Expression Reference Mar. 2, 2024

Expressions are used within the document and assessment system and can be entered by students as responses to questions. The same expression system is used for both, which allows student-entered expressions to be used to drive content like graphs.

Within an authored document or assessment, expressions can be represented in XML or in a binary storage format. When transmitted as a user entry, they are represented in a string format. All formats are equivalent and support the full scope of expressions.

An expression is made up of the objects which are organized in a tree. Every node in the tree can be evaluated, and its value passed along to the next node upward in the tree.

A tree node may evaluate to one of the following types:

* Boolean
* Integer
* Real (any real number, including rational and rational multiples of special irrational numbers)
* "Number" – a blanket term used where Integer or Real is allowed.
* String
* Span
* IntVector
* RealVector
* "NumberVector" – a blanket term used where IntVector or RealVector are allowed.
* IntMatrix
* RealMatrix
* "NumberMatrix" – a blanket term used where IntMatrix or RealMatrix are allowed.
* "Any" – a blanket term when any of the above types is allowed.
* Error

The nodes in an expression tree are:

* Boolean constants
* String constants
* Integer constants
* Real constants
* The imaginary unit.
* IntVector constants
* RealVector constants
* IntMatrix constants
* RealMatrix constants
* Variable references (typed) – variable names may include "multi-index" notation and accents.
* Vector of integer-valued expressions (⇨IntVector)
* Vector of real-valued expressions (⇨RealVector)
* Matrix of integer-valued expressions (⇨IntMatrix)
* Matrix of real-valued expressions (⇨RealMatrix)
* Unary operators
  + positive (Integer⇨Integer, Real⇨Real, IntVector⇨IntVector, RealVector⇨RealVector, IntMatrix⇨IntMatrix, RealMatrix⇨RealMatrix)
  + negative (Integer⇨Integer, Real⇨Real, IntVector⇨IntVector, RealVector⇨RealVector, IntMatrix⇨IntMatrix, RealMatrix⇨RealMatrix)
* Binary operators
  + subtract (Integer2⇨Integer, Real2⇨Real, IntVector2⇨IntVector, RealVector2⇨RealVector, IntMatrix2⇨IntMatrix, RealMatrix2⇨RealMatrix)
  + divide (Number2⇨Real, NumberVector/Number⇨RealVector, NumberMatrix/Number⇨RealMatrix)
  + power (Number2⇨Real)
  + remainder (Integer2⇨Integer)
  + less-than (Number2⇨Boolean)
  + greater-than (Number2⇨Boolean)
  + less-than-or-equal (Number2⇨Boolean)
  + greater-than-or-equal (Number2⇨Boolean)
  + equal (Any2⇨Boolean)
  + not-equal (Any2⇨Boolean)
  + scale-vector (Integer/IntVector⇨IntVector, Number/ RealVector⇨RealVector)
  + scale-matrix (Integer/IntMatrix⇨IntMatrix, Number/ RealMatrix⇨ RealMatrix)
* Trinary operators
  + approximately-equal (Number3⇨Boolean, last argument is tolerance)
  + test (If-then-else) operator (Boolean/Any2⇨Any)
* Nary operators
  + add (IntegerN⇨Integer, NumberN⇨Real, IntVectorN⇨IntVector, RealVectorN⇨RealVector, IntMatrixN⇨IntMatrix, RealMatrixN⇨RealMatrix)
  + multiply (IntegerN⇨Integer, NumberN⇨Real)
  + and (BooleanN⇨Boolean)
  + or (BooleanN⇨Boolean)
* Functions
  + absolute value (Integer⇨Integer, Real⇨Real, IntVector⇨IntVector, RealVector⇨RealVector, IntMatrix⇨IntMatrix, RealMatrix⇨RealMatrix)
  + cosine (Number⇨Real)
  + sine (Number⇨Real)
  + tangent (Number⇨Real)
  + secant (Number⇨Real)
  + cosecant (Number⇨Real)
  + cotangent (Number⇨Real)
  + inverse cosine (Number⇨Real)
  + inverse sine (Number⇨Real)
  + inverse tangent (Number⇨Real or Number2⇨Real)
  + hypotenuse (Number2⇨Real)
  + hyperbolic cosine (Number⇨Real)
  + hyperbolic sine (Number⇨Real)
  + hyperbolic tangent (Number⇨Real)
  + natural exponential (Number⇨Real)
  + natural logarithm (Number⇨Real)
  + logarithm base 10 (Number⇨Real)
  + signum (Number⇨Integer, -1/0/1)
  + ceiling (Number⇨Integer)
  + floor (Number⇨Integer)
  + round (Number⇨Integer)
  + factorial (Integer⇨Integer)
  + combinations (Integer2⇨Integer) – this is the binomial coefficient notation
  + permutations (Integer2⇨Integer)
  + permute (Integer⇨IntVector)
  + average (NumberVector or NumberN⇨Real)
  + min (NumberVector or NumberN⇨Real)
  + max (NumberVector or NumberN⇨Real)
  + square root (Number⇨Real)
  + cube root (Number⇨Real)
  + radians to degrees (Number⇨Real)
  + degrees to radians (Number⇨Real)
  + not (Boolean⇨Boolean)
  + greatest common divisor (IntVector or IntegerN⇨Integer)
  + least common multiple (IntVector or IntegerN⇨Integer)
  + greatest squared factor (Integer⇨Integer)
  + greatest cubed factor (Integer⇨Integer)
  + lowercase (String⇨String)
  + uppercase (String⇨String)
  + radian numerator (Integer⇨Integer, numerator in reduced coefficient on π for a degree measure)
  + radian denominator (Integer⇨Integer, denominator in reduced coefficient on π for a degree measure)
* Switch operator with Integer condition, any number of cases of Any type (but all compatible in type)

**Constants:**

* Boolean constants are represented as atomic TRUE or FALSE symbols that behave as a single glyph.
* String constants are represented as the string in quotation marks. The quotation marks cannot be selected or deleted – attempting to select either will select the whole string. The quotation marks allow editors to distinguish between a cursor position before the string, and a position within the string.
* Integer constants are usually represented as a sequence of decimal digits with optional minus sign. When building an expression, if the user enters "-5", for example, the expression engine would probably represent this as a unary negation applied to the integer 5. An integer constant can be represented in hexadecimal, with leading 0x or in binary with leading 0b.
* Real constants can include double-precision floating point values or "irrationals" which have a rational coefficient on either π, e, or the square root of an integer. If the integer whose root is taken is 1, this is an ordinary rational number. If the value is a Double, it is represented in decimal format, with possible engineering notation. If the value is irrational based on the root 1, it is represented as a fraction. If its value is irrational based on π or e or some other root, it is represented as a fraction multiplying the irrational value.
* Span constants must be named variable references within expressions.
* IntVector (RealVector) constants are represented as an ordered list of Integer (Real) values, separated by commas, surrounded by square brackets. The brackets behave as the quotes for string constants.
* IntMatrix (RealMatrix) constants are represented as a grid of space-separated and aligned Integer (Real) values, surrounded by square brackets. The brackets behave as the quotes for string constants.

**Variable References:**

* Represented by the variable name, in a particular font to distinguish from function names.

**Vectors and Matrices:**

* Integer Expression Vector (Number Expression Vector) are represented as an ordered list of expressions that evaluate to Integer (Number) values, separated by commas, surrounded by square brackets. Each expression would be wrapped in parentheses (or similar) to support cursor navigation.
* Integer Expression Matrix (Number Expression Matrix) values are represented as a grid of subexpressions that evaluate to Integer (Number) values, space-separated and aligned, surrounded by square brackets.

**Unary Operators:**

* The operator itself is represented by a single + or – character – its child is the object immediately following.

**Binary Operators:**

* Subtraction, less-than, greater-than, less-than-or-equal, greater-than-or-equal, equal, and not-equal are represented as a single character, their children are the objects to the left and right.
* Division has multiple representations – either as a "slash", a "super/sub with slash", or vertically as a fraction.
* The power operator is represented as a base to an exponent.
* The remainder operator is represented as a function with 2 arguments.
* The scale-vector operator is represented with juxtaposition.

**Trinary Operators:**

* Approximately-equals is represented as a three-argument function.
* The test operator is represented as an IF-THEN-ELSE sequence. Nested test operators in the ELSE clause could be represented using ELSEIF.

**Nary Operators:**

* All are represented with single characters between children, where AND and OR are characters.

**Functions:**

* All are represented by a function name, open parenthesis, then the function's arguments (separated by commas if more than one), then a closing parenthesis.

**Switch Operators:**

* Represented as a SWITCH marker followed by open parenthesis, the condition expression, and close parenthesis.
* Each case (or default) represented by CASE then the integer value, a colon, then the expression, and a closing semicolon.

**Editing vs. Evaluation vs. Presentation**

An expression editor allows the user to enter an expression in a natural and accessible way, using the keyboard or a set of buttons. The expression is represented in a glyph-oriented format, where some constructions like fractions or function names are treated as a single glyph. The expression being edited does not need to be in a valid state.

After each change during editing, the editor can attempt to parse the expression into an expression tree that could be evaluated. If the expression is valid, the tree will be a valid representation, and will store enough information to reproduce the glyph sequence used to build the expression.

A valid expression tree can then be rendered in mathematical form. Ideally, an editor would render this form as the user enters the expression so they can be certain their expression is being parsed as intended. If the expression contains no variables, it can be evaluated, and its value displayed as well. If the expression contains variables, an editor could allow the user to set variable values and then display the results of evaluation, keeping in mind that a valid expression may still evaluate to an error.

An editor could offer a "LaTeX syntax mode" where a limited subset of LaTeX would be interpreted.

**Editing Data Structures**

The editor tracks a linear sequence of glyphs and allows the user to move a cursor within that sequence. At certain points, sub-expressions are created that are edited separately, with the constraint that subexpressions appear to the user to be embedded and part of the overall sequence of glyphs – the cursor enters a subexpression, advances through its glyphs, then advances to the position following the subexpression in the containing expression. This creates a restriction on selection – a subexpression may not be only partially selected if the selection extends outside the subexpression.

Allowed Glyphs:

* TRUE and FALSE Boolean constants, π and e real constants
* Digits 0-9, and a special "E" engineering notation glyph
* The decimal point and comma
* Curly braces, square brackets, parentheses
* Double-quotes
* +, -, \*, %, |, &, =, <, >, !
* Any UTF-16 characters (if between quotes)
* Any UTF-16 letters (outside quotes, treated as variable names)
* Base/Exponent constructions (with subexpressions)
* Fraction constructions (with subexpressions)
* Radical constructions (with subexpressions)
* If-Then-Else constructions (with subexpressions)
* Case constructions (with subexpressions)
* Function invocations (with a subexpression – includes "approx-equals")

Canvas editor:

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer screen

Description automatically generatedA screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated