SAND92-2291
Unlimited Release
Printed December 1992
Updated September 2, 1999

Distribution
Category UC-705

APREPRO: An Algebraic Preprocessor for Parameterizing Finite Element Analyses

Gregory D. Sjaardema
Solid and Structural Mechanics Department
Sandia National Laboratories
Albuquerque, New Mexico 87185

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, with some additional files, implement a units conversion system and a material database lookup 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

Tables

Figures

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 in the past two years to prepare parameterized files for finite element analyses using the Sandia National Laboratories SEACAS system[1]. The recent addition of the units conversion capability and the material database access routines have 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 **}**.

Aprepro is written in the C language. The BISON[2] and FLEX[3] programs are used to generate the parsing and lexical analysis subroutines, respectively. The initial implementation of *Aprepro* was based on the mfcalc example in the BISON manual. Aprepro has been ported to several UNIX systems including Hewlett Packard HP-UX, Cray Research Unicos, Sun Microsystems SunOS, Tenon MachTen, Digital Equipment Ultrix; and to non-UNIX systems including VAX VMS, Macintosh, and Amiga.

The remainder of this document is organized as follows:

- Section 2 documents the syntax recognized by *Aprepro*,
- Sections 3, 4, and 5 describe the operators, predefined variables, and functions,
- Section 6 describes the units conversion system,
- Section 7 describes the material database support routines,
- Section 8 describes the error messages output from *Aprepro*, and
- Section 9 presents some examples of *Aprepro* usage.
- Appendix A documents the command line options for *Aprepro*, and
- Appendix B lists the defined units abbreviations.

Intentionally Left Blank

2 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: ERR: parse error (*filename*, line *line#*)

to the terminal* and 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 section 5 on page 19 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, 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 section 4 on page 17.

  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†.

- **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

* 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.

† If the variable name begins with an underscore, no warning is output when the variable is redefined. Warnings can be turned off with the **-W** or **+warning** option.

the other type of quote. For example, **{'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 section 3 on page 13. 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 multi-line string, and the double quote (**"**) which delimits a single-line string.

- **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 page 23). 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 9.2 on page 44. 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 very limited string support. The only supported operations are assigning a variable equal to a string (**a = "This is a string"**) or a function that returns a string, and 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 or false. Relational expressions can only be used on the left-hand side of a conditional expression. A relational expression is simply two expressions of any kind separated by a relational operator (See "Relational Operators" on page 14.)

- **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. 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))
```

3 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 than addition. 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` for left associativity, the expression `a = b / c` would be evaluated as `a = (b / c)` for right associativity.

3.1 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

Table 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

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 section 5.1 on page 19 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 section 6 on page 25). It is more natural for some users to type **`12~metre`** than **`12*metre`**.

3.2 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 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`	The value of 'a' is set equal to a^b	4	right
`a**=b`	The value of 'a' is set equal to a^b	4	right

3.3 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 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

3.4 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. Logical 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

3.5 String Operators

The only supported string operator at this time is string concatenation which is denoted by `//`. If `a = "Hello"` and `b = "World"`, then:

```
c = a // " " // b
```

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

4 Predefined Variables

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

Table 5. Predefined Variables

Name	*Value*	*Description*
PI	3.14159265358979323846	π
PI_2	1.57079632679489661923	$\pi/2$
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	euler-mascheroni constant[1]
PHI	1.61803398874989484820	golden ratio $(\sqrt{5}+1)/2$
VERSION	Varies, string value	current version of Aprepro
_FORMAT	"%.10g"	default output format
_C_	"$"	default comment character

[1]The euler-mascheroni constant is defined as the limit of $1+\frac{1}{2}+\ldots+\frac{1}{s}-\log s$ as s approaches infinity.

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. 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.

Table 6. 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

Table 6. Effect of various output format specifications

Format	*PI Output*	*1.0 Output*
`%.10d`	1413754136	0000000000

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* automatically changes the value of the comment variable to match the character specified on the command line.

* See appendix A on page 51.

5 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.

5.1 Mathematical Functions

The following mathematical functions are available in *Aprepro*.

Table 7. Mathematical Functions

Syntax	***Description***
abs(x)	Calculates the absolute value of x. $\|x\|$
acos(x)	Calculates the inverse cosine of x, returns radians
acosd(x)	Calculates the inverse cosine of x, returns degrees
acosh(x)	Calculates the inverse hyperbolic cosine of x
asin(x)	Calculates the inverse sine of x, returns radians
asind(x)	Calculates the inverse sine of x, returns degrees
asinh(x)	Calculates the inverse hyperbolic sine of x
atan(x)	Calculates the inverse tangent of x, returns radians
atan2(y,x)	Calculates the inverse tangent of y/x, returns radians
atan2d(x)	Calculates the inverse tangent of x, returns degrees
atand(y,x)	Calculates the inverse tangent of y/x, returns degrees
atanh(x)	Calculates the inverse hyperbolic tangent of x
ceil(x)	Calculates the smallest integer not less than x
cos(x)	Calculate the cosine of x, with x in radians
cosd(x)	Calculate the cosine of x, with x in degrees
cosh(x)	Calculates the hyperbolic cosine of x
d2r(x)	Converts degrees to radians.
dim(x,y)	Calculates x - min(x,y).
dist(x1,y1, x2,y2)	Calculates $\sqrt{((x_1 - x_2)^2 + (y_1 - y_2)^2)}$

Table 7. Mathematical Functions

Syntax	*Description*
exp(x)	Calculates e^x (Exponential)
floor(x)	Calculates the largest integer not greater than x.
fmod(x,y)	Calculates the floating-point remainder of x/y.
hypot(x,y)	Calculates $\sqrt{x^2 + y^2}$
int(x), [x]	Calculates the integer part of x truncated toward 0.
lgamma(x)	Calculates $\log(\Gamma(x))$
ln(x), log(x)	Calculates the natural (base e) logarithm of x.
logp1(x)	Calculates log(1+x)
log10(x)	Calculates the base 10 logarithm of x.
max(x,y)	Calculates the maximum of x and y.
min(x,y)	Calculates the minimum of x and y.
nint(x)	Rounds x to nearest integer. <0.5 down, >=0.5 up
polarX(r,a)	Calculates $r \times \cos(a)$, a is in degrees
polarY(r,a)	Calculates $r \times \sin(a)$, a is in degrees
r2d(x)	Converts radians to degrees.
rand(xl,xh)	Calculates a random number between xl and xh.
sign(x,y)	Calculates $x \times \text{sgn}(y)$
sin(x)	Calculates the sine of x, with x in radians.
sind(x)	Calculates the sine of x, with x in degrees.
sinh(x)	Calculates the hyperbolic sine of x
sqrt(x)	Calculates the square root of x.
tan(x)	Calculates the tangent of x, with x in radians.
tand(x)	Calculates the tangent of x, with x in radians.
tanh(x)	Calculates the hyperbolic tangent of x.
julday(mm, dd, yy)	Calculates the julian day corresponding to mm/dd/yy.
juldayhms(mm, dd, yy, hh, mm, ss)	Calculates the julian day corresponding to mm/dd/yy at hh:mm:ss
Vangle(x1,y1, x2,y2)	Calculates the angle between the vector $x_1\hat{\boldsymbol{i}} + y_1\hat{\boldsymbol{j}}$ and $x_2\hat{\boldsymbol{i}} + y_2\hat{\boldsymbol{j}}$. returns radians.
Vangled(x1,y1, x2,y2)	Calculates the angle between the vector $x_1\hat{\boldsymbol{i}} + y_1\hat{\boldsymbol{j}}$ and $x_2\hat{\boldsymbol{i}} + y_2\hat{\boldsymbol{j}}$. returns degrees.

5.2 String Functions

A few useful string functions are available:

tolower(svar)	Translates all uppercase characters in **svar** to lowercase. 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.
tostring(x)	Returns a string representation of the numerical varaible **x**. The variable **x** is unchanged.
execute(svar)	**svar** is parsed and executed as if it were a line read from the input file. For example, if **svar = "b=sqrt(25.0)"**, then **{execute(svar)}** returns the value 5 and sets **b = 5**. The expression **svar** is enclosed in delimiters prior to being executed and it must be a valid expression or an error message will be printed.
rescan(svar)	Similar to **execute(svar)**, except that **svar** is not enclosed in delimiters prior to being executed. For example, if **svar = "Point {1+5} {sqrt(5)} {sqrt(6)}"**, then **{rescan(svar)}** would print: **Point 6 2.236067977 2.449489743**. The difference between **execute(sv1)** and **rescan(sv2)** is that **sv1** must be a valid expression, but **sv2** can contain zero or more expressions.
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.
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**
word_count(svar,del)	Returns the number of words in **svar**. Words are separated by one or more of the characters in the string variable **del**
strtod(svar)	Returns a double-precision floating-point number equal to the value represented by the character string pointed to by **svar**.
error(svar)	Outputs the string **svar** to stderr and then terminates the code with an error exit status.

The following example shows the use of some of the string functions. The lines beginning with the string "Output>" show the output from *Aprepro* resulting from entering the previous line.

```
{t1 = "ATAN2"} {t2 = "(0, -1)"}
Output> ATAN2 (0, -1)
{t3 = tolower(t1//t2)}
Output> atan2(0, -1)
   ...The variable t3 is equal to the string atan2(0, -1)
{execute(t3)}
Output> 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 first 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}`**.

5.3 Additional Functions

- **File Inclusion:** *Aprepro* can read input from multiple files using the **`include()`** and **`cinclude()`** functions. If a line of the form:

```
{include("filename")}
{include(string_variable)}
```

* Described in section 5.3 on page 22
† Described in section 5.3 on page 22

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. 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** and **cinclude** is that if ***filename*** does not exist, **include** 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. Approximately 16 levels of file inclusion can be used. 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 file.

- **Conditionals:** Portions of an input file can be conditionally processed through the use of the **{Ifdef(variable)}** or **{Ifndef(variable)}** constructs. The syntax is:

```
{Ifdef(variable)}
   ...Lines processed if 'variable' is  not equal to 0
{Else}
   ...Lines processed if 'variable' is equal to 0 or undefined
{Endif}
{Ifndef(variable)}
   ...Lines processed if 'variable' is equal to 0 or undefined
{Else}
   ...Lines processed if 'variable' is not equal to 0
{Endif}
```

  The **{Else}** is optional. Note that if **variable** is undefined, its value is equal to zero. **Ifdef** constructs can be nested up to approximately 16 levels. A warning message will be printed if improper nesting is detected. **{Ifdef(variable)}**, **{Ifndef(variable)}**, **{Else}**, and **{Endif}** are the only text parsed on a line. Text following these on the same line is ignored.

- **Loops:** Repeated processing of a group of lines can be controlled with the **{loop(*control*)}**, **{endloop}** commands. The syntax is:

```
{loop(variable)}
   ...Process these lines 'variable' times
{endloop}
```

  Loops can be nested. A numerical variable or constant must be specified as the loop control specifier. You cannot use an algebraic expression such as **{loop(3+5)}**.

- **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).**

- **VERBATIM:** The printing of all lines to the output file without processing can be controlled through the use of the **{VERBATIM(ON)} {VERBATIM(OFF)}** command. 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

- **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.

6 Units Conversion System

Although great effort has been expended to ensure that the units conversion system is accurate and consistent, the author does not make any warranty expressed or implied, or assume any liability or responsibility for the use of this software. If any errors are discovered in this software, please contact the author.

6.1 Introduction

The units conversion system in *Aprepro* is implemented as a set of files that define 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*.

Files have been defined for seven consistent units systems 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 8: 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*
Temp.	kelvin	kelvin	eV	kelvin	*Tout*
Velocity	metre/sec	cm/sec	cm/sec	cm/usec	*vout*
Accel.	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*

* This can also be written as **`8~foot`** since the symbol **`~`** has been defined to be the multiplication operator.

Table 8: Units Systems and Corresponding Output Format--Metric

Quantity	**si**	**cgs**	**cgs-ev**	**shock**	*string*
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 9: 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*
Temp.	rankine	rankine	rankine	*Tout*
Velocity	inch/sec	foot/sec	foot/sec	*vout*
Accel.	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*
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. This will search the standard locations on your system for the correct files to include.

6.2 Units Conversion Implementation

The units conversion system in *Aprepro* is implemented simply as a set of files that are selectively included by a function call in *Aprepro*. There are two types of files used. The first file type is a header file which defines the base units (metre, second, kg, radian, and kelvin) in terms of the desired output formats, and the output format string variables (lout, mout, ...). There is a different header file for each unit system. The **`in-lbf-s`** header file is shown below as an example:

```
{_C_} This is the in-lbf-s units file: inch, sec, lbf
```

```
{_C_} Outputs:
{_C_}   Time:        {tout = “second”}
{_C_}   Length:      {lout = “inch”}
{_C_}   Accel:       {aout = “in/sec^2”}
{_C_}   Mass:        {mout = “lbf-sec^2/in”}
{_C_}   Force:       {fout = “lbf”}
{_C_}   Velocity:    {vout = “in/sec”}
{_C_}   Volume:      {Vout = “in^3”}
{_C_}   Density:     {dout = “lbf-sec^2/in^4”}
{_C_}   Energy:      {eout = “inch-lbf”}
{_C_}   Power:       {Pout = “inch-lbf / sec”}
{_C_}   Pressure:    {pout = “psi”}
{_C_}   Temp:        {Tout = “degR”}
{_C_}   Angular:     {Aout = “radian”}
{_C_}
{_C_} 1 meter  = {m = 1 / 2.54e-2} {lout}
{_C_} 1 second = {sec = 1} {tout}
{_C_} 1 kg     = {kg = 1/4.5359237e-1/(9.806650*m/sec^2)} {mout}
{_C_} 1 kelvin = {degK = 1.8} {Tout}
{_C_} 1 radian = {rad = 1} {Aout}
```

Note that this file defines the output units string variables at the top of the file and then defines the base units in terms of the output units at the bottom of the file. This is the only file that must be created to implement a new units system. The name of the header file matches the name of the units system and it must be all lowercase.

The second file is called the **conversion** file. This file contains the equations defining the different units in terms of the base units. This is the only file that must be changed to add a new unit abbreviation to the system unless a new base unit is added, in which case all of the files must be modified. A short excerpt of this file is shown below:

```
{_C_}{_C_}{_C_}  Length (L)
{_C_} 1 Meter= {meter = metre = m} {lout}
{_C_} 1 cm   = {cm = centimeter = centimetre = m / 100} {lout}
{_C_} 1 mm   = {mm = millimeter = millimetre = m / 1000} {lout}
{_C_} 1 um   = {um = micrometer = micrometre = m / 1e6} {lout}
{_C_} 1 km   = {km = kilometer  = kilometre = 1000 * m} {lout}
{_C_} 1 foot = {ft = foot = .3048 * m} {lout}
{_C_} 1 mile = {mi = mile = ft * 5280} {lout}
{_C_} 1 yard = {yd = yard = ft * 3} {lout}
{_C_} 1 inch = {in = inch = ft / 12} {lout}
{_C_} 1 mil  = {mil = inch / 1000} {lout}
```

This segment is the portion of the conversion file which defines the length conversions. The expression **{_C_}** at the beginning of each line of the header and conversion files is a string variable that is given the current value of the comment character. In this way, the files can be written in a generic format that can be used as input for several codes. Each expression in the file defines a unit abbreviation in terms of a previously defined unit. For example, the third line of the file defines the abbreviations **cm**, **centimeter**, and **centimetre** in terms of the **metre** which is a base unit. The eighth line of the file defines the abbreviations **mile** and **mi** in terms of the foot which is earlier defined in terms of the meter. For ease of

verification of the units files, they are written in such a way that the output is somewhat self-explanatory, for example, if the SI system is being used, the above lines would result in the following output:

```
$$$  Length (L)
$ 1 Meter              = 1 meter
$ 1 cm                 = 0.01 meter
$ 1 mm                 = 0.001 meter
$ 1 um                 = 1e-06 meter
$ 1 km                 = 1000 meter
$ 1 foot               = 0.3048 meter
$ 1 mile               = 1609.344 meter
$ 1 yard               = 0.9144 meter
$ 1 inch               = 0.0254 meter
$ 1 mil                = 2.54e-05 meter
```

which is more understandable than if a bunch of numbers were output. The conversion expressions in this file were obtained from References 6, 7, 8, and 9.

When *Aprepro* processes the function call **{Units("unit_system")}**, it first searches for the requested header file (which has the same name as the unit system) in the directories defined by the environment variable **MATSPATH** or the default location if **MATSPATH** is not defined. The first matching file is used. It then searches for the conversion file in the same directories. Units files other than those currently supported can be used by modifying the environment variable **MATSPATH**. For example the following C-shell command will cause *Aprepro* to first search the current directory, then your mats subdirectory, and finally the default units directory for the specified units system files:

```
setenv MATSPATH ".:~/:/usr/local/eng_sci/mats"
```

The units files must be in a directory called **units** under the directories specified in the **MATSPATH** environment variable*. Therefore, it is possible to have a personal copy of a header file to define a new unit system and still use the global conversion file.

The units conversion files are in the SEACAS[1] code management system which is maintained by CVS[10]. CVS maintains a complete change log and the history of previous changes so that traceability is maintained.

6.3 Usage

The following example illustrates the basic usage of the units conversion utility in *Aprepro*.

```
$ Aprepro Units Utility Example
$ {ECHO(OFF)}
   ...Turn off echoing of the conversion factors
$ {Units("shock")}
   ...Select the shock units system
$ NOTE: Dimensions - {lout}, {mout}, {dout}, {pout}
```

* This is done so that the entire system (units conversion and material database access routines) can use a single environment variable.

```
   ...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 synonym for * (multiplication)

Material 1, Elastic Plastic, {1890~kgpm3} $ {dout}
   Youngs Modulus = {28.3e6~psi}
   Yield Stress   = {30~ksi}
   ...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 ($Revision: 1.36 $) Fri Oct 23 13:32:42 1992
   ...QA header written by Aprepro
$ 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
   Yield Stress   = 0.002068427188
   ...All material parameters are now in consistent units
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 ($Revision: 1.36 $) Fri Oct 23 13:33:22 1992
$ 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
   Yield Stress   = 206842718.8
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

6.4 Additional Comments

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

- Omitting the **{ECHO(OFF)}** line prior to 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. A few lines from the **in-lbf-s** units file are shown below:

```
$ Aprepro ($Revision: 1.36 $) Fri Oct 23 13:35:02 1992
$ This is the in-lbf-s units file: inch, sec, lbf
$ Outputs:
$   Time:             second
$   Length:           inch
$   Accel:            in/sec^2
$   Mass:             lbf-sec^2/in
$   Force:            lbf
$   Velocity:         in/sec
$   Volume:           in^3
$   Density:          lbf-sec^2/in^4
$   Energy:           inch-lbf
$   Power:            inch-lbf / sec
$   Pressure:         psi
$   Temp:             degR
$   Angular:          radian
$
$ 1 meter  = 39.37007874 inch
$ 1 second = 1 second
$ 1 kg     = 0.005710147155 lbf-sec^2/in
$ 1 kelvin = 1.8 degR
$ 1 radian = 1 radian
.....
$$$  Acceleration (L/T^2)
$ Grav. Accel.  = 386.0885827 in/sec^2
$
$$$  Force  (ML/T^2)
$ 1 Newton              = 0.2248089431 lbf
$ 1 dyne                = 2.248089431e-06 lbf
$ 1 lbf                 = 1 lbf
$ 1 kip                 = 1000 lbf
......
```

- 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 #. (See Appendix A on page 51 for a description of the command options.)

- The temperature conversions are only valid for relative temperatures, for example, 100~degC is equal to 180~degF, not 212~degF.

- Since several variables are defined in the units system, it is possible to redefine one of the variable names in your input file. If the *Aprepro* warning messages are turned off, you will not be notified of the variable redefinition and erroneous results may occur. Therefore, you should not turn off *Aprepro* warning messages while using the units system, and you should investigate all redefined variable messages to ensure that you are getting the results you expect.

7 Material Database Access System

The material database access system has been implemented in *Aprepro* to facilitate the inclusion of material property data in finite element input data files. It consists of a few functions in *Aprepro* and a specified directory structure of files that contain material property data for each material in the system and template files for each material model in each analysis code. The template files format the data from the material property files into the correct format for the analysis codes.

The material database access system is part of a larger material database system called MATS which is being developed at Sandia National Laboratories, New Mexico.

7.1 Overview of the MATS System

MATS is a series of programs and datafiles which provides the analyst with a simple method for retrieving material data from a database and inserting it into an input file for an analysis. The basic MATS system consists of the algebraic preprocessing code *Aprepro*, a set of template files for each material model in each analysis code, and a set of material database files for each material of interest. The full-featured MATS system would also include a database processor which would take raw test data and/or data from other sources, and provide the user with tools to process the data. After the user is satisfied with the fit of the data to the constitutive model that will be used, the data would be written to a datafile that could be used by the basic MATS system.

Figure 1. shows a schematic representation of the proposed MATS system. It consists of three major sections:

- *Database Preprocessor:* The database processor would be a tool which would take the raw test data and convert it into the correct format for the MATS database. Note that for nonlinear materials, this is not a simple conversion that can be performed automatically. The database processor should provide a highly interactive environment including tools such as multiple curve fitting options, filtering capabilities, options to work with portions of curves, curve editing capabilities, etc.
- *Material*: The material datafiles would be stored in this section. Each major material group (for example, steel, aluminum, foam) would be a separate subdirectory under this section. These subdirectories would contain material files for each supported material of this type. For example, the aluminum subdirectory would have a material file for 6061-T6 aluminum which would contain the material data used in structural and thermal analysis codes. For example, yield stress, density, thermal conductivity, and specific heat.
- *Code:* Code template files (which will be described later) would be stored in this section. Each supported code would have a separate subdirectory under this section. These subdirectories would contain a template file for each constitutive model supported by the analysis code. For example, the PRONTO2D subdirectory

would have templates for the Elastic, Elastic/Plastic, Johnson-Cook, Low Density Foam, and other constitutive models.

Material data would be written into the database using the database processor, a simple text editor, or a stand-alone program written specifically for that function.

Although the schematic only shows a single database structure, MATS will be written to search in several user-defined locations for the database information. This will allow user-specific, group-specific, and global databases to be developed. If an analyst develops personal datafiles for certain materials, MATS can be instructed to first look for the data in the personal datafiles. If it is found, that data will be used, if it is not found, MATS will continue to search all databases specified by the user until the data is found, or all of the specified databases have been searched.

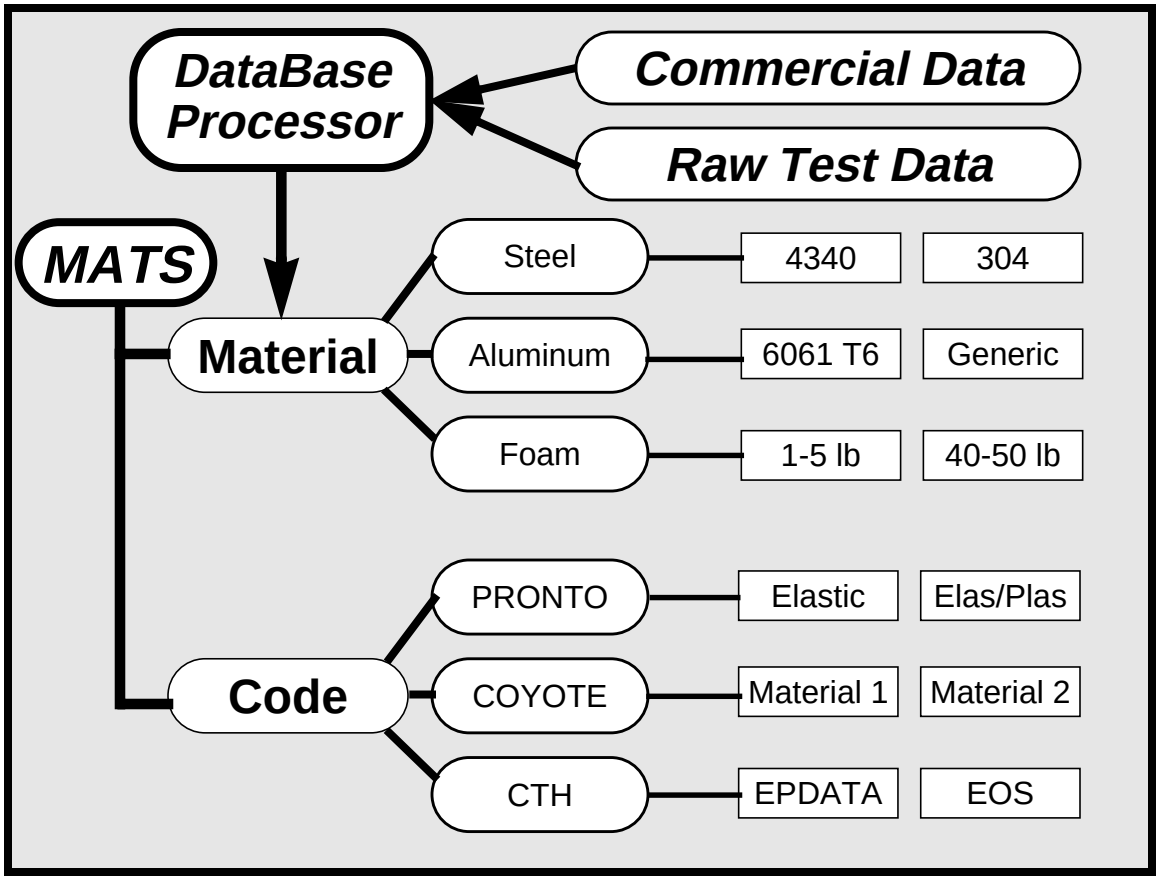

Figure 1. Schematic of Proposed MATS Database System

The remainder of this section will concentrate on the *Aprepro* interface routines to the material database system. Documentation of the overall MATS system will be published as soon as the system is developed and implemented.

7.2 Implementation of the Material Database Access Routines

The material database routines are accessed from within *Aprepro* using a command of the form:

```
{Material(mat_id, "Mat_Type", "Mat_Name", "Model_Type", "Code")}
```

This accesses the material database for the specific material **Mat_Name** which is a **Mat_Type** material and formats it in a form suitable for the **Model_Type** constitutive model in the code **Code**. For example, to use OFHC Copper with the Johnson-Cook constitutive model[11] in the PRONTO2D Code, the command would be:

```
{Material(10,"Copper","OFHC Copper","Johnson-Cook","Pronto2d")}
```

All strings are converted to lowercase so that the user only has to worry about the correct spelling.

Aprepro manipulates this line into four commands which load the correct material database file and the correct template file. The example command produces a command which has the effect of:

```
{_material_model = "johnson_cook"}
{include($MATS/material/copper/ofhc_copper)}
{include($MATS/code/pronto2d/johnson-cook)}
{material_model = " "}
```

where **$MATS** is a symbolic variable that points to the location of the material database. The symbolic variable can specify multiple locations are searched in a user-specified order to permit private databases to be searched prior to or instead of searching the default database. The mechanism for doing this is to define the **MATS** environment variable as a list of colon separated directories. For example,

```
setenv MATS ~/mats:/department/mats:/global/mats
```

In many analyses, the analyst may want to modify some of the material properties specified in the material database file. For example, only a portion of a material may be explicitly modeled and therefore, the density of the modeled portion must be increased to maintain the correct mass of the body. In this case, the following commands would be used:

```
{Material(mat_id, "Mat_Type", "Mat_Name", "Model_Type", "DEFER")}
{Density = 20000 * kg/m^3} $ User-specified density
{Material(mat_id, "DEFER", "Mat_Name", "Model_Type", "Code")}
```

This sequence of commands is manipulated into a series of commands which have the effect of first processing the material definition file, then allowing the user to modify any of the material parameters, and then formatting the data as specified in the template file for the specified code.

7.3 Code Template Files:

Each code, or "code family", has its own set of template files which extract and format the information in the material database into a code-readable format. For example, prototype

template files for use in PRONTO and SANTOS are shown below for the Elastic and Johnson Cook material models.

```
Material {_matid}, Elastic, {_Density}     {_C_} {dout}
   Youngs Modulus    = {_Youngs_Modulus}   {_C_} {pout}
   Poissons Ratio    = {_Poissons_Ratio}   {_C_} (no-dimen)
End

Material {_matid}, Johnson Cook, {_Density}  {_C_} {dout}
  Youngs Modulus     = {_Youngs_Modulus}     {_C_} {pout}
  Poissons Ratio     = {_Poissons_Ratio}     {_C_} (no-dimen)
  Yield Stress       = {_Yield_Stress}       {_C_} {pout}
  Hardening Constant = {_Hardening_Constant} {_C_} {pout}
  Hardening Exponent = {_Hardening_Exponent} {_C_} (no-dimen)
  RhoCv              = {_RhoCv}              {_C_} {pout}/{Tout}
  Rate Constant      = {_Rate_Constant}      {_C_} (no-dimen)
  Thermal Exponent   = {_Thermal_Exponent}   {_C_} (no-dimen)
  Ref Temperature    = {_Reference_Temperature}{_C_} {Tout}
  Melt Temperature         = {_Melt_Temperature}   {_C_} {Tout}
End
```

The variable names (enclosed in {}) are defined with leading underscores to reduce the redefined variable warning messages which would occur for multiple uses of the material command in a single *Aprepro* execution.

If a new, or modified, constitutive model is developed, we do not have to develop an entire new branch of the material database tree. Instead, only a new template file is created and, possibly, a few constants added to the material database. For example, if the Johnson-Cook *damage* model is implemented, the template file could look like:

```
Material {matid}, JC Damage, {Density}
   Youngs Modulus     = {Youngs_Modulus}
   Poissons Ratio     = {Poissons_Ratio}
   Yield Stress       = {Yield_Stress}
   Hardening Constant = {Hardening_Constant}
   Hardening Exponent = {Hardening_Exponent}
   RhoCV              = {RhoCv}     $ OR: {Density * Cv} ?
   Rate Constant      = {Rate_Constant}
   Thermal Exponent   = {Thermal_Exponent}
   Ref Temperature    = {Reference_Temperature}
   Melt Temperature   = {Melt_Temperature}
   D1 = {D1}, D2 = {D2}, D3 = {D3}, D4 = {D4}, D5 = {D5}
End
```

where the name in the first line of the template has been changed and the 5 constants at the end of the template have been added. These constants would then need to be added to the material files.

7.4 Material Files:

A prototype material database file is shown below. Note that the file can be divided into several sections delineating the Physical, Mechanical, and Thermal properties, for example.

```
$ {ECHO(OFF)}
Material Data File for Material -- {Material = "OFHC Copper"}
   ...NOTE: These data are for example only, DO NOT USE
------ Physical Properties
{_Density          = 8960 *kg / m^3}
------ Mechanical Properties
{_Youngs_Modulus   = E  = 124 * GPa}
{_Poissons_Ratio   = nu = 0.34}
{_Shear_Modulus    = E/2/(1+nu)}
{_Bulk_Modulus     = E/3/(1-2*nu)}
{_Yield_Stress     = 450000 * psi}
------ Thermal Properties
{_Conductivity     = k  = 386 * W / m / degK}
{_Specific_Heat    = Cp = 383 * J / kg / degK}
{_Diffusivity      = k / Density / Cp}
{_Volume_Expansion = 5.0e-5 / degK}
{_Melt_Temperature = 1356 * degK}
------- Johnson Cook Specific Properties
{_t = (_material_model=="johnson_cook" ||
         _material_model=="jc_damage")?1:0}
{ifdef(_t)}
{_Yield_Stress       = 90 * MPa}
{_Hardening_Constant = 292 * MPa}
{_Hardening_Exponent = 0.31}
   ...Several other constants
{endif}
-------- Temperature_Dependent_Material Model Specific Properties
(_t = (_material_model == "ep_temperature_dependent")?1:0}
{ifdef(_t)}
{C1 = "<<<Constant Not in Material Database>>>"
{C2 = "<<<Constant Not in Material Database>>>"
   ...The above two lines will output the message <<<Constant...Database>> to the output
      file if they are referenced
{endif}
   ...Other Models and Information
```

The material template files can have place holders for all of the information needed for the currently existing material models; if the information does not exist, the constant is set to output a warning message to the user of the information. See for example the entry for the constants **C1** and **C2** in the **ep_temperature_dependent** material block.

Many of the materials that are typically used in analyses have properties that vary depending on the temperature and/or strain rates expected in the analysis. This can be handled in a way similar to that shown in the following example which illustrates temperature-dependent material properties:

```
{NOECHO}
```

```
{range1 = (temp >   0 && temp <= 100 ) ? 1 : 0}
{range2 = (temp > 100 && temp <= 200 ) ? 1 : 0}
{range3 = (temp > 200 && temp <= 300 ) ? 1 : 0}
{range4 = (temp > 300 && temp <= 400 ) ? 1 : 0}
{ECHO}
{ifdef(range1)}
{_Linear_Expansion = 1.0e-9 / degK}
{endif}
{ifdef(range2)}
{_Linear_Expansion = 2.0e-9 / degK}
{endif}
{ifdef(range3)}
{_Linear_Expansion = 3.0e-9 / degK}
{endif}
{ifdef(range4)}
{_Linear_Expansion = 4.0e-9 / degK}
{endif}
```

In this example, the Linear Expansion Coefficient is set to a different value according to the expected temperature in the analysis*.

7.5 Additional Comments

The material database access routines are somewhat experimental at this time. They have primarily been implemented to provide an experimental testbed for implementing the MATS material database system. It is expected that the basic functionality documented in this report will remain stable; however, additional functions may be added if the need arises. The following list provides some additional information relating to the material database access routines specifically, and to the MATS system in general.

- Material property data is not and will not be distributed with *Aprepro*. It is the end users responsibility to provide this data in the form required by *Aprepro* if the database access functionality is desired. The primary reason for doing this is that the analyst should not treat this function as a black box in which appropriate material data automagically appear as the result of a simple command. Rather, it should be treated as a means of efficiently accessing (and converting to the correct units) the data that the analyst has previously collected and verified.
- A units conversion system (see section 6 on page 25) must be specified prior to accessing any data in the material database.
- The material database access routines do not verify the consistency of the material database. Procedures are needed to determine whether the data in the material databases are consistent. For example, is Poissons Ratio less than 0.5? Are the units set correctly?, etc.? This should be a separate program so that data can be entered using different programs and then checked for consistency.

*The mechanism for doing this is not very clean and will probably be changed in the future. This example is used just to show the concept.

- Since several references may be used within a single material file and similar references will be used in several material files, there should be a reference list that will cross reference an abbreviation in the material file to the full bibliographical citation for the reference. A typical reference in a material file could look like:

```
$ {Yield_Stress = 145e3*psi} $ Ref: GRJ:9
```

  which would signify that the data were found on page 9 in the document GRJ which is an abbreviation for some report reference list.

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 **ERR** 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 section A on page 51.)

Error Messages

- **Aprepro: ERR: parse error (*file*, line *line#*)** An unrecognized or ill-formed expression has been entered. Parsing of the file continues following this expression.
- **Aprepro: ERR: Can't open '*file*': No such file or directory** The file specified in the include command cannot be found or does not exist. *Aprepro* will terminate processing following this error message.
- **Aprepro: ERR: Can't open '*file*': Permission denied** The file specified in the include or output command could not be opened due to insufficient permission. *Aprepro* will terminate processing following this error message.
- **Aprepro: ERR: 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: ERR: 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: ERR: Units File not found** The units system specified in the Units command could not be found. This could be due to a misspelling of the units system name, or an incorrectly installed units system.
- **Aprepro: ERR: unit file found, no conversion file** The units system has been incorrectly installed or is not available.
- **Aprepro: ERR: Error locating material model** The specified material model datafile could not be found.
- **Aprepro: ERR: *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.

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.

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}
$ Wth  = {Wth  = 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 -= Wth}
$ {Rad2 -= Wth}
$ {Rad1 -= Wth}
   ...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.

Input:

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

Output:

```
$ Test looping - print sin, cos from 0 to 90 by 5
-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 Units and Material Database Access Example

This example illustrates the use of the units system and the material database access routines. The material data shown in this example are for illustrative purposes only and may not represent actual material data. This example also illustrates the use of the ifdef blocks to control processing of selected lines. This file was used as an input file for two analyses in which the mesh for one analysis was a subset of the other analysis. Materials 15 and 16 only appeared in the larger analysis and there were a few changes in boundary condition numbering between the two analyses. The example is annotated to explain some of the constructs used. Note that all of the dimensions in the file have unit identifiers so the unit system of the analysis can be changed simply by picking a new unit system in the Units() command.

```
{ECHO(OFF)}{Units("si")}
   ...Specify the Units system
Title
Units and Material Database Access Example
$ {InitVel = -sqrt(2.0 * ga * 500~foot)}
```

```
   …Velocity is for a 500 foot drop$
$ {Code = "Pronto3D"}
$ {ConstitModel = "JC Damage"}
   …The constitutive model used for all of the materials can now be changed simply by
     changing this line.
$
$ NOTE: dimensions - {lout}, {mout}, {dout}, {pout}
   …Echo the output units types to document processed file
$
$ {den_17 = (3.125~lbm) / (2.758e-4~metre^3)}
$ {den_18 = (1.000~lbm) / (8.747e-5~metre^3)}
   …The densities of materials 17 and 18 are modified to get the correct mass for the model
$ Control Information:
  Termination Time  {ttime = 1.0~millisecond}
  Plot Time         {ttime / 20}
  Output Time       {ttime / 200}
  Write Restart     {ttime / 10}
   …Want 20 plot steps, 200 output steps, and 10 restart steps written during the analysis
$ Boundary Conditions:
No Displacement Y  10
{ifdef(LARGE_MODEL)}
Rigid Surface 1600 {-1.85206e-1~meter} 0.0 0.0,1.0 0.0 0.0
Rigid Surface 1601 {-1.85206e-1~meter} 0.0 0.0,1.0 0.0 0.0
Rigid Surface 1602 {-1.85206e-1~meter} 0.0 0.0,1.0 0.0 0.0
{else}
Rigid Surface  400 {-1.56e-1~meter} 0.0 0.0,1.0 0.0 0.0
Rigid Surface  410 {-1.56e-1~meter} 0.0 0.0,1.0 0.0 0.0
Rigid Surface  602 {-1.56e-1~meter} 0.0 0.0,1.0 0.0 0.0
{endif}
   …Numbering of boundary conditions changes depending on which mesh is used in the
     analysis.  LARGE_MODEL is defined when full analysis is run.
  Initial Velocity Material  5  {InitVel}
  Initial Velocity Material  6  {InitVel}
  Initial Velocity Material  8  {InitVel}
  Initial Velocity Material  9  {InitVel}
  Initial Velocity Material 10  {InitVel}
{ifdef(LARGE_MODEL)}
  Initial Velocity Material 15  {InitVel}
  Initial Velocity Material 16  {InitVel}
{endif}
   …Materials 15 and 16 only appear in the large model
  Initial Velocity Material 17  {InitVel}
  Initial Velocity Material 18  {InitVel}

   …All of the material parameters are defined below.
{Material( 5, "Aluminum", "6061-T6", ConstitModel, Code)}
{Material( 6, "HE",       "PBX-9502",ConstitModel, Code)}
{Material( 8, "Aluminum", "6061-T6", ConstitModel, Code)}
{Material( 9, "Plastic",  "Lexan",   ConstitModel, Code)}
{Material(10, "Aluminum", "6061-T6", ConstitModel, Code)}
{ifdef(NOT_DEFINED)}
{Material(15, "Steel",    "13-8 H1100", ConstitModel, Code)}
{Material(16, "Aluminum", "6061-T6", ConstitModel, Code)}
```

```
{endif}
{Material(17, "Aluminum", "6061-T6", ConstitModel, "DEFER")}
{_Density = den_17}
{Material(17, "DEFER",    "6061-T6", ConstitModel, Code)}
{Material(18, "Aluminum", "6061-T6", ConstitModel, "DEFER")}
{_Density = den_18}
{Material(18, "DEFER",    "6061-T6", ConstitModel, Code)}
```

…Use all of the database properties for materials 17 and 18, execpt we need to use the calculated densities to get the correct mass in the model

Portions of the output of this example are shown below:

```
$ Aprepro ($Revision: 1.36 $) Mon Oct 26 14:15:15 1992

Title
Units and Material Database Access Example
$ -54.67235974
$ Pronto3D
$ JC Damage
$
$ NOTE: dimensions - meter, kilogram, kg/m^3, Pa
$
$ 5139.507456
$ 5185.690751
$ Control Information:
  Termination Time  0.001
  Plot Time         5e-05
  Output Time       5e-06
  Write Restart     0.0001
$ Boundary Conditions:
No Displacement Y  10
Rigid Surface  400 -0.156 0.0 0.0,1.0 0.0 0.0
Rigid Surface  410 -0.156 0.0 0.0,1.0 0.0 0.0
Rigid Surface  602 -0.156 0.0 0.0,1.0 0.0 0.0
  Initial Velocity Material  5  -54.67235974
  Initial Velocity Material  6  -54.67235974
  Initial Velocity Material  8  -54.67235974
  Initial Velocity Material  9  -54.67235974
  Initial Velocity Material 10  -54.67235974
  Initial Velocity Material 17  -54.67235974
  Initial Velocity Material 18  -54.67235974

$ 6061-T6 Aluminum
Material 5, JC Damage, 2703.78448$ kg/m^3
   Youngs Modulus= 6.894792943e+10$ Pa
   Poissons Ratio= 0.3157962771$ dimensionless
   Yield Stress= 324053592.8$ Pa
   Hardening Constant= 113763495.3$ Pa
   Hardening Exponent= 0.42$ dimensionless
   RhoCv= 2423039.586$ Pa/degK
   Rate Constant= 0.002$ dimensionless
   Thermal Exponent= 1.34$ dimensionless
   Ref Temperature= 38.88888889$ degK
   Melt Temperature= 670$ degK
```

```
   D1 = -0.77, D2 = 1.45, D3 = -0.47, D4 = 0, D5 = 1.6
$ all dimensionless
End


$ PBX 9502 (95% TATB, 5% Kel-F 800), Dobratz
Material 6, JC Damage, 1895$ kg/m^3
   Youngs Modulus= 6894757.293$ Pa
   Poissons Ratio= 0$ dimensionless
   Yield Stress= 6894757.293$ Pa
   Hardening Constant= 113763495.3$ Pa
   Hardening Exponent= 0.42$ dimensionless
   RhoCv= 2423039.586$ Pa/degK
   Rate Constant= 0.002$ dimensionless
   Thermal Exponent= 1.34$ dimensionless
   Ref Temperature= 38.88888889$ degK
   Melt Temperature= 670$ degK
   D1 = -0.77, D2 = 1.45, D3 = -0.47, D4 = 0, D5 = 1.6
$ all dimensionless
End
```

…*Rest of lines not shown*

10 References

[1]G. D. Sjaardema, “Overview of the Sandia National Laboratories Engineering Analysis Code Access System,” SAND92-2292, Sandia National Laboratories, Albuquerque, NM, January 1993.

[2]C. Donnelly and R. Stallman, “BISON--The YACC-compatible Parser Generator,” Free Software Foundation, Inc., 675 Mass Ave., Cambridge, MA, 02139, June 1992. Bison Version 1.19.

[3]V. Paxson, J. Poskanzer, and K. Gong, “FLEX--Fast Lexical Analyzer Generator,”, Free Software Foundation, Inc., 675 Mass Ave., Cambridge, MA 02139, June 1989. Flex Version 2.3.6.

[4]G. D. Sjaardema, “GREPOS: A GENESIS Database Repositioning Program,” SAND90-0566, Sandia National Laboratories, Albuquerque, NM, April 1990.

[5]G. D. Sjaardema and S. W. Attaway, “Proposed Specification for MATS,” memo to Distribution, dated January 6, 1992, Sandia National Laboratories, Albuquerque, NM.

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

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

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

[9]G. R. Simpson, “Units Computer Program”, copyright 1987.

[10]B. Berliner, “CVS II: Parallelizing Software Development,” USENIX article, Winter, 1990, Washington, D.C.

[11]G. R. Johnson and W. H. Cook, “A Constitutive Model and Data for Metals Subjected to Large Strains, High Strain Rates, and High Temperatures,” *Proceedings of Seventh International Symposium on Ballistics*, The Hague, The Netherlands, pp. 541-548, April 1983.

A Execution

Aprepro is executed with the command:

aprepro [-dsviehWMCq] [-c'*char*'] [*var=val*] [*input_file*] [*output_file*]

The effect of the parameters are:

-v prints the code name and version to the terminal. (**--version**)

-d prints the name and value of each variable defined in the input file to the terminal at the end of the run. See SYNTAX for a description of defining and using variables. (**--debug)**

-s prints statistics on hash table granularity at end of run. Primarily used for *aprepro* development. (**--statistics**)

-c '*char***'** sets the comment character that *Aprepro* writes in front of the version string and other specific output lines to the first character of '*char*' (**--comment** *'char'*)

-i puts *aprepro* into interactive mode in which there is no buffering of output. This is useful when *aprepro* is used as a pipe for another code. (**--interactive**)

-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. (**--exit_on**)

-h print a summary of the *aprepro* command line and the valid options. (**--help**)

-W do not print warning messages such as redefined variables and undefined variables. (**--nowarning**)

-M do print informational messages such as notification of included files. (**--message**)

-C print copyright message to stderr. (**--copyright)**

-q Do not print any extra information to stdout (i.e., start-up version information) (**--quiet**)

var=val sets the variable '*var*' equal to the value '*val*': 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. The command line definition of a variable does not override the definition of the same variable in the input file.

input_file specifies the file that contains the input to *Aprepro*. If this parameter is omitted, *Aprepro* will run interactively.

output_file specifies the file that *Aprepro* will write the processed data to. If this parameter is omitted, *Aprepro* will write the data to the terminal. (stdout)

The **--options** at the end of the parameter descriptions are optional long-options that can be specified instead of the short options. For example, the following two lines are equivalent:

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

Note that the short options can be concatenated.

Intentionally Left Blank

B Unit System Defined Variables

In the following list, the first column defines the unit variables that are defined in the *Aprepro* unit system and the second column is a short description of the unit. All units are defined in terms of the five SI Base Units metre (length), second (time), kilogram (mass), temperature (kelvin), and radian (angle)*. The lightly shaded 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 10: Defined Units Variables

Abbreviation	***Description***
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

* 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.

Table 10: Defined Units Variables

Abbreviation	*Description*
yr, year	Year = 365.25 days
decade	10 Years
century	100 Years
Velocity	**[L/T]**
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^3
lbmpft3	Lbm / ft^3
slugpft3	Slug / ft^3
Force	**[ML/T^2]**
N, newton	Newton = 1 kg-m/sec^2

Table 10: Defined Units Variables

Abbreviation	*Description*
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

Table 10: Defined Units Variables

Abbreviation	*Description*
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