cādence®

Analog Expression Language Reference

Product Version IC23.1
June 2023

Contents

1

Basic Concepts

Introduction to Analog Expression Language (AEL)

The Analog Expression Language (AEL) is an expression language designed to:

■ Support full or partial evaluations of expressions

■ Provide an expression evaluation mechanism especially adapted to rapid, repeated evaluation in critical time paths (that is, netlisting, simulation, and schematic annotation)

This manual contains concept and reference information about Analog Expression Language (AEL) and provides some sample applications. This user guide is aimed at developers and designers of integrated circuits and assumes that you are familiar with analog design and simulation. You should also be proficient in Cadence® SKILL language programming.

The following topics are discussed in this chapter:

■ Overview

■ Licensing Requirements

■ Partial Evaluation

■ Evaluating AEL Expressions

Overview

This section explains the concepts of the Analog Expression Language (AEL) and provides some sample applications. AEL is not a programming language, but an expression language designed to

- Support full or partial evaluations of expressions

- Provide an expression evaluation mechanism especially adapted to rapid, repeated evaluation in critical time paths (that is, netlisting, simulation, and schematic annotation)

AEL distinguishes between several types of expression components:

- Literals with suffixes or converters such as 5K = 5000

- Globals

- CDF parameter functions: *pPar( )* and *iPar( )*

- Non-CDF functions

- Literals without suffixes or converters

- Operators

AEL is used in these situations:

- In schematic annotation, you can display values or some form of expression that determines those values.

- You can dynamically define parameters for multiple simulators with a single expression and evaluate that expression *as needed* for each simulator. This is valuable because each target simulator is likely to provide a different level of support for variables and expressions.

- The large number of expressions that must be evaluated for behavioral modeling can be compiled once and executed many times. This is faster than repeatedly interpreting these expressions. Even repeated execution can be replaced by reusing previous results if expression dependency tracking is used.

In these and other situations, the controlled partial evaluation of expressions makes a single expression useful in a number of otherwise incompatible contexts. To get the same effect without controlled partial evaluation requires the maintenance of separate but related expressions, and possibly more than one expression evaluation mechanism.

Terminology

The following terms are used in this section.

Complex	In AEL, a defined data type for complex numbers. In Cadence® SKILL language, a user-defined data type for complex numbers.
Complex Strnum	A text representation of a number of the form "complex (r, i)"
Double	A SKILL f (*flonum*)
Inherited Parameters	A *parseAsCEL* parameter on the parent instance
Instance Parameters	A *parseAsCEL* parameter on the current instance
Integer	A SKILL x (*fixnum*)
List	A SKILL l (*list*) or a *typedef* by that name in the `il.h` file
Related Parameters	A term that defines instance and inherited parameters together
String	A SKILL t (*text*)
Strnum	A text representation that contains nothing but a number with optional leading and trailing white space

Globals and Constant Globals

AEL globals are values that apply to all parts of any given large application. For example, they represent values such as the sheet resistivity of a particular material or the temperature at which a given simulation is run.

Some AEL globals are built-in constants. Constant globals behave the same way as user globals, except that they exist in every AEL environment and their values cannot be changed.

Note: The names of built-in constants and mathematical functions are reserved words and cannot be used as names of user globals. Refer to *Appendix A, AEL Evaluation Symbols* for reserved names*.*

Suffixes and Converters

Suffixes and converters provide a shorthand notation for expressing powers of 10 and other multipliers. For example, *5p* reads *five pico* and represents the value *5.0e^{-12}* or

0.000000000005. (For more information on suffixes and converters, refer to <u>"AEL Evaluation Symbols"</u> on page 53.)

Defined and Deferred Globals and Functions

A user-supplied global is defined when it has a data type and a value. The value can be changed, but the data type cannot. A global is deferred if it appears in an expression but does not have a data type or value. A deferred global cannot become defined and a defined global cannot become deferred.

A defined function is one that is compiled into AEL. Both AEL-supplied and user-supplied functions are compiled into AEL. (You must have the Integrator's Toolkit™ [ITK] to do this.) The functions have a return type and, in most cases, an argument template. A function is deferred if it appears in an expression but is not compiled into AEL. A deferred function cannot become defined, and a defined function cannot become deferred.

AEL can replace a defined global with its value or a defined function call with its return value, if all arguments are also defined. If a function or global is deferred or if a defined function has any deferred arguments, AEL cannot replace the global or function with a value.

In some cases, you may want to defer a function or global so that a simulator used later in the process can define and call the function or resolve the global. For example, if you want a simulator to sweep a variable, you can defer the design global in AEL. The resulting symbolic expression can then be passed to the simulator for further evaluation.

Note: All AEL globals and functions reside in a single name space and must be unique.

Component Description Format (CDF)

Component Description Format (CDF) is a facility for dynamically defining parameters for objects at different levels of the hierarchy of a design.

AEL can affect intercomponent CDF parameter dependencies. In particular, a parameter value on a given cell can be a function of the values of other parameters on the same cell or on the parent cell. Outside of AEL, such interdependent CDF parameter relationships are supported through user-written callbacks. They are not inherent in CDF and are not supported in contexts other than form-based user input.

For a CDF parameter to be processed by AEL, it must be a text string whose content is a legal AEL expression. The string must be marked *parseAsCEL.* Expressions marked this way can refer to other *parseAsCEL* parameters on the same cell or on the parent cell. (For more information on CDF, refer to the *Component Description Format User Guide*.)

Compile and Interpret

AEL expressions can be compiled and executed in separate steps or they can be interpreted (compiled and executed in one step). In some contexts, compilation is not possible or interpretation is more appropriate. AEL is most efficiently used, however, with expressions that are compiled once and executed many times.

Because AEL supports partial evaluation, compilation is not a clearly defined task. It assumes the following conditions:

■ The values of individual globals can change between compilation and any execution.

■ The meaning or result of compilation cannot change between compilation and execution.

If the first assumption is violated, you can perform only a one-time evaluation, not a compilation. If the second assumption is violated, compilation is invalidated, requiring a recompile. Either of these situations converts the compiler into an interpreter, defeating its purpose.

An evaluation mode determines which expression components are evaluated and which are retained unaltered. The evaluation mode that is set when an environment is created cannot be changed and applies to all expressions compiled in that environment. Each expression interpreted in a given environment can specify any evaluation mode desired.

Syntax

Use the following guidelines when creating AEL expressions:

■ AEL syntax is based on the mathematical syntax in SKILL.

■ Commas are required to separate arguments to functions.

■ White space is not allowed between the last digit of a number and its suffix or converter.

■ Suffixes and converters must be followed by the end of the string, white space, an operator, or appropriate punctuation.

Licensing Requirements

For information about licensing in the Virtuoso Studio Design Environment, see *Virtuoso Software Licensing and Configuration Guide*.

Partial Evaluation

A simple expression evaluator typically yields a floating-point value. An evaluator like AEL, that supports partial evaluation and deferred symbols, cannot do so. AEL can only convert the input string into another string containing a modified representation. For example, if the input string is

```
3*2*factor
```

and `factor` is a deferred global, AEL converts this input to

```
6*factor
```

If you want to use an expression in its most symbolic form, do not request any evaluation. If you want to use an expression in its most mathematical form, request evaluation of all globals and suffixes, but do not call functions or apply operators. If you use a simulator that does not support any expressions, request full evaluation. Full evaluation requires that all globals and functions be defined and that none are deferred.

Partial evaluation allows you to request evaluation of some or all expression components. If you request full evaluation and any globals or functions are deferred, those globals and functions remain symbolic. For example,

```
1.5u*3
```

produces a string representation of a double ("$4.5e^{-6}$") if suffixes are being evaluated. It remains unchanged if they are not.

Partial Evaluation Example

The following is a sample AEL expression.

```
2 * pi * pPar("freq") / sqrt(iPar("w") * magic(c)) * 1.5u
```

This expression contains these components:

Component	Definition
2	unsuffixed literal
*	mathematical multiplication operator

Component	Definition
pi	constant global
freq	CDF parameter inherited from the current cell's parent
/	mathematical division operator
sqrt	predefined function
w	CDF parameter on the current cell instance
magic	user-supplied function
c	user-supplied global
1.5u	suffixed literal

If you request literal evaluation, the evaluator returns the expression as entered. If you request full evaluation, and neither `magic` nor `c` is deferred, the evaluator returns a string representation of a double-precision floating-point number. This number results from replacing globals with their values, replacing suffixed literals with their equivalents in scientific notation, calling functions, and applying operators.

The following string results if none of the components in the expression are deferred and you request evaluation of globals, suffixes, and CDF parameters:

```
2 * 3.14159265358979323846 * 1e9 / sqrt(3e-6 * magic(10)) * 1.5e-6
```

All predefined and user-defined globals are replaced with their values, suffixed numbers are replaced with their equivalents in scientific notation, and all CDF parameter functions are called. However, no non-CDF functions are called and no operators are applied.

Note: The values returned by `pPar( )` and `iPar( )` were arbitrarily chosen for this example.

If full evaluation of the same expression is requested and the user-supplied global `c` is deferred, the evaluator returns the following string.

```
6.283185307e9 / sqrt(3e-6 * magic(c)) * 1.5e-6
```

By deferring `c`, you also defer the following elements:

■ The function, `magic,` even though it is defined

■ The multiplication of `3e-6` by the return value of `magic`

■ The call to `sqrt,` even though it is built-in

■ The multiplication of the return value of `sqrt` by `1.5e-6`

■   The division of `6.283185307e9` by the rest of the expression

Even though you request full evaluation, only two of five operators are applied and neither of the two non-CDF functions is called. If the expression is used only for annotation or if the expression will be passed to an evaluator that can finish the job, the expression might be acceptable in this state.

Note: The cascading effect of even a single deferred value, as illustrated in this example, can have unexpected results.

Evaluating AEL Expressions

Attaching Expressions to Design Components

The documentation for CDF describes how to create a string parameter, marked *parseAsNumber* and *parseAsCEL,* and how to attach it to a given cell. When AEL expressions are used as parameters, two things happen:

■   CDF parameter values have the general resources of AEL at their disposal

■   Parameters have AEL support of instance and inherited parameters

For more information on CDF, refer to the *Component Description Format User Guide*.

Example Expressions and Evaluation

The following is an example expression using AEL syntax followed by a table listing the results of several different types of evaluation.

```
2 * pi * pPar("freq") / sqrt(iPar("w") * magic(c)) * 1.5u
```

For the purposes of this expression, the following values are assumed:

`freq` = "10M"	(on the parent of the current cell)
`w` = "5n"	(on the current cell)
`c` = 1.23	(a user-supplied AEL global)

Evaluation Method	Resulting Expression
Suffixes	`2 * pi * pPar("freq") / sqrt(iPar("w") * magic(c)) * 1.5e-6`
Globals	`2 * 3.14159 * pPar("freq") / sqrt(iPar("w") * magic(1.23)) * 1.5u`
Instance CDF parameters	`2 * pi * pPar("freq") / sqrt(5e-9 * magic(c)) * 1.5u`
Inherited CDF parameters	`2 * pi * 10e6 / sqrt(iPar("w") * magic(c)) * 1.5u`
Related parameters	`2 * pi * 10e6 / sqrt(5e-9 * magic(c)) * 1.5u`
Related parameters and suffixes	`2 * pi * 10e6 / sqrt(5e-9 * magic(c)) * 1.5e-6`
Full evaluation	`421.5e9`

2

AEL Functions

The following topics are discussed in this chapter:

AEL Utility Functions

A set of AEL utility functions (with `ael` prefixes) provides assessment and verification processes that can be used through Cadence® SKILL language programs. You can access these functions at any time within a SKILL application and are not limited to use within the AEL environment.

The following section describes the available functions for accessing AEL from the Cadence® SKILL language.

aelCheckRange

```
aelCheckRange(
     g_b1
     g_val
     g_b2
     )
     => f_violation
```

Description

Determines if a number falls within a particular range of numbers.

If the argument `val` falls within the inclusive unordered range from `b1` to `b2`, the function returns `0`. If the function returns a value less than zero, `val` violates the boundary at `b1`. If the return value is greater than zero, `val` violates the boundary at `b2`. The absolute value of the return value is equal to the absolute value of the difference between `val` and the violated boundary.
Each SKILL argument can be a *fixnum*, *flonum*, or *strnum*. If text that is not a *strnum* (for example, an expression) is passed to this function, it returns `NaN`. For this function, a *strnum* that is a complex number is an invalid argument.

The following are some examples:

If	Return value
`min(b1, b2) <= val <= max(b1, b2)`	`0`
`(b1 < b2) && (val < b1)`	`-(b1 - val)`
`(b1 < b2) && (val > b2)`	`(val - b2)`
`(b1 > b2) && (val < b2)`	`(b2 - val)`
`(b1 > b2) && (val > b1)`	`-(val - b1)`

You might want to add or subtract small values from theoretical boundaries to allow for (possibly different) tolerances, as in the last example in the following table.

Function	Result
`aelCheckRange(1, 2, 3)`	`0`
`aelCheckRange(3, 2, 1)`	`0`
`aelCheckRange(4, 1, 7)`	`-3`
`aelCheckRange(7, 1, 5)`	`4`
`aelCheckRange(low-epsilon, testVal, hi+hiTol)`	`tolerantRangeCheck`

See also   aelSignum

aelEngNotation

```
aelEngNotation(
    g_value
    x_prec
    )
    => t_strRep
```

Description

Reformats a given number into engineering notation.

This function takes a *fixnum*, a *flonum*, or a *strnum* and returns an engineering notation *strnum*. If the first argument is text that is not a *strnum* (for example, an expression), the function returns NaN. If g_value is a complex number, the return value is a string representation where the real part and the imaginary part are each in engineering notation. If the first argument is not a *fixnum*, *flonum*, or text, the function returns nil.

The x_prec argument is optional. If this argument is omitted, the function uses the currently set number of significant digits. Using this argument is the same as preceding the call to this function with the call

```
aelPushSignifDigits(x_prec)
```

and following it with

```
aelPopSignifDigits( )
```

To find out the number of significant digits in use, call the aelGetSignifDigits function.

The aelEngNotation function is called by many other functions, some of which are also part of AEL.

See also   aelGetSignifDigits

                                  aelPopSignifDigits

                                  aelPushSignifDigits

                                  aelStrDblNotation

                                  aelSuffixNotation

                                  aelSuffixWithUnits

aelGetSignifDigits

```
aelGetSignifDigits(
    )
    => x_sigDigs
```

Description

Returns the current number of significant digits used by the `aelEngNotation` function.

See also   `aelEngNotation`

`aelPopSignifDigits`

`aelPushSignifDigits`

aelNumber

```
aelNumber(
    g_value
    )
    => f_result
```

Description

Takes an integer, a double, a complex, or a *strnum* (including complex *strnums*) and returns an integer, double, a complex, or a *strnum*.

If the argument is not a *fixnum*, *flonum*, or *text*, it returns `nil`. You can use this function to test if a SKILL variable contains a number, which is one of the following:

■ An integer

■ A complex

■ A floating-point value

■ A *strnum*, including complex *strnums*

Example

```
tt = aelNumber( "123" )
123.0
floatp( tt )
t
AND
tt = aelNumber( 123 )
123
floatp( tt )
nil
```

aelPopSignifDigits

```
aelPopSignifDigits(
        )
```

Description

Discards (pops) the top value of the stack built by the `aelPushSignifDigits` function and returns a new top value.

This value is the new number of significant digits. If this function tries to discard the last value of the stack, the stack is left unchanged and the function returns `0`.

See also   `aelGetSignifDigits`

`aelPushSignifDigits`

aelPushSignifDigits

```
aelPushSignifDigits(
    x_digs
    )
    => x_digs
```

Description

Places (pushes) a given value onto the stack of values for the number of significant digits the `aelEngNotation` function uses to create engineering notation *strnums*.

Many functions call the `aelEngNotation` function indirectly and can request different numbers of significant digits.

Use the following guidelines:

■   Call the `aelPushSignifDigits` function before the `aelEngNotation` function.

■   Call the `aelPopSignifDigits` function when you are finished using the `aelEngNotation` function.

If you try to set fewer than 3 or more than 15 significant digits, the stack is left unchanged and the function returns `0`.

The SKILL versions of the `aelEngNotation`, `aelStrDblNotation`, `aelSuffixNotation`, and `aelSuffixWithUnits` function each take an optional argument, `x_prec`. This argument has the effect of preceding the main action of the called function with `aelPushSignifDigits(x_prec)` and following it with `aelPopSignifDigits`.

See also	
	`aelEngNotation`
	`aelPopSignifDigits`
	`aelStrDblNotation`
	`aelSuffixNotation`
	`aelSuffixWithUnits`

Related Environment Variable

The `useSignifDigitsForSimpleNotation` variable can be used to control the number of significant digits when the notation is set to simple.

`auCore.misc useSignifDigitsForSimpleNotation boolean nil`

When this variable is set to `true` and `gLabelsNumNotation` is set to `simple`, the label uses number of significant digits specified in the `aelPushSignifDigits()` function. By default, the `useSignifDigitsForSimpleNotation` variable is set to `nil`.

Example

The table below shows the outputs generated for different values when the `useSignifDigitsForSimpleNotation` variable is set to `t` and the number of significant digits specified in the `aelPushSignifDigits()` function is `5`.

Values	Outputs
12345678	12346000
"12345678"	1.2346e+07
12.345678	12.346
123456.78	123460.0
12345.0	12345
"12345"	12345.0
1234567890987654321	1.2346e+18
1.210000585038126e-12	1.21e-12

aelSignum

```
aelSignum(
    g_val
    )
    => -1 / 0 / 1
```

Description

Analyzes an integer, a double, or a *strnum* (not including complex *strnums*) and returns -1 if the value is negative, 0 if it is zero, or 1 if it is positive.

If text that is not a *strnum* (for example, an expression) is passed to this function, it returns NaN. If the function receives an argument that is not a *fixnum*, *flonum*, or text, it returns nil.

aelStrDblNotation

```
aelStrDblNotation(
    g_value
    x_prec
    )
    => t_strRep
```

Description

Calls the `aelEngNotation` function and, if the result is an integer, appends a decimal point and zero (`.0`).

This satisfies those situations that require a decimal point to identify a number as a double rather than an integer.

See also	`aelEngNotation`

aelSuffixNotation

```
aelSuffixNotation(
    g_value
    x_prec
    )
    => t_strRep
```

Description

Calls the `aelEngNotation` function and replaces the *e* and exponent (if any) with the appropriate AEL suffix character. If the value is too small or too large for the AEL set of suffixes, the *e* and exponent are unchanged.

See also	`aelEngNotation`

aelSuffixWithUnits

```
aelSuffixWithUnits(
    g_value
    t_units
    x_prec
    )
    => t_strRep
```

Description

Used to display values on forms.

This function is equivalent to

```
strcat (strcat(aelSuffixNotation(val) " ") units)
```

This function works like the `aelSuffixNotation` function with the following exceptions:

- Complex numbers are not supported.

- A space and the given units are appended to the end of the returned text.

AEL Environment Functions

The AEL environment functions (with `aelEnv` prefixes) for SKILL provide access to the expression evaluation capabilities of AEL. You can use these functions from a SKILL program to perform full or partial evaluation of an expression. See the following chapters for the AEL environment functions for SKILL.

The expression evaluation routine consists of the following steps:

1. Use `aelEnvCreate` to create the processing environment to allocate a distinct name space and to specify full or partial evaluation.

2. Define variables within this environment and create the expressions to be evaluated.

3. Compile and execute the expressions with `aelEnvCompile` and `aelEnvExecute` or interpret the expressions with `aelEnvInterpret`.

4. Use the fully or partially evaluated expressions.

5. Deallocate the AEL environment that you created.

The following code accesses AEL environment functions with SKILL. The sample creates an AEL environment, sets global variables within the environment, and interprets two mathematical expressions with the variables.

```
env = aelEnvCreate( 'f )
aelEnvSetGlobals( env "a" 10 "b" 20 )
aelEnvInterpret( env "a+b" )
==> "30.0"
aelEnvInterpret( env "sqrt(a+b)" )
==> "5.4885"
```

The following section describes the available functions for accessing the AEL environment from SKILL.

aelEnvCompile

```
aelEnvCompile(
     g_env
     t_expr
     )
     => x_compExpr
```

Description

Compiles a given expression according to the evaluation mode set with `aelEnvCreate` and the state of globals for the given environment.
Compiled expressions are references to dynamically allocated memory. There is no facility for extending their lifetime or visibility beyond the process in which they are created. Do not attempt to store them in a design database. Compiled expressions are invalidated and yield undefined results if you apply the `aelEnvFreeCompExpr` function to them, or if you destroy their associated environment with the `env = nil` expression.

Each global appearing in an expression being compiled is created as a deferred global in the given environment if it does not already exist there. Deferred globals have neither type nor value. AEL assumes they are of the correct type.

Call the `aelEnvListDeferredGlobals` and `aelEnvListDeferredFuncs` functions after compiling expressions. This lets you see if a typographical error created a deferred global or function when you intended to reference a defined one.

Text substitution in an AEL expression means replacing an AEL global with its text value before compiling the expression. This is similar to a parameterless macro capability. For example, if a is an AEL global with the text value of `pi/2`, text substitution replaces the global with its text value as shown in the following example.

```
3*a
3*(pi/2)
```

Note: The text substitution parenthesizes replacement text unconditionally. Text substitution of globals recurs until no globals with text values remain. (Text substitution is discussed in more detail under `aelEnvCreate`.)

Note: These functions return all deferred globals and functions in the given environment, not just those encountered in the most recent compilation.

The first detected error terminates compilation and an error code is set.

Usually, using the compile and execute model produces better performance than the interpreter model.

See also   `aelEnvCreate`

`aelEnvExecute`

`aelEnvInterpret`

`aelEnvListDeferredGlobals`

aelEnvCreate

```
aelEnvCreate(
    t_evalMode
    g_distinguishIntsFromDbls
    g_keepWhitespace
    )
    => x_env
```

Description

Creates and initializes an AEL environment.

The value returned is used to identify this environment in subsequent calls to various other functions with names beginning with `aelEnv`.

If the `distinguishIntsFromDbls` option is `nil`, all numeric values other than complex numbers are represented as double precision floating-point numerics. There are no integers.

If the `keepWhitespace` option is `nil` or omitted, all white space is removed. Otherwise, minimum white space is used regardless of the amount of white space found on input.

Note: The evaluation mode is case sensitive.

Evaluation Mode	Definition
c	All function calls (implies evaluation mode *ip*)
C	All function calls except the CDF parameter calls
f	Full evaluation, except for deferred globals and functions and their side effects (This does not override evaluation mode *C*, allowing for all but CDF parameters.)
g	Globals, except those deferred
i	Instance CDF parameters on the same cell (the *iPar( )* function)
l	Literal: no evaluation
o	All operators
p	CDF parameters inherited from the parent cell (the *pPar( )* function)
r	All related CDF parameters (same as evaluation mode *ip*)
s	Suffixes
t	Text substitution before compiling

The evaluation mode can be one mode or any combination of modes, except *l* or *f*. (The evaluation modes *l* and *f* are mutually exclusive.) Explicit and implicit redundancy is benign. For example, *isri* is a legal evaluation mode in which *i* occurs twice explicitly and is also implicit in *r*.

Note: Remember that *i* stands for instance, not inherited. Use *p* for parent when you want inherited parameters to be evaluated.

Text substitution in an AEL expression means replacing an AEL global with its text value before compiling the expression. This is similar to a parameterless macro capability. For example, if `a` is an AEL global with the text value of `pi/2`, text substitution replaces the global with its text value as shown in the following example.

```
3*a
3*(pi/2)
```

Note: The text substitution parenthesizes replacement text unconditionally. Text substitution of globals recurs until no globals with text values remain.

Text substitution is a mode you can select, like suffixes, globals, and so forth. The flag character is *t*. For example, to select text substitution and globals, you use *gt* or *tg* as the evaluation mode selection. As with other selections, it is illegal to use the evaluation mode *t* with *l* (literal), and *l* is implied in the evaluation mode *f* (full). The evaluation mode *t* is meaningless if the mode does not call for evaluation of globals, or if you have no globals with text values.

Note: Use the `env = nil` expression, where *env* is the *x_env* created by the `aelEnvCreate` function, to destroy and remove an AEL environment.

aelEnvExecute

```
aelEnvExecute(
    x_compExpr
    )
    => t_result
```

Description

Executes the previously compiled AEL expression with the current global values of the environment in which it was compiled.

The evaluation mode set with `aelEnvCreate` is in effect. The values of globals in that environment can be changed by the `aelEnvSetGlobals` function any number of times between a single compile and any number of executions.

It is invalid to call this function once you apply the `aelEnvFreeCompExpr` function to the compiled expression or destroy the associated environment by the following expression:

```
env = nil
```

If there is an error, the return value is `nil`. Otherwise, it is a string.

See also	`aelEnvCompile`
	`aelEnvInterpret`
	`aelEnvSetGlobals`
	`aelNumber`

aelEnvFreeCompExpr

```
aelEnvFreeCompExpr(
    x_compExpr
    )
    => t
```

Description

Frees the dynamically allocated memory used to store the compiled AEL expression.

It is invalid to refer to a compiled expression after you apply this function to it.

aelEnvGetErrStr

```
aelEnvGetErrStr(
      )
    => t_errMsg
```

Description

Converts the error code currently stored in the all-AEL-wide error flag into an appropriate message.

This call resets the flag to `no error`. If no error has been detected, the function returns the string `no error`.

aelEnvGetGlobal

```
aelEnvGetGlobal(
    x_env
    t_name
    )
    => g_value
```

Description

Returns the current value of the named global, previously defined or deferred, in the given environment.

This function does not work if you have used the following expression to destroy the given environment:

```
env = nil
```

The second argument can be text or a symbol. The return value is of the appropriate type.

If the global is neither defined nor deferred in the given environment, the function returns `nil`. If the global is deferred, the return value is the text string *deferred*.

aelEnvInterpret

```
aelEnvInterpret(
    x_env
    t_expr
    t_mode
    )
    => g_value
```

Description

Compiles and executes the specified expression in a single step.

This function is similar to calling the `aelEnvCompile` function, followed by the `aelEnvExecute` function, followed by the `aelEnvFreeCompExpr` function. The `aelEnvCompile` function uses the evaluation mode set when the environment was created. The `aelEnvInterpret` function uses whatever evaluation mode you specify in the `t_mode`. You can default to the evaluation mode set during the creation of the environment by passing `nil` for the evaluation mode in this call, or by omitting this argument.

Usually, using the compile and execute model produces better performance than the interpreter model.

See also	aelEnvCompile
	aelEnvCreate
	aelEnvExecute
	aelEnvFreeCompExpr

aelEnvListDeferredFuncs

```
aelEnvListDeferredFuncs(
    x_env
    )
=> l_names
```

Description

Returns a list of all functions found in the compiled or interpreted expressions in the given environment that are undefined in AEL.

See also	`aelEnvListExprFuncs`
	`aelEnvListFuncs`

aelEnvListDeferredGlobals

```
aelEnvListDeferredGlobals(
    x_env
    )
    => l_names
```

Description

Returns a list of all globals found in the compiled or interpreted expressions in the given environment that have not been defined in a call to the `aelEnvSetGlobals` function.

See also	`aelEnvListExprGlobals`
	`aelEnvListGlobals`

aelEnvListExprFuncs

```
aelEnvListExprFuncs(
    t_expr
    )
    => l_funcs
```

Description

Returns a list of all the functions found in the given expression.

aelEnvListExprGlobals

```
aelEnvListExprGlobals(
    t_expr
    )
    => l_globals
```

Description

Returns a list of all the globals found in a given expression.

aelEnvListFuncs

```
aelEnvListFuncs(
    )
    => l_names
```

Description

Returns a list of all the functions defined for all AEL environments.

See also	`aelEnvListDeferredFuncs`
	`aelEnvListExprFuncs`

aelEnvListGlobals

```
aelEnvListGlobals(
    x_env
    )
    => l_names
```

Description

Returns a list of all the globals in a given AEL environment. The list does not include AEL predefined constant globals or deferred globals.

See also	aelEnvListDeferredGlobals

aelEnvListGlobalsValues

```
aelEnvListGlobalsValues(
    x_env
    )
=> l_names
```

Description

Returns a list of all global names and their values in a given AEL environment. The list does not include AEL predefined constant globals or deferred globals.

aelEnvName

```
aelEnvName(
     t_name
     )
     => t/nil
```

Description

Accepts text or a symbol and returns a legal AEL name stripped of any leading or trailing white space that might have been in the argument.

The input is subject to lexical validation only. If the name is not a legal AEL name, `nil` is returned. No test is performed to determine if this global is defined or deferred in any AEL environment.

Legal global or function names match the regular expression

```
[a-zA-Z][a-zA-Z_$0-9]*
```

aelEnvSetGlobals

```
aelEnvSetGlobals(
     x_env
     t_name
     g_val
     )
     => x_count
```

Description

Sets the values of the named globals in a given environment, creating them if necessary.

Legal global names match the regular expression

```
[a-zA-Z][a-zA-Z_$0-9]*
```

The second argument can be text or a symbol. The third argument can be an integer, a double, or a text string. If it is a symbol, it is treated as a text string. If it is a text string or a symbol, it does not have to be a *strnum* but can be any text. The second and third arguments can be followed by any number of name-value pairs.

Note: In the 4.3 version or in later versions, the user <u>do not</u> use the `aelEnvSetGlobals` function to set lineage. Instead, use the `aelSetLineage` function.

It is an error if any value passed is of an unsupported type, or if a global with the given name already exists in the given environment and is of a different type. It is also an error to call this function once you use the following expression to destroy the given environment:

```
env = nil
```

where

```
env = aelEnvCreate(...)
```

You cannot set a global once it is deferred. You cannot defer a global once it has been set.

You can apply the `aelEnvSetGlobals` function any number of times before or after you apply `aelEnvCompile`, `aelEnvExecute`, or `aelEnvInterpret` expressions.

If the return value is `0`, it means an error prevented definition of any globals. If it is negative, it means an error occurred with some globals defined, and others not. The absolute value of the return value is the first global that did not get defined. No subsequent globals are tried. For example, if you try to define three globals and the return value is -2, the first global was set successfully, the second caused an error, and the third was not attempted.

A positive return value is the number of globals set. It should equal the number of globals submitted.

 See also

```
aelEnvCompile

aelEnvExecute

aelEnvGetErr

aelEnvGetErrStr

aelEnvInterpret

aelEnvSetGlobalList

aelSetLineage
```

aelEnvSetGlobalList

```
aelEnvSetGlobalList(
    x_env
    l_dpl
    )
    => x_count
```

Description

Using a disembodied property list as input sets the values of the named globals in a given environment.

This function is similar to `aelEnvSetGlobals`. If the name-value pairs are already collected into a disembodied property list, use `aelEnvSetGlobalList` to pass the list as the second (and last) argument.

See also	`aelEnvSetGlobals`

aelSetLineage

```
aelSetLineage(
     x_env
     g_lineage
     )
     => o_lineage
```

Description

The `aelSetLineage` function sets reference to a lineage for hierarchical expression evaluation. The lineage can either be a list or an `aelLineage` (SKILL user-defined) object. In either case, an `aelLineage` object is returned. The list is an ordered list of cell views whose *car* is the current instance, whose *cadr* is its parent, and so on. The last element of this list should be the top cell view.

Note: The list must be the instantiations of the cells from within the circuit being used for any hierarchical expression evaluations to work correctly. It cannot just be a list of the individual cell view schematics that make up the hierarchy.

Example

```
mxEnv = aelEnvCreate("f" t t)
top_cv=dbOpenCellView("sample" "mytop" "schematic")
mid_cv=dbOpenCellView("sample" "mymid" "schematic")
bot_cv=dbOpenCellView("sample" "mybot" "schematic")
foreach(inst top_cv~>instances
     if(inst~>cellName == "mymid" then
          mid_inst = inst
     )
)
foreach(inst mid_cv~>instances
     if(inst~>cellName == "mybot" then
          bot_inst = inst
     )
)
midDBList = list(mid_inst top_cv)
botDBList = list(bot_inst mid_inst top_cv)
foreach(inst mid_cv~>instances
     if(inst~>cellName == "res" then
          if(car(inst~>prop~>name) == "r" &&
               index(car(inst~>prop~>value) "pPar") then
               tmpList = cons(inst midDBList)
               aelSetLineage(mxEnv tmpList)
               value = aelEnvInterpret(mxEnv car(inst~>prop~>value))
               printf("Mid: %L=%L -- %L\n" inst~>name value inst~>prop~>value)
          )
     )
)
foreach(inst bot_cv~>instances
     if(inst~>cellName == "res" then
          if(car(inst~>prop~>name) == "r" &&
               index(car(inst~>prop~>value) "pPar") then
```

```
        tmpList = cons(inst botDBList)
        aelSetLineage(mxEnv tmpList)
        value = aelEnvInterpret(mxEnv car(inst~>prop~>value))
        printf("Bot: %L=%L -- %L\n" inst~>name value inst~>prop~>value) ;
    )

  )
)
```

A

AEL Evaluation Symbols

The following tables contain information on suffixes, converters, constants, mathematical functions, and operators used in AEL expressions. This information is included as a quick reference to standards used in AEL.

Suffixes

Character	Name	Multiplier	Examples
Y	Yotta	10^{24}	10Y [10e+24]
Z	Zetta	10^{21}	10Z [10e+21]
E	Exa	10^{18}	10E [10e+18]
P	Peta	10^{15}	10P [10e+15]
T	Tera	10^{12}	10T [10e+12]
G	Giga	10^{9}	10G [10,000,000,000]
M	Mega	10^{6}	10M [10,000,000]
k or K	kilo	10^{3}	10k [10,000]
%	percent	10^{-2}	5% [0.05]
m	milli	10^{-3}	5m [5.0e-3]
u	micro	10^{-6}	1.2u [1.2e-6]
n	nano	10^{-9}	1.2n [1.2e-9]
p	pico	10^{-12}	1.2p [1.2e-12]
f	femto	10^{-15}	1.2f [1.2e-15]
a	atto	10^{-18}	1.2a [1.2e-18]
z	zepto	10^{-21}	1.2z [1.2e-21]
y	yocto	10^{-24}	1.2y [1.2e-24]

Suffix	Meaning
T	Tera ($1e^{12}$)
G	Giga ($1e^{9}$)
M	Mega ($1e^{6}$)
K, k	Kilo ($1e^{3}$)
%	percent ($1e^{-2}$)
m	milli ($1e^{-3}$)

Suffix	Meaning
u	micro ($1e^{-6}$)
n	nano ($1e^{-9}$)
p	pico ($1e^{-12}$)
f	femto ($1e^{-15}$)
a	atto ($1e^{-18}$)

Converters

Converter	Converts from	Converts to
mil	thousandths of an inch	meters
dBW	decibel watts	watts
dBm	decibel milliwatts	watts
deg	degrees	radians
degC	Celsius	Kelvin
degF	Fahrenheit	Kelvin

Constants

Constant	Definition
boltzmann	$1.380622e^{-23}$
charge	$1.6021917e^{-19}$
degPerRad	57.2957795130823208772
epp0	$8.854e^{-12}$
pi	3.14159265358979323846
sqrt2	1.41421356237309504880
twoPi	6.28318530717958647688

Mathematical Functions

Function	Description
acos(x)	returns arc cosine of x
acosh(x)	returns hyperbolic arc cosine of x
asin(x)	returns arc sine of x
asinh(x)	returns hyperbolic arc sine of x
atan(x)	returns arc tangent of x
atan2(x, y)	returns arctan (y/x) (handles zero case)

Function	Description
atanh(x)	returns hyperbolic arc tangent of x
ceil(x)	Returns the integer of x raised to the closest integer value
complex(x1,x2)	Returns a complex number with the i and j of x1 and x2 respectively
conjugate(x)	Returns the complex conjugate of x
cos(x)	returns cosine of x
cosh(x)	returns hyperbolic cosine of x
exp(x)	returns exponent of x
fabs(x)	returns floating-point absolute value of x
floor(x)	Returns the integer of x lowered to the closest integer value
hypot(x,y)	returns hypotenus for sides x,y
ln(x)	returns natural log of x
log(x)	returns natural log of x
log10(x)	returns log of x
pow(x,y)	returns x raised to power y
sin(x)	returns sine of x
sinh(x)	returns hyperbolic sine of x
sqrt(x)	returns square root of x
tan(x)	returns tangent of x
tanh(x)	returns hyperbolic tangent of x

Mathematical Operators

Operator	Definition
–	unary minus
!	unary not
~	unary 1's complement
**	exponentiation
*	multiply
/	divide
+	plus
–	binary minus
<<	left shift
>>	right shift
<	less than
<=	less than or equal to
>=	greater than or equal to
>	greater than
==	equal to (equality, not assignment)
!=	not equal to (equality, not assignment)
&	bitwise AND
^	bitwise exclusive OR
|	bitwise inclusive OR
&&	logical AND
||	logical OR
?:	conditional expression (as in C)