

# **Analog Expression Language Reference**

**Product Version ICADVM20.1  
October 2020**

© 2020 Cadence Design Systems, Inc. All rights reserved.  
Printed in the United States of America.

Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

Open SystemC, Open SystemC Initiative, OSCI, SystemC, and SystemC Initiative are trademarks or registered trademarks of Open SystemC Initiative, Inc. in the United States and other countries and are used with permission.

**Trademarks:** Trademarks and service marks of Cadence Design Systems, Inc. contained in this document are attributed to Cadence with the appropriate symbol. For queries regarding Cadence's trademarks, contact the corporate legal department at the address shown above or call 800.862.4522. All other trademarks are the property of their respective holders.

**Restricted Permission:** This publication is protected by copyright law and international treaties and contains trade secrets and proprietary information owned by Cadence. Unauthorized reproduction or distribution of this publication, or any portion of it, may result in civil and criminal penalties. Except as specified in this permission statement, this publication may not be copied, reproduced, modified, published, uploaded, posted, transmitted, or distributed in any way, without prior written permission from Cadence. Unless otherwise agreed to by Cadence in writing, this statement grants Cadence customers permission to print one (1) hard copy of this publication subject to the following conditions:

1. The publication may be used only in accordance with a written agreement between Cadence and its customer.
2. The publication may not be modified in any way.
3. Any authorized copy of the publication or portion thereof must include all original copyright, trademark, and other proprietary notices and this permission statement.
4. The information contained in this document cannot be used in the development of like products or software, whether for internal or external use, and shall not be used for the benefit of any other party, whether or not for consideration.

**Patents:** The Cadence Products covered in this manual are protected by U.S. Patents

5,790,436; 5,812,431; 5,859,785; 5,949,992; 6,493,849; 6,278,964; 6,300,765; 6,304,097; 6,414,498; 6,560,755; 6,618,837; 6,693,439; 6,826,736; 6,851,097; 6,711,725; 6,832,358; 6,874,133; 6,918,102; 6,954,908; 6,957,400; 7,003,745; 7,003,749.

**Disclaimer:** Information in this publication is subject to change without notice and does not represent a commitment on the part of Cadence. Except as may be explicitly set forth in such agreement, Cadence does not make, and expressly disclaims, any representations or warranties as to the completeness, accuracy or usefulness of the information contained in this document. Cadence does not warrant that use of such information will not infringe any third party rights, nor does Cadence assume any liability for damages or costs of any kind that may result from use of such information.

**Restricted Rights:** Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR52.227-14 and DFAR252.227-7013 et seq. or its successor

---

# Contents

---

<u>Preface</u>	7
<u>Scope</u>	7
<u>Licensing Requirements</u>	8
<u>Related Documentation</u>	8
<u>Installation, Environment, and Infrastructure</u>	8
<u>Virtuoso Tools</u>	8
<u>Additional Learning Resources</u>	8
<u>Video Library</u>	8
<u>Virtuoso Videos Book</u>	9
<u>Rapid Adoption Kits</u>	9
<u>Help and Support Facilities</u>	9
<u>Customer Support</u>	10
<u>Feedback about Documentation</u>	10
<u>Understanding Cadence SKILL</u>	11
<u>Using SKILL Code Examples</u>	11
<u>Sample SKILL Code</u>	11
<u>Accessing API Help</u>	12
<u>Typographic and Syntax Conventions</u>	13
<u>Identifiers Used to Denote Data Types</u>	14
<u>1</u>	
<u>Basic Concepts</u>	17
<u>Overview</u>	18
<u>Terminology</u>	19
<u>Globals and Constant Globals</u>	19
<u>Suffixes and Converters</u>	19
<u>Defined and Deferred Globals and Functions</u>	20
<u>Component Description Format (CDF)</u>	20
<u>Compile and Interpret</u>	21
<u>Syntax</u>	21
<u>Partial Evaluation</u>	21

# Analog Expression Language Reference

---

<u>Partial Evaluation Example</u>	22
<u>Evaluating AEL Expressions</u>	24
<u>Attaching Expressions to Design Components</u>	24
<u>Example Expressions and Evaluation</u>	24

## 2

### AEL Functions 27

<u>AEL Utility Functions</u>	28
<u>aelCheckRange</u>	29
<u>aelEngNotation</u>	31
<u>aelGetSignifDigits</u>	32
<u>aelNumber</u>	33
<u>aelPopSignifDigits</u>	34
<u>aelPushSignifDigits</u>	35
<u>aelSignum</u>	37
<u>aelStrDbfNotation</u>	38
<u>aelSuffixNotation</u>	39
<u>aelSuffixWithUnits</u>	40
<u>AEL Environment Functions</u>	41
<u>aelEnvCompile</u>	42
<u>aelEnvCreate</u>	44
<u>aelEnvExecute</u>	46
<u>aelEnvFreeCompExpr</u>	47
<u>aelEnvGetErrStr</u>	48
<u>aelEnvGetGlobal</u>	49
<u>aelEnvInterpret</u>	50
<u>aelEnvListDeferredFuncs</u>	51
<u>aelEnvListDeferredGlobals</u>	52
<u>aelEnvListExprFuncs</u>	53
<u>aelEnvListExprGlobals</u>	54
<u>aelEnvListFuncs</u>	55
<u>aelEnvListGlobals</u>	56
<u>aelEnvListGlobalsValues</u>	57
<u>aelEnvName</u>	58
<u>aelEnvSetGlobals</u>	59

## Analog Expression Language Reference

---

<u>aelEnvSetGlobalList</u> .....	61
<u>aelSetLineage</u> .....	62

### A

<u>AEL Evaluation Symbols</u> .....	65
<u>Suffixes</u> .....	66
<u>Converters</u> .....	68
<u>Constants</u> .....	68
<u>Mathematical Functions</u> .....	68
<u>Mathematical Operators</u> .....	70

<u>Index</u> .....	71
--------------------	----

## Analog Expression Language Reference

---

---

# Preface

---

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.

This preface contains the following topics:

- [Scope](#)
- [Licensing Requirements](#)
- [Related Documentation](#)
- [Additional Learning Resources](#)
- [Customer Support](#)
- [Feedback about Documentation](#)
- [Understanding Cadence SKILL](#)
- [Typographic and Syntax Conventions](#)
- [Identifiers Used to Denote Data Types](#)

## Scope

Unless otherwise noted, the functionality described in this guide can be used in both mature node (for example, IC6.1.8) and advanced node and methodologies (for example, ICADVM20.1) releases.

Label	Meaning
-------	---------

# Analog Expression Language Reference

## Preface

---

(ICADVM20.1 Only)	Features supported only in the ICADVM20.1 advanced nodes and advanced methodologies release.
(IC6.1.8 Only)	Features supported only in mature node releases.

## Licensing Requirements

For information about licensing in the Virtuoso design environment, see [\*Virtuoso Software Licensing and Configuration Guide\*](#).

## Related Documentation

### Installation, Environment, and Infrastructure

- [\*Cadence Installation Guide\*](#)
- [\*Cadence SKILL Language User Guide\*](#)

### Virtuoso Tools

- [\*Virtuoso Schematic Editor L User Guide\*](#).
- [\*Virtuoso Analog Design Environment L User Guide\*](#)
- [\*Virtuoso Analog Design Environment XL User Guide\*](#)
- [\*Virtuoso Analog Design Environment GXL User Guide\*](#)

## Additional Learning Resources

### Video Library

The [\*Video Library\*](#) on the Cadence Online Support website provides a comprehensive list of videos on various Cadence products.



To view a list of videos related to a specific product, you can use the *Filter Results* feature available in the pane on the left. For example, click the *Virtuoso Layout Suite* product link to view a list of videos available for the product.

You can also save your product preferences in the Product Selection form, which opens when you click the *Edit* icon located next to *My Products*.

## Virtuoso Videos Book

You can access certain videos directly from Cadence Help. To learn more about this feature and to access the list of available videos, see [Virtuoso Videos](#).

## Rapid Adoption Kits

Cadence provides a number of [Rapid Adoption Kits](#) that demonstrate how to use Virtuoso applications in your design flows. These kits contain design databases and instructions on how to run the design flow.

To explore the full range of training courses provided by Cadence in your region, visit [Cadence Training](#) or write to [training\\_enroll@cadence.com](mailto:training_enroll@cadence.com).

**Note:** The links in this section open in a separate web browser window when clicked in Cadence Help.

## Help and Support Facilities

Virtuoso offers several built-in features to let you access help and support directly from the software.

- The Virtuoso *Help* menu provides consistent help system access across Virtuoso tools and applications. The standard Virtuoso *Help* menu lets you access the most useful help and support resources from the Cadence support and corporate websites directly from the CIW or any Virtuoso application.
- The Virtuoso Welcome Page is a self-help launch pad offering access to a host of useful knowledge resources, including quick links to content available within the Virtuoso installation as well as to other popular online content.

The Welcome Page is displayed by default when you open Cadence Help in standalone mode from a Virtuoso installation. You can also access it at any time by selecting *Help – Virtuoso Documentation Library* from any application window, or by clicking the *Home* button on the Cadence Help toolbar (provided you have not set a custom home page).

For more information, see [Getting Help](#) in *Virtuoso Design Environment User Guide*.

## Customer Support

For assistance with Cadence products:

- **Contact Cadence Customer Support**

Cadence is committed to keeping your design teams productive by providing answers to technical questions and to any queries about the latest software updates and training needs. For more information, visit <https://www.cadence.com/support>.

- **Log on to Cadence Online Support**

Customers with a maintenance contract with Cadence can obtain the latest information about various tools at <https://support.cadence.com>.

## Feedback about Documentation

You can contact Cadence Customer Support to open a service request if you:

- Find erroneous information in a product manual
- Cannot find in a product manual the information you are looking for
- Face an issue while accessing documentation by using Cadence Help

You can also submit feedback by using the following methods:

- In the Cadence Help window, click the *Feedback* button and follow instructions.
- On the Cadence Online Support [Product Manuals](#) page, select the required product and submit your feedback by using the *Provide Feedback* box.

## Understanding Cadence SKILL

Cadence SKILL is a high-level, interactive programming language based on the popular artificial intelligence language, Lisp. It lets you customize and extend your design environment. Using SKILL you can validate the steps of your algorithm incrementally before incorporating them into a larger program.

For more information about the SKILL language, see [Getting Started](#) in the *SKILL Language User Guide*.

## Using SKILL Code Examples

The SKILL APIs in this user manual are explained with illustrative code examples.

You can copy these examples from the manual and paste them directly into the Command Interpreter Window (CIW) or use the code in non-graphical SKILL mode.

## Sample SKILL Code

The following code sample shows the syntax of a SKILL API that accepts three arguments.

### axlGetRunStatus

```
axlGetRunStatus(
    t_sessionName      ← Required argument
    [ ?optionName t_optionName ] ← Optional keyword argument
    [ ?historyName t_historyName ] ← Optional keyword argument
)
=> l_statusValues      ← Return value
```

The first argument `t_sessionName` is a required argument, where `t` signifies the data type of the argument. The second and third arguments `?optionName t_optionName` and `?historyName t_historyName` are optional keyword arguments (identified by a question mark), which are specified in name-value pairs and can be placed in any order during the function call.

## Analog Expression Language Reference

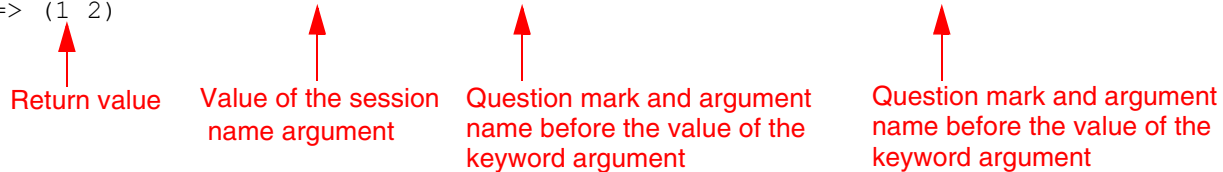
### Preface

---

The return value is the value that the SKILL API returns after evaluating the expression. In this case, it is a list of status values, *l\_statusValues*.

#### Example

```
axlSession=axlGetWindowSession( hiGetCurrentWindow() )
=> "session0"
axlGetRunStatus("session0" ?historyName "Interactive.10" ?optionName "tests")
=> (1 2)
```



Return value

Value of the session name argument

Question mark and argument name before the value of the keyword argument

Question mark and argument name before the value of the keyword argument

## Accessing API Help

Quick reference information for SKILL APIs is available from the CIW and the SKILL API Finder. To access the reference information for a particular SKILL API, do one of the following:

- Type `help <function_name>` in the CIW.
- Type `startFinder ( [ ?funcName t_functionName ] )` in the CIW.
- Start the **SKILL API Finder** from the CIW by choosing *Tools – Finder* or type `cdsFinder` on the UNIX command line.

In the *Search in* field of the displayed Cadence SKILL API Finder window, type the SKILL API name for which you want to display the help information and click *Go*.

The matches for the searched SKILL API appear in the *Results* area.

To view the complete documentation of the searched SKILL API, select the API name in the *Results* area and click the *More Info* button. The complete documentation of the selected SKILL API appears in a new Cadence Help window.

## Typographic and Syntax Conventions

The following typographic and syntax conventions are used in this manual.

<i>text</i>	Indicates names of manuals, menu commands, buttons, and fields.
text	Indicates text that you must type as presented. Typically used to denote command, function, routine, or argument names that must be typed literally.
<i>z_argument</i>	Indicates text that you must replace with an appropriate argument value. The prefix (in this example, <i>z_</i> ) indicates the data type the argument can accept and must not be typed.
	Separates a choice of options.
{ }	Encloses a list of choices, separated by vertical bars, from which you <b>must</b> choose one.
[ ]	Encloses an optional argument or a list of choices separated by vertical bars, from which you <b>may</b> choose one.
[ ?argName t_arg ]	Denotes a <i>key argument</i> . The question mark and argument name must be typed as they appear in the syntax and must be followed by the required value for that argument.
...	Indicates that you can repeat the previous argument.
	Used with brackets to indicate that you can specify zero or more arguments.
	Used without brackets to indicate that you must specify at least one argument.
, ...	Indicates that multiple arguments must be separated by commas.
=>	Indicates the values returned by a Cadence® SKILL® language function.
/	Separates the values that can be returned by a Cadence SKILL language function.

If a command-line or SKILL expression is too long to fit within the paragraph margins of this document, the remainder of the expression is moved to the next line and indented. In code excerpts, a backslash ( \ ) indicates that the current line continues on to the next line.

## Identifiers Used to Denote Data Types

Data type identifiers are used to indicate the type of value required by an API argument. These data types are denoted by a single letter that is prefixed to the argument label and is separated from the argument by an underscore; for example,  $\tau$  is the data type in  $\tau\_viewName$ . Data types and underscores are used only as identifiers; they must not be typed when specifying the argument in a function.

Prefix	Internal Name	Data Type
$a$	array	array
$A$	amsobject	AMS object
$b$	ddUserType	DDPI object
$B$	ddCatUserType	DDPI category object
$C$	opfcontext	OPF context
$d$	dbobject	Cadence database object (CDBA)
$e$	envobj	environment
$f$	flonum	floating-point number
$F$	opffile	OPF file ID
$g$	general	any data type
$G$	gdmSpecIIUserType	generic design management (GDM) spec object
$h$	hdbobject	hierarchical database configuration object
$I$	dbgenobject	CDB generator object
$K$	mapioobject	MAPI object
$l$	list	linked list
$L$	tc	Technology file time stamp
$m$	nmplIUserType	nmplI user type
$M$	cdsEvalObject	cdsEvalObject
$n$	number	integer or floating-point number
$o$	userType	user-defined type (other)
$p$	port	I/O port
$q$	gdmSpecListIIUserType	gdm spec list

## Analog Expression Language Reference

### Preface

---

Prefix	Internal Name	Data Type
<i>r</i>	defstruct	defstruct
<i>R</i>	rodObj	relative object design (ROD) object
<i>s</i>	symbol	symbol
<i>S</i>	stringSymbol	symbol or character string
<i>t</i>	string	character string (text)
<i>T</i>	txobject	transient object
<i>u</i>	function	function object, either the name of a function (symbol) or a lambda function body (list)
<i>U</i>	funobj	function object
<i>v</i>	hdbpath	hdbpath
<i>w</i>	wtype	window type
<i>sw</i>	swtype	subtype session window
<i>dw</i>	dwtype	subtype dockable window
<i>x</i>	integer	integer number
<i>y</i>	binary	binary function
<i>&amp;</i>	pointer	pointer type

---

For more information, see [\*Cadence SKILL Language User Guide\*](#).

# **Analog Expression Language Reference**

## **Preface**

---



---

# Basic Concepts

---

The following topics are discussed in this chapter:

- Overview on page 18
- Partial Evaluation on page 21
- Evaluating AEL Expressions on page 24

## 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.

## Analog Expression Language Reference

### Basic Concepts

---

### 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 <i>f</i> ( <i>flonum</i> )
Inherited Parameters	A <i>parseAsCEL</i> parameter on the parent instance
Instance Parameters	A <i>parseAsCEL</i> parameter on the current instance
Integer	A SKILL <i>x</i> ( <i>fixnum</i> )
List	A SKILL <i>l</i> ( <i>list</i> ) or a <i>typedef</i> by that name in the <code>il.h</code> file
Related Parameters	A term that defines instance and inherited parameters together
String	A SKILL <i>t</i> ( <i>text</i> )
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

## Analog Expression Language Reference

### Basic Concepts

---

*0.000000000005*. (For more information on suffixes and converters, refer to [“AEL Evaluation Symbols”](#) on page 65.)

## 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.

## 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
```

## Analog Expression Language Reference

### Basic Concepts

---

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
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

---

## Analog Expression Language Reference

### Basic Concepts

---

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
```



## Analog Expression Language Reference

### Basic Concepts

---

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	$2 * \pi * \text{pPar}(\text{"freq"}) / \sqrt{\text{iPar}(\text{"w"}) * \text{magic}(\text{c})} * 1.5\text{e-}6$
Globals	$2 * 3.14159 * \text{pPar}(\text{"freq"}) / \sqrt{\text{iPar}(\text{"w"}) * \text{magic}(1.23)} * 1.5\text{u}$
Instance CDF parameters	$2 * \pi * \text{pPar}(\text{"freq"}) / \sqrt{5\text{e-}9 * \text{magic}(\text{c})} * 1.5\text{u}$
Inherited CDF parameters	$2 * \pi * 10\text{e}6 / \sqrt{\text{iPar}(\text{"w"}) * \text{magic}(\text{c})} * 1.5\text{u}$
Related parameters	$2 * \pi * 10\text{e}6 / \sqrt{5\text{e-}9 * \text{magic}(\text{c})} * 1.5\text{u}$
Related parameters and suffixes	$2 * \pi * 10\text{e}6 / \sqrt{5\text{e-}9 * \text{magic}(\text{c})} * 1.5\text{e-}6$
Full evaluation	421.5e9

---

# **Analog Expression Language Reference**

## **Basic Concepts**

---

---

# AEL Functions

---

The following topics are discussed in this chapter:

- [AEL Utility Functions](#) on page 28
- [AEL Environment Functions](#) on page 41

## **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.

## Analog Expression Language Reference

### AEL Functions

---

#### 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
<code>min(b1, b2) &lt;= val &lt;= max(b1, b2)</code>	0
<code>(b1 &lt; b2) &amp;&amp; (val &lt; b1)</code>	<code>-(b1 - val)</code>
<code>(b1 &lt; b2) &amp;&amp; (val &gt; b2)</code>	<code>(val - b2)</code>
<code>(b1 &gt; b2) &amp;&amp; (val &lt; b2)</code>	<code>(b2 - val)</code>
<code>(b1 &gt; b2) &amp;&amp; (val &gt; b1)</code>	<code>-(val - b1)</code>

## Analog Expression Language Reference

### AEL Functions

---

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
<code>aelCheckRange(1, 2, 3)</code>	0
<code>aelCheckRange(3, 2, 1)</code>	0
<code>aelCheckRange(4, 1, 7)</code>	-3
<code>aelCheckRange(7, 1, 5)</code>	4
<code>aelCheckRange(low-epsilon, testVal, hi+hiTol)</code>	<code>tolerantRangeCheck</code>

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	<code>aelEngNotation</code>
	<code>aelPopSignifDigits</code>
	<code>aelStrDblNotation</code>
	<code>aelSuffixNotation</code>
	<code>aelSuffixWithUnits</code>

### **Related Environment Variable**

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

## Analog Expression Language Reference

### AEL Functions

---

`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

## Analog Expression Language Reference

### AEL Functions

---

#### **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`.

#### **aelStrDbINotation**

```
aelStrDbINotation(  
    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.

## Analog Expression Language Reference

### AEL Functions

---

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

See also

`aelEnvCreate`  
`aelEnvExecute`  
`aelEnvInterpret`  
`aelEnvListDeferredGlobals`

## Analog Expression Language Reference

### AEL Functions

---

#### 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 <i>ip</i> )
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 <i>C</i> , allowing for all but CDF parameters.)
g	Globals, except those deferred
i	Instance CDF parameters on the same cell (the <i>iPar( )</i> function)
l	Literal: no evaluation
o	All operators
p	CDF parameters inherited from the parent cell (the <i>pPar( )</i> function)
r	All related CDF parameters (same as evaluation mode <i>ip</i> )
s	Suffixes
t	Text substitution before compiling

---

## Analog Expression Language Reference

### AEL Functions

---

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.

## Analog Expression Language Reference

### AEL Functions

---

#### **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 do not 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.

## Analog Expression Language Reference

### AEL Functions

---

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
```

## Analog Expression Language Reference

### AEL Functions

---

```
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) ;
)
)
)
```

## **Analog Expression Language Reference**

### **AEL Functions**

---



---

## 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.

## Analog Expression Language Reference

### AEL Evaluation Symbols

### 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 ( $1e^{-3}$ )

## Analog Expression Language Reference

### AEL Evaluation Symbols

---

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

## Analog Expression Language Reference

### AEL Evaluation Symbols

---

### 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)

## Analog Expression Language Reference

### AEL Evaluation Symbols

---

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

## Analog Expression Language Reference

### AEL Evaluation Symbols

---

### 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)

# Index

---

## A

AEL [18](#)  
    example situations [18](#)  
    expression components [18](#)  
    SKILL functions [28](#)  
aelCheckRange [29](#)  
aelEngNotation [31](#)  
aelEnvCompile [42](#)  
aelEnvCreate [44](#)  
aelEnvExecute [46](#)  
aelEnvFreeCompExpr [47](#)  
aelEnvGetErrStr [48](#)  
aelEnvGetGlobal [49](#)  
aelEnvInterpret [50](#)  
aelEnvListDeferredFuncs [51](#)  
aelEnvListDeferredGlobals [52](#)  
aelEnvListExprFuncs [53](#)  
aelEnvListExprGlobals [54](#)  
aelEnvListFuncs [55](#)  
aelEnvListGlobals [56](#)  
aelEnvListGlobalsValues [57](#)  
aelEnvName [58](#)  
aelEnvSetGlobalList [61](#)  
aelEnvSetGlobals [59](#)  
aelGetSignifDigits [32](#)  
aelNumber [33](#)  
aelPopSignifDigits [34](#)  
aelPushSignifDigits [35](#)  
aelSetLineage [62](#)  
aelSignum [37](#)  
aelStrDbINotation [38](#)  
aelSuffixNotation [39](#)  
aelSuffixWithUnits [40](#)  
Analog Expression Language. *See*  
    AEL [18](#)

## B

basic syntax [21](#)

## C

CDF [20](#)

compile [21](#)  
compile function [42](#)  
complex number [19](#)  
complex strnum [19](#)  
Component Description Format. *See*  
    CDF [20](#)  
constant globals [19](#)  
constant table [68](#)  
converter table [68](#)  
converters [20](#)

## D

deferred functions [20, 51](#)  
deferred globals [20, 52](#)  
defined functions [20](#)  
defined globals [20](#)  
double [19](#)

## E

engineering notation [31](#)  
error retrieval [48](#)  
evaluation  
    example [22, 24](#)  
    full [23](#)  
    partial [21](#)  
evaluation mode [21, 44](#)  
execute function [46](#)  
expressions  
    attaching to components [24](#)  
    compiling [21](#)  
    example [24](#)  
    interpreting [21](#)  
    using as parameters [24](#)

## F

functions  
    deferred [20, 51](#)  
    defined [20](#)  
    utility [28](#)

## G

globals  
    constant [19](#)  
    deferred [20](#), [52](#)  
    defined [20](#)

## I

integer [19](#)  
interpret [21](#)  
interpret function [50](#)

## L

list [19](#)

## M

mathematical functions [70](#)  
mathematical operators [70](#)

## O

operator [70](#)

## P

parameters  
    inherited [19](#)  
    instance [19](#)  
    related [19](#)  
partial evaluation [21](#)  
    conditions [21](#)  
    example [22](#)

## Q

quick reference [65](#)

## R

range checking [29](#)

## S

significant digits [32](#)  
SKILL functions  
    aelCheckRange [29](#)  
    aelEngNotation [31](#)  
    aelEnvCompile [42](#)  
    aelEnvCreate [44](#)  
    aelEnvExecute [46](#)  
    aelEnvFreeCompExpr [47](#)  
    aelEnvGetErrStr [48](#)  
    aelEnvGetGlobal [49](#)  
    aelEnvInterpret [50](#)  
    aelEnvListDeferredFuncs [51](#)  
    aelEnvListDeferredGlobals [52](#)  
    aelEnvListExprFuncs [53](#)  
    aelEnvListExprGlobals [54](#)  
    aelEnvListFuncs [55](#)  
    aelEnvListGlobals [56](#)  
    aelEnvListGlobalsValues [57](#)  
    aelEnvName [58](#)  
    aelEnvSetGlobalList [61](#)  
    aelEnvSetGlobals [59](#)  
    aelGetSignifDigits [32](#)  
    aelNumber [33](#)  
    aelPopSignifDigits [34](#)  
    aelPushSignifDigits [35](#)  
    aelSignum [37](#)  
    aelStrDblNotation [38](#)  
    aelSuffixNotation [39](#)  
    aelSuffixWithUnits [40](#)  
string [19](#)  
strnum [19](#)  
suffix notation [39](#)  
suffix table [66](#)  
suffixes [20](#)  
syntax  
    basic [21](#)  
    guidelines [21](#)

## T

terminology, basic [19](#)

## U

utility functions [28](#)