin with a letter. The function's arguments are enclosed in parentheses.

For example, in the line `atan2(a,1.0)`, `atan2` is the function name, and `a` and `1.0` are the arguments. See Chapter 6 for a list of the available functions and their arguments.

Variables A variable is a name that references a numeric or string value. A variable is defined by giving it a name and assigning it a value. For example, the expression `a = 1.0` defines the variable `a` with the numeric value `1.0`; the expression `b= "A string"` defines the variable `b` with the value `A string`. Variable names are sequences of letters, digits, colons (:), and underscores (_) that begin with either a letter or an underscore. Variable names cannot match any function name and they are case-sensitive, that is, `abc_de` and `AbC_dE` are two distinct variable names. A few variables are predefined, these are listed in Chapter 5.

Any variable that is not defined is equal to 0. A warning message is output to the terminal if an undefined variable is used, or if a previously defined variable is redefined. If the variable name begins with an underscore, no warning is output when the variable is redefined.[2].

Immutable Variables An immutable variable is a variable whose value cannot be changed. An immutable variable name is follows the same rules as a regular variable except that the name cannot begin with an underscore. Immutable variables are created inside an `IMMUTABLE(ON)` block (See Section **??**) or when Aprepro is executed with the `--immutable` or `-X` command line options (See Chapter 2). A value defined on the command line is immutable ()(See Chapter 2). If the value of an immutable variable is attempted to be modified, an error message of the form: [Aprepro: (IMMUTABLE) Variable 'variable' is immutable and cannot be modified (file, line line#)] will be output to the standard error stream and the expression containing the assignment to the immutable variable will return nothing.

Numbers Numbers can be integers like `1234`, decimal numbers like `1.234`, or in scientific notation like `1.234E-26`. All numbers are stored internally as floating point numbers.

Strings Strings are sequences of numbers, characters, and symbols that are delimited by either single quotes (`'this is a string'`) or double quotes (`"this is another string"`). Strings that are delimited by one type of quote can include the other type of quote. For example,

[1] Error messages are printed to standard error. On UNIX systems they can be redirected to a file using your shells redirection syntax. See the man page for your shell for more information.

[2] Warnings can be turned off with the `-W` or `--warning` option.

`'This is a valid "string"'`. Strings delimited by single quotes can span multiple lines; strings delimited by double quotes must terminate on a single line or a parsing error message will be issued.

Operators Operators are any of the symbols defined in Chapter 4. Examples are + (addition), - (subtraction), * (multiplication), / (division), = (assignment), and ^ (exponentiation)

Delimiters The delimiters recognized by APREPRO are: the comma (`,`) which separates arguments in function lists, the left curly brace ({) which begins an expression, the right curly brace (}) which ends an expression, the left parenthesis `(` which begins a function argument list, the right parenthesis `)` which ends a function argument list, the single quote (`'`) which delimits a multiline string, and the double quote (`"`) which delimits a single-line string. If a left or right curly brace is needed in the processes output without being interpreted by APREPRO, precede the curly brace with a backslash. For example, `\{ \}`.

Expressions An expression consists of any combination of numeric and string constants, variables, operators, and functions. Four types of expressions are recognized in APREPRO: algebraic, string, relational, and conditional.

Algebraic Expressions Almost any valid FORTRAN or C algebraic expression can be recognized and evaluated by APREPRO. An expression of the form `a=b+10/37.5` will evaluate the expression on the right-hand-side of the equals sign, print the value to the output file, and assign the value to the variable `a`. An expression of the form `b+10/37.5` will evaluate the expression and print the value to the output file. If you want to assign a value to a variable without printing the result, the expression must be inside an `ECHO(ON|OFF)` block (see 27). Variables can also be set on the command line prior to reading any input files using the `var=val` syntax. An example of this usage is given in Section 9.3. Only a single expression is allowed within the { } delimiters. For example, {`x=sqrt(y^2 + sin(z))`}, {`x=y=z`}, and{`x=y`} {`a=z`} are valid expressions, but {`x=y a=z`} is invalid because it contains two expressions within a single set of delimiters.

String Expressions APREPRO has limited string support. The only supported operations are (1) assigning a variable equal to a string (`a = "This is a string"`), (2) functions that return a string (See Table 6.2), and (3) concatenating two strings into another string (`a = "Hello" // " " // "World"`).

Relational Expressions: Relational expressions are expressions that return the result of comparing two expressions. A relational expression is either true (returns 1) or false (returns 0). A relational expression is simply two expressions of any kind separated by a relational operator (See Section 4.3).

Conditional Expressions APREPRO recognizes a conditional expression of the form:

```
relational_expression ? true_exp :  false_exp
```

where `relational_expression` can be any valid relational expression, and `true_exp` and `false_exp` are two algebraic expressions or string expressions. If the relational expression is true, then the result of `true_exp` is returned, otherwise the result of `false_exp` is returned. For example, if the following command were entered:

```
a = (sind(20.0) > cosd(20.0) ? 1 : -1)
```

then, `a` would be assigned the value `-1` since the relational expression to the left of the question mark is false. Both `true_exp` and `false_exp` are always evaluated prior to evaluating the relational expression. Therefore, you should not write an equation such as

```
sind(20.0*a) > cosd(20.0*a) ? a=sind(20.0) : a=cosd(20.0)
```

since the value of `a` can change during the evaluation of the expression. Instead, this equation should be written as:

```
a = (sind(20.0*a) > cosd(20.0*a) ? sind(20.0) : cosd(20.0))
```

4 Operators

The operators recognized by APREPRO are listed below. The letters **a** and **b** can represent variables, numbers, functions, or expressions unless otherwise noted. The tables below also list the precedence and associativity of the operators. *Precedence* defines the order in which operations should be performed. For example, in the expression:

```
{3 * 4 + 6 / 2}
```

the multiplications and divisions are performed first, followed by the addition because multiplication and division have higher precedence (10) than addition (9). The precedence is listed from 1 to 14 with 1 being the lowest precedence and 14 being the highest.

Associativity defines which side of the expressions should be simplified first. For example the expression: `3 + 4 + 5` would be evaluated as `(3 + 4) + 5` since addition is left associative; in the expression **a** = **b** / **c**, the `b/c` would be evaluated first followed by the assignment of that result to `a` since equality is right associative

4.1 Arithmetic Operators

Arithmetic operators combine two or more algebraic expressions into a single algebraic expression. These have obvious meanings except for the pre- and post- increment and decrement operators. The pre-increment and pre-decrement operators first increment or decrement the value of the variable and then return the value. For example, if `a = 1`, then `b=++a` will set both `b` and `a` equal to `2`. The post-increment and post-decrement operators first return the value of the variable and then increment or decrement the variable. For example, if `a = 1`, then `b=a++` will set `b` equal to `1` and `a` equal to `2`. The modulus operator `%` calculates the integer remainder. That is both expressions are truncated an integer value and then the remainder calculated. See the `fmod` function in Table 6.1 for the calculation of the floating point remainder. The tilde character `~` is used as a synonym for multiplication to improve the aesthetics of the unit conversion system (see Chapter 7). It is more natural for some users to type `12~metre` than `12*metre`.

Table 4.1: Arithmetic Operators

Syntax	Description	Precedence	Associativity
`a+b`	Addition	9	left
`a-b`	Subtraction	9	left
`a*b`, `a~b`	Multiplication	10	left
`a/b`	Division	10	left
`a^b`, `a**b`	Exponentiation	12	right
`a%b`	Modulus, (remainder)	10	left
`++a`, `a++`	Pre- and Post-increment a	13	left
`--a`, `a--`	Pre- and Post-decrement a	13	left

4.2 Assignment Operators

Assignment operators combine a variable and an algebraic expression into a single algebraic expression, and also set the variable equal to the algebraic expression. Only variables can be specified on the left-hand-side of the equal sign.

Table 4.2: Assignment Operators

Syntax	Description	Precedence	Associativity
`a=b`	The value of a is set equal to b	1	right
`a+=b`	The value of a is set equal to $a + b$	2	right
`a-=b`	The value of a is set equal to $a - b$	2	right
`a*=b`	The value of a is set equal to $a * b$	3	right
`a/=b`	The value of a is set equal to a/b	3	right
`a^=b`, `a**=b`	The value of a is set equal to a^b	4	right

4.3 Relational Operators

Relational operators combine two algebraic expressions into a single relational expression. Relational expressions and operators can only be used before the question mark (`?`) in a conditional expression.

Table 4.3: Relational Operators

Syntax	Description	Precedence	Associativity
`a < b`	true if a is less than b	8	left
`a > b`	true if a is greater than b	8	left
`a <= b`	true if a is less than or equal to b	8	left
`a >= b`	true if a is greater than or equal to b	8	left
`a == b`	true if a is equal to b	8	left
`a != b`	true if a is not equal to b	8	left

4.4 Boolean Operators

Boolean operators combine one or more relational expressions into a single relational expression. If `la` and `lb` are two relational expressions, then:

Table 4.4: Boolean Operators

Syntax	Description	Precedence	Associativity
`la || lb`	true if either la or lb are true.	6	left
`la && lb`	true if both la and lb are true.	7	left
`!la`	true if la is false.	11	left

The evaluation of the expression is not short-circuited if the truth value can be determined early; both sides of the expression are evaluated and then the truth of the expression is returned.

4.5 String Operators

The only supported string operator at this time is string concatenation which is denoted by `//`. For example,

```
{a = "Hello"} {b = "World"}
{c = a // " " //  b}
```

sets `c` equal to `"Hello World"`. Concatenation has precedence 14 and left associativity.

5 Predefined Variables

A few commonly used variables are predefined in APREPRO[1]. These are listed below. The default output format _FORMAT is specified as a C language format string, see your C language documentation for more information. The default output format (_FORMAT) and comment (_C_) variables are defined with a leading underscore in their name so they can be redefined without generating an error message.

Table 5.1: Predefined Variables

Name	Value	Description
PI	3.14159265358979323846	π
PI_2	1.57079632679489661923	$\pi/2$
TAU	6.28318530717958623200	2π
SQRT2	1.41421356237309504880	$\sqrt{2}$
DEG	57.2957795130823208768	$180/\pi$ degrees per radian
RAD	0.01745329251994329576	$\pi/180$ radians per degree
E	2.71828182845904523536	base of natural logarithm
GAMMA	0.57721566490153286060	γ, euler-mascheroni constant
PHI	1.61803398874989484820	golden ratio $(\sqrt{5}+1)/2$
TRUE	1	
FALSE	0	
VERSION	Varies, string value	current version of APREPRO
_FORMAT	`"%.10g"`	default output format
_C_	`"$"`	default comment character

Note that the output format is used to output both integers and floating point numbers. Therefore, it should use the `%g` format descriptor which will use either the decimal (`%d`), exponential (`%e`), or float (`%f`) format, whichever is shorter, with insignificant zeros suppressed.

If the output format is set to the empty string, the output will use as many variables as needed to fully represent the double precision value. This can be selected at startup time using the command-line option –**full_precision** or **-p**.

The table below illustrates the effect of different format specifications on the output of the variable **PI** and the value 1.0 . See the documentation of your C compiler for more information. For most cases, the default value is sufficient.

If you need to temporarily change the output format for a specific case, you can use the `format(var, format)` command which will return a string representing *var* printed using the format specified in *format*.

Table 5.2: Effect of various output format specifications

_FORMAT	PI Output	1.0 Output
%.10g	3.141592654	1
%.10e	3.1415926536e+00	1.0000000000e+00
%.10f	3.1415926536	1.0000000000
%.10d	1413754136	0000000000
""	3.141592653589793	1

[1]The units system described in Chapter 7 also predefines several variables when it is activated

The comment character should be set to the character that the program which will read the processed file uses as a comment character. The default value of `"$"` is the comment character used by the SEACAS codes at Sandia National Laboratories. The **-c** command line option (described in Chapter 2) changes the value of the comment variable to match the character specified on the command line.

6 Functions

Several mathematical and string functions are implemented in Aprepro. To cause a function to be used, you enter the name of the function followed by a list of zero or more arguments in parentheses. For example

```
{sqrt(min(a,b*3))}
```

uses the two functions `sqrt()` and `min()`. The arguments `a` and `b*3` are passed to `min()`. The result is then passed as an argument to `sqrt()`. The functions in Aprepro are listed below along with the number of arguments and a short description of their effect.

6.1 Mathematical Functions

The following mathematical functions are available in Aprepro.

Table 6.1: Mathematical Functions

Syntax	Description
abs(x)	Absolute value of x. $\|x\|$.
acos(x)	Inverse cosine of x, returns radians.
acosd(x)	Inverse cosine of x, returns degrees.
acosh(x)	Inverse hyperbolic cosine of x.
asind(x)	Inverse sine of x, returns degrees.
asin(x)	Inverse sine of x, returns radians.
asinh(x)	Inverse hyperbolic sine of x.
atan(x)	Inverse tangent of x, returns radians.
atan2(x,y)	Inverse tangent of x/y, returns radians.
atan2d(x,y)	Inverse tangent of x/y, returns degrees.
atand(x)	Inverse tangent of x, returns degrees.
atanh(x)	Inverse hyperbolic tangent of x.
cbrt(x)	Cube root of x. $\sqrt[3]{x}$.
ceil(x)	Smallest integer not less than x.
cos(x)	Cosine of x, with x in radians
cosd(x)	Cosine of x, with x in degrees
cosh(x)	Hyperbolic cosine of x.
CtoF(x)	Convert from degrees Celsius to degrees Fahrenheit.
d2r(x)	Degrees to radians.
dim(x,y)	$x - \min(x, y)$
dist(x1,y1, x2,y2)	$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$
erf(x)	Error Function $\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt$
erfc(x)	Complementary Error Function $1 - \mathrm{erf}(x)$
exp(x)	Exponential e^x
expm1(x)	Exponential. Accurate version of $e^x - 1.0$ for small x.
find_word(word,svar,del)	Find 1-based index of *word* in *svar*. Words are separated by one or more of the characters in the string variable del. Returns 0 if *word* is not found.
floor(x)	Largest integer not greater than x.
fmod(x,y)	Floating-point remainder of x/y.

Table 6.1: Mathematical Functions

Syntax	Description
FtoC(x)	Convert from degrees Fahrenheit to degrees Celsius.
hypot(x,y)	$\sqrt{x^2+y^2}$.
int(x), [x]	Integer part of x truncated toward 0.
julday(mm, dd, yy)	Julian day corresponding to mm/dd/yy.
juldayhms(mm, dd, yy, hh, mm, ss)	Julian day corresponding to mm/dd/yy at hh:mm:ss
lgamma(x)	$\log(\Gamma(x))$.
ln(x)	Natural (base e) logarithm of x.
log(x)	Natural (base e) logarithm of x.
log10(x)	Base 10 logarithm of x.
log1p(x)	$log(1+x)$ Accurate even for very small values of x
max(x,y)	Maximum of x and y.
min(x,y)	Minimum of x and y.
nint(x)	Rounds x to nearest integer. < 0.5 down; $>= 0.5$ up.
polarX(r,a)	$r * \cos(a)$, a is in degrees
polarY(r,a)	$r * \sin(a)$, a is in degrees
pow(x,y)	Power x^y.
r2d(x)	Radians to degrees.
rand(xl,xh)	Random value between xl and xh; uniformly distributed.
rand_lognormal(m,s)	Random value with lognormal distribution with mean m and std-dev s.
rand_normal(m,s)	Random value normally distributed with mean m and stddev s.
rand_weibull(a, b)	Random value with weibull distribution with $\alpha = a$ and $\beta = b$.
seconds()	Returns the number of seconds since the epoch. The value is useful as the *seed* value in the function `srand`.
sign(x,y)	$x * \text{sgn}(y)$
sin(x)	Sine of x, with x in radians.
sind(x)	Sine of x, with x in degrees.
sinh(x)	Hyperbolic sine of x
sqrt(x)	Square root of x. $\sqrt{x}$
srand(seed)	Seed the random number generator with the given integer value. At the beginning of APREPRO execution, `srand()` is called with the current time as the seed.
strtod(svar)	Returns a double-precision floating-point number equal to the value represented by the character string pointed to by *svar*.
tan(x)	Tangent of x, with x in radians.
tand(x)	Tangent of x, with x in radians.
tanh(x)	Hyperbolic tangent of x.
tgamma(x)	Gamma Function $\Gamma(x) = \int_0^\infty t^{x-1}e^{-t}dt$.
Vangle(x1,y1,x2,y2)	Angle (radians) between vector $x_1\hat{i} + y_1\hat{j}$ and $x_2\hat{i} + y_2\hat{j}$.
Vangled(x1,y1,x2,y2)	Angle (degrees) between vector $x_1\hat{i} + y_1\hat{j}$ and $x_2\hat{i} + y_2\hat{j}$.
word_count(svar,del)	Number of words in *svar*. Words are separated by one or more of the characters in the string variable del.

Table 6.2: String Functions

Syntax	Description
DUMP()	Output a list of all defined variables and their value.

Table 6.2: String Functions

Syntax	Description
DUMP_JSON()	Output a list of all defined variables and their value in JSON format.
DUMP_FUNC()	Output a list of all double and string functions recognized by APREPRO.
DUMP_PREVAR()	Output a list of all predefined variables and their value.
IO(x)	Convert x to an integer and then to a string. Can be used to output integer values if your output format (`_FORMAT`) is set to something that doesn't output integers correctly.
Units(svar)	See Chapter 7. *svar* is one of the defined units systems: 'si', 'cgs', 'cgs-ev', 'shock', 'swap', 'ft-lbf-s', 'ft-lbm-s', 'in-lbf-s'
error(svar)	Outputs the string *svar* to stderr and then terminates the code with an error exit status.
execute(svar)	*svar* is parsed and executed as if it were a line read from the input file.
exodus_info(filename, prefix)	Open the EXODUS file and return a string which is the concatenation of all EXODUS info lines that begin with "prefix". The prefix is stripped from the line.
exodus_info(filename, begin, end)	Open the EXODUS file and return a string which is the concatenation of all EXODUS info lines following the line that matches "begin" up to the line that matches "end".
exodus_meta(filename)	Open the EXODUS file and create several variables based on the metadata in the EXODUS file.
extract(s, b, e)	Return substring *[b,e)*. *b* is included; *e* is not. If *b* not found, return empty; If *e* not found, return rest of string. If *b* empty, start at beginning; if *e* empty, return rest of string.
file_to_string(fn)	Opens the file specified by *fn* and returns the contents as a multi-line string.
format(var, format)	Returns a string representing the double value *var* printed using the format specified in *format*.
get_date()	Returns a string representing the current date in the form YYYY/MM/DD.
get_iso_date()	Returns a string representing the current date in the form YYYYMMDD.
get_time()	Returns a string representing the current time in the form HH:MM:SS.
get_word(n,svar,del	Returns a string containing the *n*th word of *svar*. The words are separated by one or more of the characters in the string variable *del*
getenv(svar)	Returns a string containing the value of the environment variable *svar*. If the environment variable is not defined, an empty string is returned.
help()	Tell how to get help on variables, functions, ...
include_path(path)	Specify an optional path to be prepended to a filename when opening a file. Can also be specified via the `-I` command line option when executing aprepro.
import(svar)	include contents of the file specified by the value of *svar*. See Section 6.2.2 for details.
include(file)	include contents of the file. See Section 6.2.2 for details.

Table 6.2: String Functions

Syntax	Description
cinclude(file)	conditionally include contents of the file. See Section 6.2.2 for details.
import(svar)	include contents of the file pointed to by *svar*. See Section 6.2.2 for details.
output(filename)	Creates the file specified by *filename* and sends all subsequent output from aprepro to that file. Calling `output(ëstdout)` will close the current output file and return output to the terminal (standard output).
output_append(fn)	If file with name *fn* exists, append output to it; otherwise create the file and send all subsequent output from aprepro to that file.
rescan(svar)	The difference between `execute(sv1)` and `rescan(sv2)` is that *sv1* must be a valid expression, but *sv2* can contain zero or more expressions.
to_lower(svar)	Translates all uppercase characters in *svar* to lowercase. It modifies *svar* and returns the resulting string.
tolower(svar)	Translates all uppercase characters in *svar* to lowercase. It modifies *svar* and returns the resulting string.
to_string(x)	Returns a string representation of the numerical variable x. The variable x is unchanged.
tostring(x)	Returns a string representation of the numerical variable x. The variable x is unchanged.
to_upper(svar)	Translates all lowercase character in *svar* to uppercase. It modifies *svar* and returns the resulting string.
toupper(svar)	Translates all lowercase character in *svar* to uppercase. It modifies *svar* and returns the resulting string.
version()	Return the version string. (See also the *_VERSION* variable).

Table 6.3: Array Functions

Syntax	Description
csv_array(filename, [skip])	Create a 2D array from the data in a CSV (Comma-Separated-Value) file optionally skipping rows. If *skip* is an integer, then skip that many rows; if *skip* is a character, then skip lines beginning with that character.
array_from_string(string, delim)	Create a 1D array from the data in a delimited *string*. The array double values are separated by one or more of the characters in the string variable *delim*.
identity(size)	Create a 2D identity array with *size* rows and columns. The elements along the diagonal are equal to 1.0
linear_array(init, final, count)	Create a 1D array of *count* rows. Values are linearly spaced from *init* to *final*.
make_array(rows, cols, init=0)	Create a 2D array of size *rows* by *cols* initialized to *init*. The array is initialized to 0 if *init* is not specified.
transpose(array)	Return the transpose of input array

Table 6.4: Functions with Array variables as parameters

Syntax	Description
cols(array)	Returns the number of columns in the array *array*.
rows(array)	Returns the number of rows in the array *array*.
print_array(array)	Prints the data in the array *array*.

The following example shows the use of some of the string functions. The input:

```
{t1 = "ATAN2"}
{t2 = "(0, -1)"}
{t3 = tolower(t1//t2)}
{execute(t3)}
```

produces the output:

```
ATAN2
(0, -1)
atan2(0, -1)    The variable t3 is equal to the string atan2(0,-1)
3.141592654     The result is the same as executing {atan2(0, -1)}
```

This is admittedly a very contrived example; however, it does illustrate the workings of several of the functions. In the example, an expression is constructed by concatenating two strings together and converting the resulting string to lowercase. This string is then executed and simply prints the result of evaluating the expression.

The following example uses the **rescan** function to illustrate a basic macro capability in APREPRO. The example calculates the coordinates of eleven points (Point1 ... Point11) equally spaced about the circumference of a 180 degree arc of radius 10.

```
{ECHO(OFF)} {num = 0}
{rad = 10}
{nintv = 10}
{nloop = nintv + 1}
{line = 'Define {"Point"//tostring(++num)}, {polarX(rad, (num-1) *
  180/nintv)} {polarY(rad, (num-1)*180/nintv)}'}
{ECHO(ON)}
{loop(nloop)}
{rescan(line)}
{endloop}
```

Output:

```
Define  Point1,  10    0
Define  Point2,  9.510565163    3.090169944
Define  Point3,  8.090169944    5.877852523
Define  Point4,  5.877852523    8.090169944
Define  Point5,  3.090169944    9.510565163
Define  Point6,  6.123233765e-16    10
Define  Point7,  -3.090169944    9.510565163
Define  Point8,  -5.877852523    8.090169944
Define  Point9,  -8.090169944    5.877852523
Define  Point10,  -9.510565163    3.090169944
Define  Point11,  -10    1.224646753e-15
```

Note the use of the `ECHO(OFF|ON)` block to suppress output during the initialization phase, and the loop construct to automatically repeat the rescan line. The variable `num` is converted to a string after it is incremented and then concatenated to build the name of the point. In the definition of the variable `line`, single quotes are first used since this is a multi-line string; double quotes are then used to embed another string within the first string. To modify this example to calculate the coordinates of 101 points rather than eleven, the only change necessary would be to set {`nintv=100`}.

6.2 Additional Functions

6.2.1 [*var*] or [*expression*]

Surrounding a variable or expression by square brackets will return the integer value of that variable or expression truncated toward zero. For example `[sqrt(2)]` will return the value `1`.

6.2.2 File Inclusion

APREPRO can read input from multiple files using the `include()`, `cinclude()`, and `import()` functions. If a line of the form:

{`include("`*`filename`*`")`}
{`include(string_variable)`}
{`import(string_expression)`}

is read, APREPRO will open and begin reading from the file ***filename***. A string variable can be used as the argument instead of a literal string value. In the `import()` command, the argument can be an expression that evaluates to a string, a string variable, or a literal string. For example:

```
{base = "filename"}
{ext = "apr"}
{import(base // "." // ext)}
```

Will result in the contents of the file ***filename.apr*** being included.

When the end of the file is reached, it will be closed and APREPRO will continue reading from the previous file. The difference between `include()`, `import()`, and `cinclude()` is that if ***filename*** does not exist, `include()` and `import()` will terminate APREPRO with a fatal error, but `cinclude()` will just write a warning message and continue with the current file. The `cinclude()` function can be thought of as a *conditional include*, that is, include the file if it exists. Multiple include files are allowed and an included file can also include additional files. This option can be used to set variables globally in several files. For example, if two or more input files share common points or dimensions, those dimensions can be set in one file that is included in the other files.

If `ECHO(OFF)` is in effect during in an included file, `ECHO(ON)` will automatically be executed at the end of the included file.

6.2.3 Conditionals

Portions of an input file can be conditionally processed through the use of the `if(expression)`, `elseif(expression)`, `else`, and `endif` construct.[1] The syntax is:

[1] The `Ifdef(expression)` and `Ifndef(expression)` construct is deprecated. Please use `if(expression)` and `if(!expression)` instead.

```
{if(expression)}
... Lines processed if 'expression' is true or non-zero.
{elseif(expression2)}
... Lines processed if 'expression' is false and 'expression2' is true.
{else}
... Lines processed if both 'expression' and 'expression2' are false.
{endif}
```

The `elseif()` and `else` are optional. Note that if *expression* is a simple *variable*, then its value will be zero or false if it is undefined; a zero value evaluates to false and a non-zero value is true. The `if` construct can be nested multiple levels. A warning message will be printed if improper nesting is detected. The `if(expression)`, `elseif(expression)`, `else`, and `endif` are the only text parsed on a line. Text that follows these on the same line is ignored. For example:

```
{if(a > 10 && b < 10)} This will be ignored no matter what
... Lines processed if a > 10 and b < 10.
{endif}
```

6.2.4 Switch Statements

The `switch` statement is a control construct which allows the value of a variable or expression to change the control flow via a multi-way branch. The construct is begun with a `switch(expression)` statement followed by one or more `case(expression)` statements and an optional `default` statement. The construct is ended with an `endswitch` statement. The expression in the `switch(expression)` statement is evaluated and compared to each `case(expression)` statement in order. If the values of the two expressions are equal, then the code following that `case(expression)` is evaluated up to the next `case()` or `default` statement. If the expressions in more than one `case()` match the initial `switch()` expression, only the first one will be activated. If none of the `case()` expressions match the `switch()` expression, then the code following the `default` command will be evaluated. An example of the syntax is:

```
{a = 10*PI}
{switch(10*PI + sin(0))}
... This is ignored since it is after the switch, but before any case() statements
{case(1)}
... This is not executed since 1 is not equal to 10*PI+sin(0)
{case(a)}
... This is executed since a matches the value of 10*PI+sin(0)
{case(10*PI+sin(0))}
... This is not executed since a previous case was executed.
{default}
... This is not executed since a previous case was executed.
{endswitch}
... This is executed since the switch construct
 is finished.
```

Switch constructs cannot be nested, but a `switch()` can be used inside an `if()` construct and an `if()` can be used inside a `case()` construct. The `switch(expression)`, `case(expression)`, `default`, and `endswitch` are the only text parsed on a line. Text that follows these on the same line is ignored.

6.2.5 Loops

Repeated processing of a group of lines can be controlled with the `loop(control)`, and `endloop` commands. The syntax is:

```
{loop(variable, index_variable, initial_value, increment)}
... Process these lines variable times.
{endloop}
```

The number of iterations of the loop is specified by *variable* which can be an explicit integer value, or an existing variable. If it is a variable, then the truncated integer value of the variable will be used. For example, if the variable contains the value 3.14, then the loop will be iterated 3 times.

The optional *index_variable* will be automatically initialized to the optional *initial_value* (0 if not specified) at the beginning of the loop and incremented by the optional *increment* (1 if not specified) value each time through the loop. Loops can be nested.

If the *index_variable* is not specified in the `loop` command, then the variable *__loop_#* will be used where "#" represents the one-based nesting level of the current loop.

A numerical variable or constant must be specified as the loop control specifier. You cannot use an algebraic expression such as

```
{loop(3+5)}  ... INVALID
```

The `endloop` command must be on a line by itself with no other text except optional whitespace (spaces or tabs).

These are all valid loop invocations:

```
{loop(variable, index_variable, initial_value, increment)}

{loop(variable, index_variable, initial_value)}
... same as {loop(variable, index_variable, initial_value, 1)}

{loop(variable, index_variable)}
... same as {loop(variable, index_variable, 0, 1)}

{loop(variable)}
... same as {loop(variable, __loop_1, 0, 1}
```

A couple examples of using loops are shown here:

```
{loop(10, _i, 1, 2)}
... These lines will be executed 10 times.
... The variable _i will have the values 1, 3, 5, ..., 19.
{endloop}

{outer=10} {inner=4}
{loop(outer, _i)}
{loop(inner)}
... These lines will be executed $10 * 4$ times.
... The variable _i will have the values 0 through 9.
... The variable __loop_2 will have the values 0, 1, 2, 3 and incremented in each iteration of the inner lo
{endloop}
{endloop}
```

6.2.6 ECHO

The printing of lines to the output file can be controlled through the use of the `ECHO(OFF)` and `ECHO(ON)` commands. The syntax is:

```
{ECHO(OFF)}
... These lines will be processed, but not printed to output
{ECHO(ON)}
... These lines will be both processed and printed to output.
```

`ECHO` will automatically be turned on at the end of an included file. The commands `ECHO` and `NOECHO` are synonyms for `ECHO(ON)` and `ECHO(OFF)`.

6.2.7 Suppress individual expression output

In addition to the `ECHO` option shown in the previous section, it is possible to suppress the echoing of the results of an expression by surrounding the expression with double braces instead of the normal single braces. For example:

```
{{_i = 0}}
```

Will disable the echoing of that expression (although the newline will still be echoed).

6.2.8 VERBATIM

The printing of all lines to the output file without processing can be controlled through the use of the `VERBATIM(ON)` and `VERBATIM(OFF)` commands. The syntax is:

```
{VERBATIM(ON)}
... These lines will be printed to output, but not processed
{VERBATIM(OFF)}
... These lines will be printed to output and processed
```

NOTE: there is a major difference between the `ECHO/NOECHO` commands, the `Ifdef/Endif` commands, and the `VERBATIM(ON|OFF)` commands:

- `ECHO(ON|OFF)` Lines processed, but not printed if `ECHO(OFF)`
- `Ifdef/Endif` Lines not processed or printed if in `Ifndef` block
- `VERBATIM(ON|OFF)` Lines not processed, but are printed.

6.2.9 IMMUTABLE

Variables can either be created as mutable or immutable. By default, all variables created during a run of aprepro are mutable unless the `--immutable` or `-X` command line option is used to execute Aprepro. An `IMMUTABLE` block can also be used to change Aprepro such that all variables are created as immutable. The syntax is:

```
{IMMUTABLE(ON)}
... All variables created will be immutable
```

```
{IMMUTABLE(OFF)}
... The mutable/immutable state changes back to the default which
is typically mutable unless APREPRO executed with the
--immutable or -X options.
```

Note that any variables created as immutable are still immutable following the `IMMUTABLE(OFF)` command.

6.2.10 Output File Specification

The `output` function can be used to change the file to which APREPRO is outputting the processed data. The syntax is: {`output("`***filename***`")`}, where ***filename*** is the name of the new output file. A string variable can be used as the function argument. The previous output file is closed. An error message is written and the code terminates if the file cannot be opened. If `output("`***stdout***`")` is specified, then the current output file is closed and output is again written to the standard output which is where output is written by default.

6.2.11 EXODUS Metadata Extraction

APREPRO can parse the metadata from a binary EXODUS [2] file and create several variables which can then be used for calculations or decisions. The function syntax is `exodus_meta(filename)`. The argument to the function is a string containing the filename of the Exodus file. If the file does not exist in the current directory, APREPRO will prepend the path specified by the `--include` or `-I` command line option.

The following scalar variables will be defined:

Table 6.5: EXODUS Scalar Variables

Variable	Description
ex_dimension	Spatial dimension
ex_node_count	Number of nodes
ex_element_count	Number of elements
ex_block_count	Number of element blocks
ex_sideset_count	Number of sidesets
ex_nodeset_count	Number of nodesets
ex_timestep_count	Number of timesteps
ex_version	Version of the EXODUS database

The following string variables will be defined if the model contains one or more of the specific entity type. The strings will be a comma-separated concatenation of the names of the entity. The `get_word` function can be used to extract a specific sub-string.

Table 6.6: EXODUS String Variables

Variable	Description
ex_title	The title of the database
ex_block_names	Element Block names. Will be "block_" + block id if no names on the database.
ex_block_topology	The topology of the element blocks. Converted to all lowercase.

Table 6.6: EXODUS String Variables

Variable	Description
ex_sideset_names	Sideset names. Will be "sideset_" + sideset id if no names on the database.
ex_nodeset_names	Nodeset names. Will be "nodeset_" + nodeset id if no names on the database.

The following array variables will be defined if the model contains one or more of the specific entity type.

Table 6.7: EXODUS Array Variables

Variable	Rows	Columns	Description
ex_block_ids	ex_block_count	1	Element Block Ids.
ex_block_info	ex_block_count	4	Element Block info: id, number of elements in block, number of nodes per element, number of attributes.
ex_sideset_ids	ex_sideset_count	1	Sideset Ids.
ex_sideset_info	ex_sideset_count	3	Sideset info: id, number of faces in sideset, number of distribution factors.
ex_nodeset_ids	ex_nodeset_count	1	Nodeset Ids.
ex_nodeset_info	ex_nodeset_count	3	Nodeset info: id, number of nodes in nodeset, number of distribution factors.
ex_timestep_times	ex_timestep_count	1	Timestep times.

The following shows an example of the variables which are defined:

```
exodus_meta("filename.e")
DUMP()
Element Block IDs:
print_array(transpose(ex_block_ids))

Element Block Info: (id, num, nnpe, nattrib)
print_array(ex_block_info)

$   Variable     = Value
$  ex_block_count = 8
$  ex_nodeset_ids  (array) rows = 6, cols = 1
$  ex_block_names = "block_8,block_7,block_6,block_5,block_4,block_3,block_2,block_1"
$  ex_block_topology = "hex,hex,hex,hex,hex,hex,hex,hex"
$  ex_nodeset_info  (array) rows = 6, cols = 3
$  ex_sideset_count = 0
$  ex_dimension = 3
$  ex_element_count = 64
$  ex_nodeset_names = "nodeset_10,nodeset_100,nodeset_20,nodeset_200,nodeset_30,nodeset_300"
$  ex_nodeset_count = 6
$  ex_block_ids  (array) rows = 8, cols = 1
$  ex_block_info  (array) rows = 8, cols = 4
$  ex_timestep_times  (array) rows = 11, cols = 1
$  ex_version = 2.029999971
$  ex_timestep_count = 11
$  ex_title    = "Sierra output: dummy title"
```

```
$  ex_node_count = 125

Element Block IDs:
 8 7 6 5 4 3 2 1

Element Block Info: (id, num, nnpe, nattrib)
 8 8 8 0
 7 8 8 0
 6 8 8 0
 5 8 8 0
 4 8 8 0
 3 8 8 0
 2 8 8 0
 1 8 8 0
```

6.2.12 EXODUS Info Records Extraction

APREPRO can extract all or a portion of the "information records" from a binary EXODUS file and return the results as a string variable.

There are two forms of the function. The first function has the syntax: `exodus_info(filename, prefix)`. This will read the information records from the EXODUS database specified by the string variable *filename* and search for lines that begin with the specified *prefix*. If a line is found, the *prefix* will be stripped from the line and the remaining characters on the line will be concatenated onto the return string followed by a newline character.

The second function has the syntax: `exodus_info(filename, begin, end)`. This will read the information records from the EXODUS database specified by the string variable *filename* and search for a line that matches the string variable *begin*. It will then append all subsequent information lines onto the return string until a line that matches the string variable *end* or it reaches the end of the information records. If there is another line matching *begin*, it will resume appending lines to the return string.

The returned string can then be operated on as a normal APREPRO variable.

7 Units Conversion System

7.1 Introduction

The units conversion system as implemented in APREPRO defines several variables that are abbreviations for unit quantities. For example, if the output format for the current unit system was inches, the variable `foot` would have the value `12`. Therefore, an expression such as `8*foot` would be equal to `96` which is the number of inches in 8 feet[1].

Seven consistent units systems have been defined including four metric based systems: `si`, `cgs`, `cgs-ev`, and `shock`; and three english-based systems: `in-lbf-s`, `ft-lbf-s`, and `ft-lbm-s`. The output units for these unit systems are shown in Table 8 (metric) and Table 9 (english). A list of the defined units abbreviations is given in Table 10.

In addition to the definition of the conversion factors, several string variables are also defined which describe the output format of the current units system. For example, the string variable `dout` defines the output format for density units. For the `in-lbf-sec` units system, `dout` = `"lbf-sec^2/in^4"` which is the output format for densities in this system. The string variables can be used to document the APREPRO output. The string variable names are listed in the last column of Table 8 and Table 9.

Table 7.1: Units Systems and Corresponding Output Format–Metric

Quantity	si	cgs	cgs-ev	shock	string
Length	metre	centimetre	centimetre	centimetre	lout
Mass	kilogram	gram	gram	gram	mout
Time	second	second	second	micro-sec	tout
Temperature	kelvin	kelvin	eV	kelvin	Tout
Velocity	metre/sec	cm/sec	cm/sec	cm/usec	vout
Acceleration	metre/sec^2	cm/sec^2	cm/sec^2	cm/usec^2	aout
Force	newton	dyne	dyne	g-cm/usec^2	fout
Volume	metre^3	cm^3	cm^3	cm^3	Vout
Density	kg/m^3	g/cc	g/cc	g/cc	dout
Energy	joule	erg	erg	g-cm^2/usec^3	eout
Power	watt	erg/sec	erg/sec	g-cm^2/usec^4	Pout
Pressure	pascal	dyne/cm^2	dyne/cm^2	Mbar	pout

Table 7.2: Units Systems and Corresponding Output Format–English

Quantity	in-lbf-s	ft-lbf-s	ft-lbm-s	string
Length	inch	foot	foot	lout
Mass	lbf-sec^2/in	slug	pound-mass	mout
Time	second	second	second	tout
Temperature	rankine	rankine	rankine	Tout
Velocity	inch/sec	foot/sec	foot/sec	vout
Acceleration	inch/sec^2	foot/sec^2	foot/sec^2	aout
Force	pound-force	pound-force	poundal	fout
Volume	inch^3	foot^3	foot^3	Vout

[1] This can also be written as 8~foot since ~ has been defined to be the same as $*$ (multiplication).

Density	lbf-sec^2/in^4	slug/ft^3	lbm/ft^3	dout
Energy	inch-lbf	foot-lbf	ft-poundal	eout
Power	inch-lbf/sec	foot-lbf/sec	ft-poundal/sec	Pout
Pressure	lbf/in^2	lbf/ft^2	poundal/ft^2	pout

The units definitions are accessed through the `Units` function in Aprepro:

```
{Units("unit_system")}
```

where *unit_system* is one of the strings listed in the first row of the previous two tables.

7.2 Defined Units Variables

In the following table, the first column lists the variables that are defined in the Aprepro unit system and the second column is a short description of the unit. All units variables are defined in terms of the five SI Base Units metre (length), second (time), kilogram (mass), temperature (kelvin), and radian (angle)[2]. The bolded rows delineate the type of unit variable and the base quantities used to define it where L is length, T is time, M is mass, and t is temperature. For example density is defined in terms of M/L^3 which is mass/ length^3.

Table 7.3: Defined Units Variables

Length [L]	
m, meter, metre	`Metre (base unit)`
cm, centimeter, centimetre	Metre / 100
mm, millimeter, millimetre	Metre / 1,000
um, micrometer, micrometre	Metre / 1,000,000
km, kilometer, kilometre	Metre * 1,000
in, inch	Inch
ft, foot	Foot
yd, yard	Yard
mi, mile	Mile
mil	Mil (inch/1000)

Time [T]	
second, sec	`Second (base unit)`
usec, microsecond	Second / 1,000,000
msec, millisecond	Second / 1,000
minute	Minute
hr, hour	Hour
day	Day
yr, year	Year = 365.25 days
decade	10 Years
century	100 Years

Velocity [L/T]

[2]The radian is actually a SI Supplementary Unit since it has not been decided whether it is a Base Unit or a Derived Unit. There are three other SI Base Units, the candela, ampere, and mole, but they are not yet used in the Aprepro units system.

mph	Miles per hour
kph	Kilometres per hour
mps	Metre per second
kps	Kilometre per second
fps	Foot per second
ips	Inch per second

Acceleration $[L/T^2]$	
ga	Gravitational acceleration

Mass $[M]$	
kg	`Kilogram (base unit)`
g, gram	Gram
lbm	Pound (mass)
slug	Slug
lbfs2pin	Lbf-sec^2/in

Density $[M/L^3]$	
gpcc	Gram / cm^3
kgpm3	Kilogram / m^3
lbfs2pin4	Lbf-sec^2 / in^4
lbmpin3	Lbm / in
lbmpft3	Lbm / ft^3
slugpft3	Slug / ft^3

Force $[ML/T^2]$	
N, newton	Newton = 1 kg-m/sec^2
dyne	Dyne = newton/10,000
gf	Gram (force)
kgf	Kilogram (force)
lbf	Pound (force)
kip	Kilopound (force)
pdl, poundal	Poundal
ounce	Ounce = lbf / 16

Energy $[ML^2/T^2]$	
J, joule	Joule = 1 newton-metre
ftlbf	Foot-lbf
erg	Erg = 1e-7 joule
calorie	International Table Calorie
Btu	International Table Btu
therm	EEC therm
tonTNT	Energy in 1 ton TNT
kwh	Kilowatt hour

Power $[ML^2/T^3]$	
W, watt	Watt = 1 joule / second
Hp	Elec. Horsepower (746 W)

Temperature [t]	
degK, kelvin	`Kelvin (Base Unit)`
degC	Degree Celsius
degF	Degree Fahrenheit
degR, rankine	Degree Rankine
eV	Electron Volt

Pressure [$M/L/T^2$]	
Pa, pascal	Pascal = 1 newton / metre^2
MPa	Megapascal
GPa	Gigapascal
bar	Bar
kbar	Kilobar
Mbar	Megabar
atm	Standard atmosphere
torr	Torr = 1 mmHg
mHg	Metre of mercury
mmHg	Millimetre of mercury
inHg	Inch of mercury
inH2O	Inch of water
ftH2O	Foot of water
psi	Pound per square inch
ksi	Kilo-pound per square inch
psf	Pound per square foot

Volume [L^3]	
liter	Metre^3 / 1000
gal, gallon	Gallon (U.S.)

Angular	
rad	`Radian (base unit)`
rev	Full circle = 360 degree
deg, degree	Degree
arcmin	Arc minute = 1/60 degree
arcsec	Arc second = 1/360 degree
grade	Grade = 0.9 degree

The conversion expressions were obtained from References [3], [4], [5], and [6].

7.3 Physical Constants

The units system also defines several physical constants which are listed in the following table. The values for these were obtained from `https://en.wikipedia.org/wiki/List_of_physical_constants`. Although the APREPRO units system should adjust the values correctly for different systems, it is recommended that the "si" system be used to avoid any possible conversion errors.

Table 7.4: Physical Constants

Quantity	Value
Avogadro_constant	$6.02214076x10^{23}$ $/mol$
Bohr_magneton	$9.2740100783x10^{-24}$ J/T
Bohr_radius	$5.29177210903x10^{-11}$ m
Boltzmann_constant	$1.380649x10^{-23}$ $J/^{\circ}K$
Coulomb_constant	$8.9875517923x10^{9}$ $Nm^2 \cdot C^{-2}$
Faraday_constant	96485.3321233100184 C/mol
Fermi_coupling_constant	$1.166378710x10^{-5}$ GeV^{-2}
Hartree_energy	$4.3597447222071x10^{-18}$ J
Josephson_constant	$483597.8484x10^{9}$ Hz/V
Newtonian_constant_of_gravitation	$6.67430x10^{-11}$ $m^3/kg \cdot s^{-2}$
Gravitational_constant	$6.67430x10^{-11}$ $m^3/kg \cdot s^{-2}$
Planck_constant	$6.62607015x10^{-34}$ J/Hz
Rydberg_constant	10973731.568160 m
Rydberg_unit_of_energy	2.1798723611035$x10^{-}$18 J
Stefan_Boltzmann_constant	$5.670374419x10^{-8}$ $W \cdot m^{-2} \cdot {}^{\circ}K^{-4}$
Thomson_cross_section	$6.6524587321x10^{-29}$ m^2
W_to_Z_mass_ratio	0.88153
Wien_entropy_displacement_law_constant	$3.002916077x10^{-3}$ $m \cdot {}^{\circ}K$
Wien_frequency_displacement_law_constant	$5.878925757x10^{10}$ $Hz/^{\circ}K$
Wien_wavelength_displacement_law_constant	$2.897771955x10^{-3}$ $m \cdot {}^{\circ}K$
atomic_mass_constant	$1.66053906660x10^{-27}$ kg
atomic_mass_of_carbon_12	$1.99264687992x10^{-26}$ kg
characteristic_impedance_of_vacuum	376.730313668 Ω
classical_electron_radius	$2.8179403262x10^{-15}$ m
conductance_quantum	$7.748091729x10^{-5}$ S
cosmological_constant	$1.089x10^{-52}$ m^{-2}
electron_g_factor	-2.00231930436256
electron_mass	$9.1093837015x10^{-31}$ kg
elementary_charge	$1.602176634x10^{-19}$ C
fine_structure_constant	$7.2973525693x10^{-3}$
first_radiation_constant	$3.741771852x10^{-16}$ $W \cdot m^2$
hyperfine_transition_frequency_of_133Cs	139192631770 Hz
inverse_conductance_quantum	12906.40372 Ω
inverse_fine_structure_constant	137.035999084
magnetic_flux_quantum	$2.067833848x10^{-15}$ $V \cdot s$
molar_Planck_constant	$3.9903127128934314x10^{-10}$ $J \cdot s/mol$
molar_gas_constant	8.31446261815324 $J/mol/^{\circ}K$
molar_mass_constant	$0.99999999965x10^{-3}$ kg/mol
molar_mass_of_carbon_12	$11.9999999958x10^{-3}$ kg/mol
muon_g_factor	-2.0023318418
muon_mass	$1.883531627x10^{-28}$ kg
neutron_mass	$1.67492749804x10^{-27}$ kg
nuclear_magneton	$5.0507837461x10^{-27}$ J/T
proton_g_factor	5.5856946893
proton_mass	$1.67262192369x10^{-27}$ kg
proton_to_electron_mass_ratio	1836.15267343
quantum_of_circulation	$3.6369475516x10^{-4}$ m^2/s
reduced_Planck_constant	$1.054571817x10^{-34}$ $J \cdot s$
sec_radiation_constant	$1.438776877x10^{-2}$ $m \cdot {}^{\circ}K$

speed_of_light_in_vacuum	299792458 m/s
tau_mass	$3.16754x10^{-27}$ kg
top_quark_mass	$3.0784x10^{-25}$ kg
vacuum_electric_permittivity	$8.8541878128x10^{-12}$ F/m
vacuum_magnetic_permeability	$1.25663706212x10^{-6}$ $N \cdot A^{-2}$
von_Klitzing_constant	25812.80745 Ω
weak_mixing_angle	0.22290

7.4 Usage

The following example illustrates the basic usage of the APREPRO units conversion utility.

```
$ Aprepro Units Utility Example
$ {{Units(''shock'')}} ...Select the shock units system, use double brace to suppress echoing
$  NOTE:  Dimensions  -  {lout},  {mout},  {dout},  {pout}
...This will document what quantities are used in the file after it is run through Aprepro
{len1  = 10.0 * inch}  ...Define a length in an english unit (inches)
$ {len2  = 12.0~inch} ...~ is a synonym for * (multiplication)
Material  1,  Elastic Plastic,  {1890~kgpm3}  $  {dout}
   Youngs Modulus = {28.3e6~psi} $ pout
   Yield  Stress  = {30~ksi}
   Initial Veclocity = {10~mph} $ vout
   ...Define the density and material parameters in whatever units they are available
End
Point 100 {0.0}  {0.0}
Point 110 {len1} {0.0}
Point 120 {len1} {len2}
Point 130 {0.0}  {len1}
```

The output from this example input file is:

```
$  Aprepro  Units  Utility  Example
$  NOTE:  Dimensions  -  cm,  gram,  g/cc,  Mbar  ...The documentation of what quantities this file uses
$ 25.4
$ 30.48

Material 1, Elastic Plastic, 1.89 $ g/cc
   Youngs Modulus = 1.951216314 $ Mbar
   Yield  Stress  =  0.002068427188  ...All material parameters are now in consistent units
   Initial Velocity = 0.00044704 $ cm/usec
End

Point  100  0  0
Point  110  25.4 0
Point  120  25.4 30.48
Point  130  0   25.4   ...Lengths have all been converted to centimetres
```

The same input file can be used to output in SI units simply by changing Units command from *shock* to *si*. The output in SI units is:

```
$ Aprepro Units Utility Example
$  NOTE:  Dimensions  -  meter,  kilogram,  kg/m^3,  Pa
...Quantities are now output in standard SI units
```

```
$ 0.254
$ 0.3048

Material  1,  Elastic  Plastic,  1890  $  kg/m^3
   Youngs  Modulus =  1.951216314e+11 $ Pa
   Yield  Stress   =  206842718.8
   Initial Velocity = 4.4704 $ meter/sec
End

Point  100 0  0
Point  110 0.254  0
Point  120 0.254  0.3048
Point  130 0   0.254  ...Lengths have all been converted to metres
```

7.5 Additional Comments

A few additional comments and warnings on the use of the units system are detailed below.

Using only single braces for the {Units("unit_system")} function will print out the contents of the units header and conversion files. Each line in the output will be preceded by the current comment character which is $ by default.

The comment character can be changed by invoking APREPRO with the `-c` option. For example `aprepro -c# input_file output_file` will change the comment character at the beginning of the lines to #.

The temperature conversions are only valid for relative temperatures, for example, 100˜degC is equal to 180˜degF, not 212˜degF. The functions `CtoF(x)` and `FtoC(x)` can be used to convert absolute temperatures. For example,

```
100 C = {CtoF(100)} F
98.6 F = {FtoC(98.6)} C
```

produces the output:

```
100 C = 212 F
98.6 F = 37 C
```

Several variables are defined in the units system; they are all *immutable* variables, so it is not possible to redefine their values. However, if you inadvertently are using a variable with the same name as a variable defined in the units system, you may get the incorrect value of that variable since it cannot be redefined from the value set by the units system. You can enter the command `DUMP()` to see a list of all defined variables and their current value.

The APREPRO variable `_UNITS_SYSTEM` is defined to the name of the current units system that is loaded or "none" if no units system has been loaded.

8 Error, Warning, and Informational Messages

Several error, warning, and informational messages will be printed by APREPRO if certain conditions are encountered during the parsing of an input file. The messages are of the form:

```
Aprepro: Type: Message (file, line line#)
```

Where `Type` is `ERROR` for an error message, `WARN` for a warning message, or `INFO` for an informational message; `Message` is an explanation of the problem, `file` is the filename being processed at the time of the message, and `line#` is the number of the line within that file. Error messages are always output, Warning messages are output by default and can be turned off by using the `-W` or `--warning` command option, and Informational messages are turned off by default and can be turned on by using the `-M` or `--message` command option. (See Chapter refch:execution.)

8.1 Error Messages

Aprepro: ERROR: parse error (file, line line#) An unrecognized or ill-formed expression has been entered. Parsing of the file continues following this expression.

Aprepro: ERROR: Can't open 'file': No such file or directory The file specified in the include or import command cannot be found or does not exist. Aprepro will terminate processing following this error message.

Aprepro: ERROR: Can't open 'file': Permission denied The file specified in the include, import, or output command could not be opened due to insufficient permission. Aprepro will terminate processing following this error message.

Aprepro: ERROR: Improperly Nested ifdef/ifndef statements (file, line line#) An invalid ifdef/ifndef block has been detected. Typically this is caused by an extra endif or else statement.

Aprepro: ERROR: Zero divisor (file, line line#) An expression tried to divide by zero. The expression is given the value of the dividend and parsing continues.

Aprepro: ERROR: Invalid units system type. Valid types are: 'si', 'cgs', 'cgs-ev', 'shock', 'swap', 'ft-lbf- The units system specified in the command could not be found. This is most likely due to a misspelling of the units system name.

Aprepro: ERROR: function (file, line line#) DOMAIN error: Argument out of domain The arithmetic function function has been passed an invalid argument. For example, the above error would be printed for each of the expressions:

```
{sqrt(-1.0)} {log(0.0)} {asin(1.1)}
```

since the arguments are out of the valid domain for the function. The value returned by the function following an error is system-dependent. See the function's man page on your system for more information.

8.2 Warning Messages

Aprepro: WARN: Undefined variable 'variable' (file, line line#) A variable is used in an expression before it has been defined. The variable is set equal to zero or the null string ("") and parsing continues.

Aprepro: WARN: Variable 'variable' redefined (file, line line#) A previously defined variable is being set equal to a new value.

Aprepro: (IMMUTABLE) Variable 'variable' is immutable and cannot be modified (file, line line#) The value of a variable that was created as an immutable variable was modified. No value will be returned by the expression. See page 11 and page 6.2.9 for a description of immutable variables.

8.3 Informational Messages

Aprepro: INFO: Included File: 'filename' (file, line line#) The file filename is being included at line line# of file file. This message will also be printed during the execution of a loop block since temporary files are used to implement the looping function, and during the execution of the units conversion and material database access routines.

9 Examples

9.1 Mesh Generation Input File

The first example shown in this section is the point definition portion of an input file for a mesh generation code. First, the locations of the arc center points 1, 2, and 5 are specified. Then, the radius of each arc is defined ({Rad1}, {Rad2}, and {Rad5}). Note that the lines are started with a dollar sign, which is a comment character to the mesh generation code. Following this, the locations of points 10, 20, 30, 40, and 50 are defined in algebraic terms. Then, the points for the inner wall are defined simply by subtracting the wall thickness from the radius values.

```
Title
Example for Aprepro
$ Center Points
Point 1 {x1 = 6.31952E+01} {y1 = 7.57774E+01}
Point 2 {x2 = 0.00000E+00} {y2 = -3.55000E+01}
Point 5 {x5 = 0.00000E+00} {y5 = 3.62966E+01}
$ Width = {Width = 3.0}
... Wall thickness
$ Rad5 = {Rad5 = 207.00}
$ Rad2 = {Rad2 = 203.2236}
$ Rad1 = {Rad1 = Rad2 - dist(x1,y1; x2,y2)}
$ Angle between Points 2 and 1: {Th12 = atan2d((y1-y2),(x1-x2))}
Point 10 0.00 {y5 - Rad5}
Point 20 {x20 = x1+Rad1} {y5-sqrt(Rad5^2-x20^2)}
Point 30 {x20} {y1}
Point 40 {x1+Rad1*cosd(Th12)} {y1+Rad1*sind(Th12)}
Point 50 0.00 {y2 + Rad2}
$ Inner Wall (3 mm thick)
$ {Rad5 -= Width}
$ {Rad2 -= Width}
$ {Rad1 -= Width}
... Rad1, Rad2, and Rad5 are reduced by the wall thickness
Point 110 0.00 {y5 - Rad5}
Point 120 {x20 = x1+Rad1} {y5-sqrt(Rad5^2-x20^2)}
Point 130 {x20} {y1}
Point 140 {x1+Rad1*cosd(Th12)} {y1+Rad1*sind(Th12)}
Point 150 0.00 {y2 + Rad2}
```

The output obtained from processing the above input file by Aprepro is shown below.

```
Title
Example for Aprepro
$ Center Points
Point 1 63.1952 75.7774
Point 2 0 -35.5
Point 5 0 36.2966
$ Rad5 = 207
$ Rad2 = 203.2236
$ Rad1 = 75.2537088
$ Angle between Points 2 and 1: 60.40745947
Point 10 0.00 -170.7034
Point 20 138.4489088 -117.5893956
Point 30 138.4489088 75.7774
```

```
Point  40  100.3576382  141.214957
Point  50  0.00  167.7236
$  Inner Wall  (3  mm  thick)
$  204
$  200.2236
$  72.2537088
Point  110  0.00  -167.7034
Point  120  135.4489088  -116.2471416
Point  130  135.4489088  75.7774
Point  140  98.87615226  138.6062794
Point  150  0.00  164.7236
```

9.2 Macro Examples

Aprepro can also be used as a simple macro definition program. For example, a mesh input file may have many lines with the same number of intervals. If those lines are defined using a variable name for the number of intervals, then preprocessing the file with Aprepro will set all of the intervals to the same value, and simply changing one value will change them all. The following input file fragment illustrates this

```
$ {intA = 11} {intB = int(intA / 2)} line 10 str 10 20 0 {intA}

line 20 str 20 30 0 {intB}
line 30 str 30 40 0 {intA}
line 40 str 40 10 0 {intB}
```

Which when processed looks like:

```
$  11  5

line 10 str 10 20 0 11
line 20 str 20 30 0 5
line 30 str 30 40 0 11
line 40 str 40 10 0 5
```

9.3 Command Line Variable Assignment

This example illustrates the use of assigning variables on the command line. While generating a complicated 2D or 3D mesh, it is often necessary to reposition the mesh using GREPOS. If the following file called shift.grp is created:

```
Offset  X  {xshift}  Y  {yshift}
Exit
```

then, the mesh can be repositioned simply by typing:

```
Aprepro  xshift=100.0  yshift=-200.0  shift.grp  temp.grp
Grepos  input.mesh  output.mesh  temp.grp
```

9.4 Loop Example

This example illustrates the use of the loop construct to print a table of sines and cosines from 0 to 90 degrees in 5 degree increments.

Note the use of the double braces at the end of the first line to suppress the output of the initialization of the *angle* variable.

Input:

```
$ Test looping - print sin, cos from 0 to 90 by 5
{Loop(19, angle, 0, 5)}
{angle} {sind(angle)} {cosd(angle)}
{EndLoop}
```

Output:

```
$ Test looping - print sin, cos from 0 to 90 by 5
 0  0    1
 5  0.08715574275 0.9961946981
10  0.1736481777  0.984807753
15  0.2588190451  0.9659258263
20  0.3420201433  0.9396926208
25  0.4226182617  0.906307787
30  0.5    0.8660254038
35  0.5735764364  0.8191520443
40  0.6427876097  0.7660444431
45  0.7071067812  0.7071067812
50  0.7660444431  0.6427876097
55  0.8191520443  0.5735764364
60  0.8660254038  0.5
65  0.906307787  0.4226182617
70  0.9396926208  0.3420201433
75  0.9659258263  0.2588190451
80  0.984807753  0.1736481777
85  0.9961946981  0.08715574275
90  1    6.123233765e-17
```

9.5 If Example

This example illustrates the if conditional construct.

```
{diff = sqrt(3)*sqrt(3) - 3}
$ Test if - else lines
{if(sqrt(3)*sqrt(3) - 3 == diff)}
 complex if works
{else}
 complex if does not work
{endif}

{if (sqrt(4) == 2)}
 {if (sqrt(9) == 3)}
  {if (sqrt(16) == 4)}
    square roots work
```

```
  {else}
    sqrt(16) does not work
  {endif}
 {else}
   sqrt(9) does not work
 {endif}
{else}
  sqrt(4) does not work
{endif}

{v1 = 1} {v2 = 2}
{if (v1 == v2)}
  Bad if
  {if (v1 != v2)}
   should not see (1)
  {else}
   should not see (2)
  {endif}
   should not see (3)
{else}
  {if (v1 != v2)}
   good nested if
  {else}
   bad nested if
  {endif}
  good
  make sure it is still good
{endif}
```

The output of this is:

```
-4.440892099e-16
$ Test if - else lines
 complex if works

       square roots work

1 2
     good nested if
  good
  make sure it is still good
```

9.6 Aprepro Exodus Example

The input below illustrates the use of the `exodus_meta` and `exodus_info` functions.

```
{exodus_meta("exodus.g")}

        Title = {ex_title}
    Dimension = {ex_dimension}
   Node Count = {ex_node_count}
Element Count = {ex_element_count}

Element Block Info:
```

```
        Count = {ex_block_count}
        Names = {ex_block_names}
     Topology = {ex_block_topology}
          Ids = {print_array(transpose(ex_block_ids))}
{print_array(ex_block_info)}

Nodeset Info:
        Count = {ex_nodeset_count}
        Names = {ex_nodeset_names}
          Ids = {print_array(transpose(ex_nodeset_ids))}
{print_array(ex_nodeset_info)}

Sideset Info:
        Count = {ex_sideset_count}
        Names = {ex_sideset_names}
          Ids = {print_array(transpose(ex_sideset_ids))}
{print_array(ex_sideset_info)}

Timestep Info:
        Count = {ex_timestep_count}
        Times = {print_array(transpose(ex_timestep_times))}

NOTE: Array index are 0-based by default; get_word is 1-based... {i_=0}
{loop(ex_block_count)}
Element block {ex_block_ids[i_]} named '{get_word(++i_,ex_block_names,",")}' has topology '{get_word(i_,ex_bl
{endloop}

Extract Information Records using begin ... end
{info1 = exodus_info("exodus.g", "start extract", "end extract")}
Rescan String:
{rescan(info1)}

Extract Information Records using prefix and then rescan:
{info2 = exodus_info("exodus.g", "PRE: ")}
{rescan(info2)}
```

When processed by APREPRO, the following output will be produced:

```
        Title = GeneratedMesh: 2x3x4+shell:xYz+nodeset:XyZ+sideset:xyzXYZ+times:7+variables:glob
    Dimension = 3
   Node Count = 60
Element Count = 50

Element Block Info:
        Count = 4
        Names = inner_core,Shell-MinX,Shell-MaxY,Shell-MinZ
     Topology = hex8,shell4,shell4,shell4
          Ids =  1 2 3 4
 1 24 8 0
 2 12 4 1
 3 8 4 1
 4 6 4 1

Nodeset Info:
        Count = 3
        Names = nodelist_1,nodelist_2,nodelist_3
```

```
          Ids =  1 2 3
 1 20 20
 2 15 15
 3 12 12

Sideset Info:
        Count = 6
        Names = surface_1,surface_2,surface_3,surface_4,surface_5,surface_6
          Ids =  1 2 3 4 5 6
 1 12 48
 2 8 32
 3 6 24
 4 12 48
 5 8 32
 6 6 24

Timestep Info:
        Count = 7
        Times =  0 1 2 3 4 5 6

NOTE: Array index are 0-based by default; get_word is 1-based... 0
Element block 1 named 'inner_core' has topology 'hex8'
Element block 2 named 'Shell-MinX' has topology 'shell4'
Element block 3 named 'Shell-MaxY' has topology 'shell4'
Element block 4 named 'Shell-MinZ' has topology 'shell4'

Extract Information Records using begin ... end
...(The next three lines are the "raw" info records)
loop(6)
a_++^2
endloop

Rescan String:
...(This shows the "rescanned" info records)
0
1
4
9
16
25

Extract Information Records using prefix and then rescan:
...(The next seven lines are the "raw" info records)
Units("si")
1 foot = 1~foot lout
12 inch = 12~inch lout
1728 in^3 = 1728~in^3 Vout
10~foot * 14~in
60 mph = 60~mph vout
88 fps = 88~fps vout

... (this is the output from the "rescanned" info records)
1 foot = 0.3048 meter
12 inch = 0.3048 meter
1728 in^3 = 0.02831684659 meter^3
```

```
1.0838688
60 mph = 26.8224 meter/sec
88 fps = 26.8224 meter/sec
```

9.7 Aprepro Test File Example

The input below is from one of the aprepro test files. It illustrates looping, assignments, trigonometric functions, ifdefs, string processing, and many other APREPRO constructs.

```
$ Test program for Aprepro
$
Test number representations
{1} {10e-1} {10.e-1} {.1e+1} {.1e1}
{1} {10E-1} {10.E-1} {.1E+1} {.1E1}

Test assign statements:
{_a = 5} {b=_a} $ Should print 5 5
{_a +=b} {_a}  $ Should print 10 10
{_a -=b} {_a} $ Should print 5 5
{_a *=b} {_a} $ Should print 25 25
{_a /=b} {_a} $ Should print 5 5
{_a ^=b} {_a} $ Should print 3125 3125
{_a = b} {_a**=b} {_a} $ Should print 5 3125 3125

Test trigonometric functions (radians)
{pi = d2r(180)} {atan2(0,-1)} {4*atan(1.0)} $ Three values of pi
{_a = sin(pi/4)} {pi-4*asin(_a)} $ sin(pi/4)
{_b = cos(pi/4)} {pi-4*acos(_b)} $ cos(pi/4)
{_c = tan(pi/4)} {pi-4*atan(_c)} $ tan(pi/4)

Test trigonometric functions (degrees)
{r2d(pi)} {pid = atan2d(0,-1)} {4 * atand(1.0)}
{ad = sind(180/4)} {180-4*asind(ad)} $ sin(180/4)
{bd = cosd(180/4)} {180-4*acosd(bd)} $ cos(180/4)
{cd = tand(180/4)} {180-4*atand(cd)} $ tan(180/4)

Test max, min, sign, dim, abs
{pmin = min(0.5, 1.0)} {nmin = min(-0.5, -1.0)} $ Should be 0.5, -1
{pmax = max(0.5, 1.0)} {nmax = max(-0.5, -1.0)} $ Should be 1.0, -0.5
{zero = 0} {sign(0.5, zero) + sign(0.5, -zero)} $ Should be 0 1
{nonzero = 1} {sign(0.5, nonzero) + sign(0.5, -nonzero)} $ Should be 1 0
{dim(5.5, 4.5)} {dim(4.5, 5.5)} $ Should be 1 0

$ Test ifdef lines
{ifyes = 1} {ifno = 0}
 {Ifdef(ifyes)}
This line should be echoed. (a)
 {Endif}
This line should be echoed. (b)
     {Ifdef(ifno)}
This line should not be echoed
   {Endif}
This line should be echoed. (c)
  {Ifdef(ifundefined)}
```

```
This line should not be echoed
        {Endif}
This line should be echoed. (d)

$ Test if - else lines
             {Ifdef(ifyes)}
This line should be echoed. (1)
   {Else}
This line should not be echoed (2)
 {Endif}
  {Ifdef(ifno)}
This line should not be echoed. (3)
 {Else}
This line should be echoed (4)
   {Endif}

$ Test if - else lines
 {Ifndef(ifyes)}
This line should not be echoed. (5)
  {Else}
This line should be echoed (6)
   {Endif}
    {Ifndef(ifno)}
This line should be echoed. (7)
 {Else}
This line should not be echoed (8)
  {Endif}
$ Lines a, b, c, d, 1, 4, 6, 7 should be echoed
$ Check line counting -- should be on line 74: {Parse Error}
{ifdef(ifyes)} {This should be an error}
{endif}

$ Test int and [] (shortcut for int)
{int(5.01)} {int(-5.01)}
{[5.01]} {[-5.01]}

$ Test looping - print sin, cos from 0 to 90 by 5
{Loop(19, angle, 0, 5)}
{_angle} {_sa=sind(_angle)} {_ca=cosd(_angle)} {hypot(_sa, _ca)}
{EndLoop}

$$$$ Test formatting and string concatenation {{_SAVE = _FORMAT}}
{loop(20, _i, 1)}
{IO(_i)} Using the format {_FORMAT = "%." // tostring(_i) // "g"}, PI = {PI}
{endloop}
Reset format to default: {_FORMAT = _SAVE}

$$$$ Test string rescanning and executing
{ECHO(OFF)}
{Test = ' This is line 1: {a = atan2(0,-1)}
        This is line 2: {sin(a/4)}
 This is line 3: {cos(a/4)}'}
{Test2 = 'This has an embedded string: {T = "This is a string"}'}
{ECHO(ON)}
Original String:
```

```
{Test}
Rescanned String:
{rescan(Test)}
Original String:
{Test2}
Print Value of variable T = {T}
Rescanned String:
{rescan(Test2)}
Print Value of variable T = {T}

Original String: {t1 = "atan2(0,-1)"}
Executed String: {execute(t1)}

string = {_string = " one two, three"}
delimiter "{_delm = " ,"}"
word count = {word_count(_string,_delm)}
second word = "{get_word(2,_string,_delm)}"

string = {_string = " (one two, three * four - five"}
delimiter "{_delm = " ,(*-"}"
word count = {word_count(_string,_delm)}
second word = "{get_word(2,_string,_delm)}"


string = {_string = " one two, three"}
delimiter "{_delm = " ,"}"
word count = { iwords = word_count(_string,_delm)}

 {loop(iwords, _n, 1)}
word {_n} = "{get_word(_n,_string,_delm)}"
   {endloop}

$ Check parsing of escaped braces...
\{ int a = b + {PI/2} \}
\{ \}
```

When processec by Aprepro, there will be four warning messages and two error messages:

```
Aprepro: ERROR: syntax error, unexpected UNDVAR (test.inp_app, line 78)
Aprepro: WARNING: Undefined variable 'This' (test.inp_app, line 79)
Aprepro: ERROR: syntax error, unexpected UNDVAR (test.inp_app, line 79)
Aprepro: WARNING: User-defined Variable 'a' redefined (_string_, line 0)
Aprepro: WARNING: Undefined variable 'T' (test.inp_app, line 203)
Aprepro: WARNING: Undefined variable 'new_var' (test.inp_app, line 238)
```

The processed output from this example is:

```
$ Aprepro (Revision: 2.28) Mon Jan 21 10:58:23 2013
$ Test program for Aprepro
$
Test number representations
1 1 1 1 1
1 1 1 1 1

Test assign statements:
5 5 $ Should print 5 5
```

```
10 10  $ Should print 10 10
5 5 $ Should print 5 5
25 25 $ Should print 25 25
5 5 $ Should print 5 5
3125 3125 $ Should print 3125 3125
5 3125 3125 $ Should print 5 3125 3125

Test trigonometric functions (radians)
3.141592654 3.141592654 3.141592654 $ Three values of pi
0.7071067812 4.440892099e-16 $ sin(pi/4)
0.7071067812 0 $ cos(pi/4)
1 0 $ tan(pi/4)

Test trigonometric functions (degrees)
180 180 180
0.7071067812 2.842170943e-14 $ sin(180/4)
0.7071067812 0 $ cos(180/4)
1 0 $ tan(180/4)

Test max, min, sign, dim, abs
0.5 -1 $ Should be 0.5, -1
1 -0.5 $ Should be 1.0, -0.5
0 1 $ Should be 0 1
1 0 $ Should be 1 0
1 0 $ Should be 1 0

$ Test ifdef lines
1 0
This line should be echoed. (a)
This line should be echoed. (b)
This line should be echoed. (c)
This line should be echoed. (d)

$ Test if - else lines
This line should be echoed. (1)
This line should be echoed (4)

$ Test if - else lines
This line should be echoed (6)
This line should be echoed. (7)
$ Lines a, b, c, d, 1, 4, 6, 7 should be echoed
$ Check line counting -- should be on line 74:

$ Test int and [] (shortcut for int)
5 -5
5 -5

$ Test looping - print sin, cos from 0 to 90 by 5
0 0 1 1
5 0.08715574275 0.9961946981 1
10 0.1736481777 0.984807753 1
15 0.2588190451 0.9659258263 1
20 0.3420201433 0.9396926208 1
25 0.4226182617 0.906307787 1
30 0.5 0.8660254038 1
```

```
35 0.5735764364 0.8191520443 1
40 0.6427876097 0.7660444431 1
45 0.7071067812 0.7071067812 1
50 0.7660444431 0.6427876097 1
55 0.8191520443 0.5735764364 1
60 0.8660254038 0.5 1
65 0.906307787 0.4226182617 1
70 0.9396926208 0.3420201433 1
75 0.9659258263 0.2588190451 1
80 0.984807753 0.1736481777 1
85 0.9961946981 0.08715574275 1
90 1 6.123233996e-17 1

$$$$ Test formatting and string concatenation
1 Using the format %.1g, PI = 3
2 Using the format %.2g, PI = 3.1
3 Using the format %.3g, PI = 3.14
4 Using the format %.4g, PI = 3.142
5 Using the format %.5g, PI = 3.1416
6 Using the format %.6g, PI = 3.14159
7 Using the format %.7g, PI = 3.141593
8 Using the format %.8g, PI = 3.1415927
9 Using the format %.9g, PI = 3.14159265
10 Using the format %.10g, PI = 3.141592654
11 Using the format %.11g, PI = 3.1415926536
12 Using the format %.12g, PI = 3.14159265359
13 Using the format %.13g, PI = 3.14159265359
14 Using the format %.14g, PI = 3.1415926535898
15 Using the format %.15g, PI = 3.14159265358979
16 Using the format %.16g, PI = 3.141592653589793
17 Using the format %.17g, PI = 3.1415926535897931
18 Using the format %.18g, PI = 3.14159265358979312
19 Using the format %.19g, PI = 3.141592653589793116
20 Using the format %.20g, PI = 3.141592653589793116
Reset format to default: %.10g

$$$$ Test string rescanning and executing

Original String:
 This is line 1: a = atan2(0,-1)
        This is line 2: sin(a/4)
 This is line 3: cos(a/4)
Rescanned String:
 This is line 1: 3.141592654
        This is line 2: 0.7071067812
 This is line 3: 0.7071067812
Original String:
This has an embedded string: T = "This is a string"
Print Value of variable T = 0
Rescanned String:
This has an embedded string: This is a string
Print Value of variable T = This is a string

Original String: atan2(0,-1)
Executed String: 3.141592654
```

```
string =  one two, three
delimiter " ,"
word count = 3
second word = "two"

string =  (one two, three * four - five
delimiter " ,(*-"
word count = 5
second word = "two"


string =  one two, three
delimiter " ,"
word count = 3

word 1 = "one"
word 2 = "two"
word 3 = "three"

$ Check parsing of escaped braces...
{ int a = b + 1.570796327 }
{ }
```

10 Aprepro Library Interface

The previous chapters have described the standalone version of APREPRO. The functionality provided in the standalone version can also be provided to other programs through the APREPRO library C++ interface. The APREPRO library provides a `SEAMS::Aprepro` class which has three methods for parsing the input:

1. Read from stdin, echo data to stdout. At end of input, the parsed output is available in the `Aprepro::parsing_results()` stream.
2. Read and parse a file. The entire file will be parsed with no output. After the file is parsed, the parsed output is available in the `Aprepro::parsing_results()` stream.
3. Read and parse a string containing the APREPRO input. The results from parsing the string are returned in the `Aprepro::parsing_results()` stream. Note that when using this method, you cannot use loops, if blocks, verbatim, and echo.

10.1 Adding basic APREPRO parsing to your application

The APREPRO capability is provided as a set of C++ classes. The main `SEAMS::Aprepro` class defined in the ***aprepro.h*** include file is the main interface used by external programs.

The basic method for using the `SEAMS::Aprepro` class is:

- create a `SEAMS::Aprepro` object
- parse the data
- retrieve the parsed data.

An example of this is shown below:

```
#include <aprepro.h>
int main(int argc, char *argv[])
{
  SEAMS::Aprepro aprepro;
  bool result = aprepro.parse_stream(infile, argv[argc-1]);
  if (result) {
     std::cout << "PARSING RESULTS:\n" << aprepro.parsing_results().str();
  }
}
```

10.2 Additional APREPRO parsing capabilities

In addition to the basic parsing shown above, additional capabilities are available including predefining variables, adding additional functions, and modifying the aprepro options.

10.2.1 Adding new variables

The `add_variable()` member function is used to define new variables that will be available during the aprepro parsing. The function signatures are:

```
void add_variable(const std::string &name, const std::string &value, bool is_immutable=false);
void add_variable(const std::string &name, double value, bool is_immutable=false);
```

Where `name` is the name of the variable to be defined, `value` is the value of the variable (either a double or a string). To create the variable as immutable, pass *true* as the third option.

10.2.2 Adding new functions

Additional functions can be made available during parsing as shown in the example below.

```
// This function is used below in the example showing how an
// application can add its own functions to an aprepro instance.
double succ(double i) {
  return ++i;
}
// EXAMPLE: Add a function to aprepro...
SEAMS::symrec *ptr = aprepro.putsym("succ", SEAMS::Aprepro::FUNCTION, 0);
ptr->value.fnctptr_d = succ;
ptr->info = "Return the successor to d";
ptr->syntax = "succ(d)";
```

Following this, the user can use the `succ(d)` command in the same way as any of the other Aprepro functions. This can be used to provide functions that access data internal to your program. The function will also appear in the `DUMP_FUNC()` function list.

10.2.3 Modifying Aprepro Execution Settings

The standalone Aprepro can be executed with several command line options which change the behavior of Aprepro as defined in Chapter 2. Similar behavior modifications are available in the Aprepro library via the `set_option()` command. The syntax is:

```
void set_option(const std::string &option);
```

Where `option` is one of:

`--debug`	Dump all variables, debug loops/if/endif
`--dumpvars`	Dump all variables at end of run
`--dumpvars_json`	Dump all variables at end of run in json format
`--version`	Print version number to stderr.
`--immutable`	All variables are immutable – cannot be modified
`--errors_fatal`	Exit program with nonzero status if errors are encountered
`--errors_and_warnings_fatal`	Exit program with nonzero status if warnings are encountered
`--require_defined`	Treat undefined variable warnings as fatal
`--one_based_index`	Array indexing is one-based (default = zero-based)
`--interactive`	Interactive use, no buffering
`--message`	Print INFO messages
`--info=file`	Output INFO messages (e.g. DUMP() output) to file.

`--nowarning`	Do not print warning messages.
`--copyright`	Print copyright message to stderr.
`--message`	Print INFO messages.
`--trace`	Trace program execution. Primarily for aprepro developer.
`--interactive`	Interactive use; do not buffer output.
`--exit_on`	End with Exit or EXIT or exit or Quit or QUIT or quit encountered in parsing stream.
`--include=`*file_or_path*	If a path is specified, then optionally prepend it to all included filenames; if a file is specified, the process the contents of the file before processing input files.
`--help`	Output the following text:

```
APREPRO PREPROCESSOR OPTIONS:
  --debug or -d: Dump all variables, debug loops/if/endif and keep temporary files
       --dumpvars or -D: Dump all variables at end of run
  --dumpvars_json or -J: Dump all variables at end of run in json format
        --version or -v: Print version number to stderr
      --immutable or -X: All variables are immutable--cannot be modified
   --errors_fatal or -f: Exit program with nonzero status if errors are encountered
 --errors_and_warnings_fatal or -F: Exit program with nonzero status if warnings are encountered
--require_defined or -R: Treat undefined variable warnings as fatal
--one_based_index or -1: Array indexing is one-based (default = zero-based)
    --interactive or -i: Interactive use, no buffering
    --include=P or -I=P: Include file or include path
                       : If P is path, then optionally prepended to all include filenames
                       : If P is file, then processed before processing input file
                       : variables defined in P will be immutable.
       --exit_on or -e: End when 'Exit|EXIT|exit' entered
           --help or -h: Print this list
        --message or -M: Print INFO messages
            --info=file: Output INFO messages (e.g. DUMP() output) to file.
      --nowarning or -W: Do not print WARN messages
--comment=char or -c=char: Change comment character to 'char'
      --copyright or -C: Print copyright message
  --keep_history or -k: Keep a history of aprepro substitutions.
                        (not for general interactive use)
         --quiet or -q: Do not print the header output line
               var=val: Assign value 'val' to variable 'var'
                        Use var=s̈val¨ for a string variable

 Units Systems: si, cgs, cgs-ev, shock, swap, ft-lbf-s, ft-lbm-s, in-lbf-s
 Enter DUMP() for list of user-defined variables
 Enter DUMP_FUNC() for list of functions recognized by aprepro
 Enter DUMP_PREVAR() for list of predefined variables in aprepro

 ->->-> Send email to gdsjaar@sandia.gov for aprepro support.
```

For additional functions that are rarely used, see the ***aprepro.h*** include file.

10.3 Aprepro Library Test/Example Program

A test program is provided with the APREPRO library which provides examples of the three parsing methods, defining variables, and defining functions. This is defined in the ***apr_test.cc*** file in the APREPRO library distribution. The contents of this file are shown below:

```cpp
#include <fstream>
#include <iostream>

#include "aprepro.h"

// This function is used below in the example showing how an
// application can add its own functions to an aprepro instance.
double succ(double i) { return ++i; }

int main(int argc, char *argv[])
{
  bool readfile = false;

  std::string output_file;

  SEAMS::Aprepro aprepro;

  // EXAMPLE: Add a function to aprepro...
  SEAMS::symrec *ptr   = aprepro.putsym("succ", SEAMS::Aprepro::SYMBOL_TYPE::FUNCTION, false);
  ptr->value.fnctptr_d = succ;
  ptr->info            = "Return the successor to d";
  ptr->syntax          = "succ(d)";

  // EXAMPLE: Add a couple variables...
  aprepro.add_variable("Greg", "Is the author of this code", true); // Make it immutable
  aprepro.add_variable("BirthYear", 1958);

  for (int ai = 1; ai < argc; ++ai) {
    std::string arg = argv[ai];
    if (arg == "-o") {
      output_file = argv[++ai];
    }
    else if (arg == "-i") {
      // Read from cin and echo each line to cout All results will
      // also be stored in Aprepro::parsing_results() stream if needed
      // at end of file.
      aprepro.ap_options.interactive = true;
      bool result                    = aprepro.parse_stream(std::cin, "standard input");
      if (result) {
        if (!output_file.empty()) {
          std::ofstream ofile(output_file);
          ofile << aprepro.parsing_results().str();
        }
        else {
          std::cout << aprepro.parsing_results().str();
        }
      }
    }
    else if (arg[0] == '-') {
      aprepro.set_option(argv[ai]);
    }
    else {
      // Read and parse a file.  The entire file will be parsed and
      // then the output can be obtained in an std::ostringstream via
      // Aprepro::parsing_results()
```

```
56        std::fstream infile(argv[ai]);
57        if (!infile.good()) {
58          if (!aprepro.ap_options.include_path.empty() && argv[ai][0] != '/') {
59            std::string filename = aprepro.ap_options.include_path + "/" + argv[ai];
60            infile.open(filename, std::fstream::in);
61          }
62        }
63        if (!infile.good()) {
64          std::cerr << "APREPRO: Could not open file: " << argv[ai] << '\n';
65          return 0;
66        }
67
68        bool result = aprepro.parse_stream(infile, argv[ai]);
69        if (result) {
70          if (!output_file.empty()) {
71            std::ofstream ofile(output_file);
72            ofile << aprepro.parsing_results().str();
73          }
74          else {
75            std::cout << aprepro.parsing_results().str();
76          }
77        }
78
79        readfile = true;
80      }
81    }
82    if (readfile) {
83      std::cerr << "Aprepro: There were " << aprepro.get_error_count()
84                << " errors detected during parsing.\n";
85      return aprepro.get_error_count() == 0 ? EXIT_SUCCESS : EXIT_FAILURE;
86    }
87
88    // Read and parse a string's worth of data at a time.
89    // Cannot use looping/ifs/... with this method.
90    std::string line, tmp;
91    while (std::cout << "\nexpression: " && std::getline(std::cin, tmp) && !tmp.empty()) {
92
93      line += tmp;
94
95      if (*tmp.rbegin() == '\\') {
96        line.erase(line.length() - 1);
97        continue;
98      }
99
100     line += "\n";
101
102     bool result = aprepro.parse_string_interactive(line);
103
104     if (result) {
105       std::string res_str = aprepro.parsing_results().str();
106       std::cout << "----------: " << res_str;
107
108       // Example showing how to get the substitution history for the current line.
109       if (aprepro.ap_options.keep_history) {
110         std::vector<SEAMS::history_data> hist = aprepro.get_history();
```

```
111         for (const auto &curr_history : hist) {
112
113           std::cout << curr_history.original << " was substituted with "
114                     << curr_history.substitution << " at index " << curr_history.index << '\n'
115         }
116
117         aprepro.clear_history();
118       }
119     }
120
121     aprepro.clear_results();
122
123     line.clear();
124   }
125   std::cerr << "Aprepro: There were " << aprepro.get_error_count()
126             << " errors detected during parsing.\n";
127   return aprepro.get_error_count() == 0 ? EXIT_SUCCESS : EXIT_FAILURE;
128 }
```

Bibliography

[1] Gregory D. Sjaardema, "Overview of the Sandia National Laboratories Engineering Analysis Code Access System," SAND92-2292, Sandia National Laboratories, Albuquerque, NM, January 1993, Reprinted August 1994.

[2] Larry A. Schoof and Victor R. Yarberry, "EXODUSII: A Finite Element Data Model," SAND92-2137, Sandia National Laboratories, Albuquerque, NM, September, 1994.[1]

[3] F. W. Walker, J. R. Parrington, and F. Feiner, "Nuclides and Isotopes, 14th Edition," General Electric Corporation, San Jose, California, 1989.

[4] J. C. Jaeger and N. G. W. Cook, *Fundamentals of Rock Mechanics*, Third Edition, Chapman and Hall Publishers, London, 1979.

[5] T. W. Lambe and R. V. Whitman, *Soil Mechanics,* John Wiley & Sons, New York, New York, 1969.

[6] G. R. Simpson, "Units Computer Program", copyright 1987.

[1] This document is very out of date. A new document is being prepared and a draft of the current state is available at http://sandialabs.github.io/seacas/exodusII-new.pdf.