Contents

Articles

References

Article Licenses

Additional MathObjects documentation

These documents have been gathered from other sites, principally the AIM-WeBWorK site.

The contents of these documents has been folded in to the main MathObjects documentation.

http://webwork.maa.org/moodle/mod/wiki/view.php?id=160&page=MathObjectsContextOverview

moved to http://webwork.maa.org/wiki/Introduction_to_MathObjects

http://webwork.maa.org/moodle/mod/wiki/view.php?id=160&page=ContextList

moved to http://webwork.maa.org/wiki/IntroductionToContexts#List_of_Basic_Contexts"

http://webwork.maa.org/moodle/mod/wiki/view.php?id=160&page=ContextModification1

moved to http://webwork.maa.org/wiki/IntroductionToContexts#Common_Context_methods"

http://webwork.maa.org/moodle/mod/wiki/view.php?id=160&page=MathObjectsOverview

moved to http://webwork.maa.org/wiki/Introduction_to_MathObjects#Examples_of_Using_MathObjects"

http://webwork.maa.org/moodle/mod/wiki/view.php?id=160&page=MathObjectDetails

this detailed information (and more) is provided by the POD documentation page: http://webwork.maa.org/doc/cvs/pg_CURRENT/doc/MathObjects/UsingMathObjects.html

Answer Checker Options (MathObjects)

The action of a MathObjects answer checker can be modified by passing options to the `cmp()` method. For example:

```
ANS(Real(pi)->cmp(showTypeWarnings => 0));
```

will prevent the answer checker from reporting errors due to the student entering the wrong type of answer (say a vector rather than a number).

Options common to all answer checkers

There are a number of options common to all the checkers:

Option	Description	Default
`showTypeWarnings => 1` or `0`	Show/don't show messages about student answers not being of the right type.	`1`
`showEqualErrors => 1` or `0`	Show/don't show messages produced by trying to compare the professor and student values for equality, e.g., conversion errors between types.	`1`
`ignoreStrings => 1` or `0`	Show/don't show type mismatch errors produced by strings (so that `NONE` will not cause a type mismatch in a checker looking for a list of numbers, for example).	`1`
`studentsMustReduceUnions => 1` or `0`	Controls whether or not students answers are allowed to include overlapping, redundant, or uncombined intervals.	`1`
`showUnionReduceWarnings => 1` or `0`	Do/don't show warnings for unreduced unions. This includes warnings about overlapping sets, uncombined intervals (like `(0,1] U [1,infinity]`), and repeated elements in sets.	`1`
`cmp_class => "a thing"`	Specifies the name to use in error messages (like type warnings) to identify the type of object the student is expected to enter.	the object's class name

Options for Individual MathObject Classes

In addition to these, the individual MathObject types have their own options. These are documented in the individual pages for the various MathObject classes, linked below:

- **Real** class
- **Infinity** class
- **Complex** class
- **Point** class
- **Vector** class
- **Matrix** class
- **List** class
- **Interval** class
- **Set** clas
- **Union** class
- **String** class
- **Formula** class

Changing the Default Options

You can set the default options for each MathObject class using the `cmpDefaults` field of the Content. To set the options for the Real class, for example, use

```
Context()->{cmpDefaults}{Real} = {ignoreStrings => 0, ignoreInfinity => 0};
```

This will force the `ignoreStrings` and `ignoreInfinity` values to be set to 0 automatically for all answer checkers formed from Real MathObjects. These values can still be overridden by explicitly setting the option on the `cmp()` call.

Answer Checkers (MathObjects)

Basic Answer Checkers

Checkers are available for each of the types of values that the parser can produce (numbers, complex numbers, infinities, points, vectors, intervals, sets, unions, formulas, lists of numbers, lists of points, lists of intervals, lists of formulas returning numbers, lists of formulas returning points, and so on).

To use one of these checkers, simply call the `cmp()` method of the object that represents the correct answer. For example,

```
$n = Real(sqrt(2));
ANS($n->cmp);
```

will produce an answer checker that matches the square root of two. Similarly,

```
ANS(Vector(1,2,3)->cmp);
```

matches the vector `<1,2,3>` (or any computation that produces it, e.g., `<i+2j+3k>`, or `<4,4,4>-<3,2,1>`), while

```
ANS(Interval("(-inf,3]")->cmp);
```

matches the given interval. Other examples include

```
ANS(Infinity->cmp);
ANS(String('NONE')->cmp);
ANS(Union("(-inf,$a) U ($a,inf)")->cmp);
```

and so on.

Formulas are handled in the same way:

```
ANS(Formula("x+1")->cmp);

$a = random(-5,5,1); $b = random(-5,5,1); $x = random(-5,5,1);
$f = Formula("x^2 + $a x + $b")->reduce;
ANS($f->cmp);
ANS($f->eval(x=>$x)->cmp);

$x = Formula('x');
ANS((1+$a*$x)->cmp);

Context("Vector")->variables->are(t=>'Real');
$v = Formula("<t,t^2,t^3>"); $t = random(-5,5,1);
ANS($v->cmp);
ANS($v->eval(t=>$t)->cmp);
```

Answer Checkers for Lists

Lists of items can be checked easily:

```
ANS(List(1,-1,0)->cmp);
ANS(Compute("1,-1,0")->cmp);

ANS(List(Point($a,$b),Point($a,-$b))->cmp);
ANS(Compute("($a,$b),($a,-$b)")->cmp);

ANS(List(Vector(1,0,0),Vector(0,1,1))->cmp);
ANS(Compute("<1,0,0>,<0,1,1>")->cmp);

ANS(Compute("(-inf,2),(4,5)")->cmp);

ANS(Formula("x, x+1, x^2-1")->cmp);
ANS(Compute("<x,2x>,<x,-2x>,<0,x>")->cmp);

ANS(List('NONE')->cmp);
```

and so on. The last example may seem strange, as you could have used `ANS(String('NONE')->cmp)`, but there is a reason for using this type of construction. You might be asking for one or more numbers (or points, or whatever) or the word `NONE` if there are no numbers (or points). If you used `String('NONE')->cmp`, the student would get an error message about a type mismatch if he entered a list of numbers, but with `List('NONE')->cmp`, he will get appropriate error messages for the wrong entries in the list.

In the case of a list of points where the answer is `NONE`, you should tell MathObject what the expected type of answer is, so that appropriate messages can be given. E.g.,

```
ANS(List('NONE')->cmp(typeMatch => 'Value::Point');
```

would give appropriate messages for points rather than numbers (the default type).

It is often appropriate to use the list checker in this way even when the correct answer is a single value, if the student might type a list of answers.

On the other hand, using the list checker has its disadvantages. For example, if you use

```
ANS(Interval("(-inf,3]")->cmp);
```

and the student enters `(-inf,3)`, she will get a message indicating that the type of interval is incorrect, while that would not be the case if

```
ANS(List(Interval("(-inf,3]"))->cmp);
```

were used. (This is because the student doesn't know how many intervals there are, so saying that the type of interval is wrong would inform her that there is only one.)

The rule of thumb is: the individual checkers can give more detailed information about what is wrong with the student's answer; the list checker allows a wider range of answers to be given without giving away how many answers there are. If the student knows there's only one, use the individual checker; if there may or may not be more than one, use the list checker.

Note that you can form lists of formulas as well. The following all produce the same answer checker:

```
ANS(List(Formula("x+1"),Formula("x-1"))->cmp);

ANS(Formula("x+1,x-1")->cmp); # easier

$f = Formula("x+1"); $g = Formula("x-1");
ANS(List($f,$g)->cmp);

$x = Formula('x');
ANS(List($x+1,$x-1)->cmp);
```

See the files in pg/doc/MathObjects/problems [1] for more examples of using the parser's answer checkers.

Controlling the Details of the Answer Checkers

The various features of the answer checkers can be controlled by passing options to the `cmp()` method of a MathObject.

```
ANS($mathObject->cmp(options));
```

For example:

```
ANS(Real(pi)->cmp(showTypeWarnings => 0));
```

will prevent the answer checker from reporting errors due to the student entering the wrong type of answer (say a vector rather than a number).

See the Answer Checker Options documentation for details about the available options, and the individual MathObject class documentation for the options specific to each class. See the Answer Checkers and the Context page for information about how the Context can control the answers that students can enter.

Custom Answer Checkers

While the built-in answer checkers do a good job of testing a student response for agreement with a given correct answer, sometimes an answer may require a more sophisticated or customized check. For example, you might ask a student to provide a solution to a given implicit equation for which there are infinitely many solutions and would like to count any of them as correct. You can do this by providing your own answer checker as a parameter to a MathObject answer checker. This lets you get the advantage of the MathObject parsing, syntax checking, type checking, and error messages, while still giving you the control of deciding when the student's answer is correct.

See the Custom Answer Checkers and Custom Answer Checkers for Lists documentation for details about how to create your own custom checkers using MathObjects.

References

[1] https://github.com/openwebwork/pg/tree/master/doc/MathObjects/problems

Answer Checkers and the Context

Disabling Functions

Some things, like whether trigonometric functions are allowed in the answer, are controlled through the `Context()` rather than through the MathObjects answer checker itself. For example,

```
Context()->functions->disable('sin','cos','tan');
```

would remove those three functions from use. One would need to remove `cot`, `sec`, `csc`, `arcsin`, `asin`, etc., to do this properly, however. It is also possible to remove whole categories of functions at once, e.g.

```
Context()->functions->disable('Trig');
```

would disable all trig functions, while

```
Context()->functions->disable('All');
Context()->functions->enable('sqrt');
```

would allow only the `sqrt` function to be used in student answers. The list of categories is available in the Context Function Categories list.

Note that some functions can be obtained via operators (e.g., `abs(x)` can be obtained via `|x|` and `sqrt(x)` can be obtained by `x^(1/2)` or `x^.5`, so you might need to remove more than just the named functions to limit these operations).

Disabling Operators

Which arithmetic operations are available is controlled through `Context()->operations`. For example,

```
Context()->operations->undefine('^','**');
```

would disable the ability for students to enter powers. Note that multiplication and division have several forms (in order to make a non-standard precedence that allows things like `sin(2x)` to be entered as `sin 2x`). So if you want to disable them you need to include all of them. E.g.,

```
Context()->operations->undefine('*',' *','* ');
Context()->operations->undefine('/',' /','/ ','//');
```

would be required in order to make multiplication and division unavailable. See the Context Operator Table for more details.

Finally, absolute values are treated as a specialized form of parenthesis, so to remove them, use

```
Context()->parens->remove('|');
```

The pg/macros/ [1] directory contains a number of predefined contexts that limit the operations that can be performed in a student answer. For example, the `contextLimitedNumeric.pl` file defines contexts in which students can enter numbers, but no operations, so they would have to reduce their answer to a single number by hand. There are limited contexts for complex numbers, points, and vectors, and there are also specialized contexts for entering polynomials, or where powers are restricted in various ways.

Tolerances and Limits

The tolerances used in comparing numbers are part of the Context as well. You can set these via:

```
Context()->flags->set(
  tolerance    => .0001,       # the relative or absolute tolerance
  tolType      => 'relative',  # or 'absolute'
  zeroLevel    => 1E-14,       # when to use zeroLevelTol
  zeroLevelTol => 1E-12,       # smaller than this matches zero
                               #  when one of the two is less
                               #  than zeroLevel
  limits       => [-2,2],      # limits for variables in formulas
  num_points   => 5,           # the number of test points
);
```

Note that for testing formulas, you can override these values by setting these fields of the formula itself:

```
$f = Formula("sqrt(x-10)");
$f->{limits} = [10,12];

$f = Formula("log(xy)");
$f->{limits} = .1,2],[.1,2; # x and y limits
```

You can also specify the test points explicitly:

```
$f = Formula("sqrt(x-10)");
$f->{test_at} = 11],[11.5],[12;  # use these plus random ones

$f = Formula("log(xy)");
$f->{test_points} = [[.1,.1],[.1,.5],[.1,.75],
                     [.5,.1],[.5,.5],[.5,.75]];  # test only at these
```

You can specify the value at the same time you create the object, as in

```
$f = Formula("sqrt(x-1)")->with(limits=>[10,12]);
```

It is also possible to set the limits of variables in the context itself,

```
Context()->variables->set(x => {limits => [10,12]});
```

or when a variable is created in the Context,

```
Context()->variables->add(t => ['Real',limits=>[1,2]]);
```

or even when the answer checker is specified,

```
ANS($f->cmp(limits=>[10,12]));
```

See Also

- Context Function Categories
- Context Operator Table
- MathObject Answer Checkers

References

[1] https://github.com/openwebwork/pg/tree/macros

Common Contexts

The main pre-defined contexts are the following:

Name	Description
`Numeric`	No points, vectors, matrices, complex numbers, or intervals are allowed.
`Complex`	No points, vectors, matrices, or intervals are allowed.
`Point`	Nearly the same as the `Vector` below, but the angle bracket and `ijk` notation is not allowed, and vector operations on points are not defined.
`Vector`	The constants `i`, `j`, and `k` are defined as coordinate unit vectors, and vector cross and dot products are allowed. No complex numbers, matrices, or intervals are allowed.
`Vector2D`	Same as `Vector` above, but `i` and `j` are defined as coordinate unit vectors in , and `k` is not defined.
`Matrix`	Same as `Vector` above, but square brackets define matrices instead of points or intervals.
`Interval`	Similar to `Numeric` context, but `(a,b)`, `(a,b]`, `[a,b)`, and `[a,b]` create real Intervals rather than lists or errors. Finite sets of reals are created using `{a,b,c}` (with as many or as few numbers are needed).
`Full`	This context is used to seed the others, and is only intended for internal use. Matrix, vector, point, intervals, and complex numbers all can be created in this context; `i` is the complex i equal to <math>\sqrt{-1}. The variables are `x`, `y`, and `z`.

Common MathObject Methods

There are a number of methods that are common to all MathObjects, which are described below. Some classes have additional methods, and many of these are listed in the documentation for the individual MathObject Classes. You call a method on a MathObject as you would a method for any Perl object, using the `->` operator with the MathObject on the left and the method name and its arguments (if any) on the right. E.g.,

```
$mo->method;               # when there are no arguments
$mo->method($arg);         # for one argument
$mo->method($arg1,$arg2); # for two arguments
# and so on
```

Methods Common to All Classes

Method	Description
`$mo->cmp` `$mo->cmp(options)`	Returns an answer checker for the MathObject. The `cmp` method can accept a number of settings that control the tolerances for the comparison, special options of the comparison (e.g., parallel vectors rather than equal vectors), the types of error messages produced (e.g., messages about individual coordinates that are wrong), and other features of the check. These are described in the MathObjects Answer Checkers [1] POD documentation. All of the answer checkers are defined in the file `pg/lib/Value/AnswerChecker.pm`.
`$mo->value`	Returns an array containing the data that represents the object. For a Real, it is just the perl real number that corresponds to it; for a Complex number, it is the real and imaginary parts (as Reals). For an Infinity, it is the string needed to obtain the Infinity. For a Point or Vector, it is the coordinates of the Point or Vector (as MathObjects). For a Matrix, it is an array of rows of the Matrix, where the rows are references to arrays of MathObjects. For an Interval, it is the two endpoints (as Reals) followed by strings that are the open and close parentheses or brackets for the Interval. For a Set, it is an array of the elements of the set (as Reals). For a Union, it is an array of the Sets and Intervals that make up the Union. For a String, it is a perl string representing the value of the string. For a Formula, it is a little more complicated. If the Formula is an explicit Point, Vector, Matrix, Interval, etc. (e.g., `Formula("<x,x+1>")`), then the value is an array of Formulas that are the coordinates. If the Formula is an expression (e.g., `Formula("<x,1> + <2x,-x>")` or `Formula("norm(<x,x+1>)")` then the value is just the original Formula.
`$mo->data`	Returns a reference to the array of data that represents the object.
`$mo->length`	Returns the number of elements in the array returned by the `value()` method, e.g., the number of coordinates in a point or vector, or the number of rows in a matrix.
`$mo->extract(`*i*`,...)`	Returns the i-th element from the data for `$mo`. If more than one index is used, recursively extracts from the result, so that for a Matrix, `$M->extract(`*i*`,`*j*`)` returns the (i,j)-th entry, and for a list of lists, `$L->extract(`*i*`,`*j*`)` is the j-th element of the i-th list.
class`->new(data)` `$mo->new(data)`	Creates a new instance of the *class* or the class of `$mo` using the given data. This what the various class constructor functions do; e.g., `Real(5)` effectively calls `Value::Real->new(5)`.
class`->make(data)` `$mo->make(data)`	Creates a new instance of the *class* or the class of `$mo` using the given data, but without all the error checking of type conversion. So `data` must be an array of MathObjects of the correct type. This is used internally when the types of the data are known to be correct, for efficiency, but it is always safer to use `new`.

`$mo->with(name => value)`	This copies the object, and sets its property with the given name to the given value. You can supply multiple name/value pairs separated by commas. This gives you the ability to initialize the object when you create it, or to make a copy with specific settings. E.g.
	`$f = Formula("sqrt(x-10)")->with(limits => [10,12]);`
	`$a = Real(pi/2)->with(period => 2*pi);`
	Note that the copy is not a deep copy (i.e., if `$mo` has references to other objects, the returned object will reference the same ones rather than copies of them). In particular, this is the case for the `{data}` property. Use `$mo->copy` to get a deeper copy, or `$mo->new($mo->string)` to get a separate version that shares no data with `$mo`

`$mo->without("name",...)`	Returns a copy of `$mo` where the named properties have been removed. Note that, as with `with()`, the copy is not a deep one, so the two copies of `$mo` will share any references to other objects. In particular, this is the case for the `{data}` property. Use `$mo->copy` to get a deeper copy, or `$mo->new($mo->string)` to get a separate version that shares no data with `$mo`
`$mo->copy`	Returns a copy of `$mo` with its `{data}` array copied as well, so that the copy doesn't share the data with the original. Note that setting `$mo2 = $mo1` does not copy `$mo1`; instead, both `$mo2` and `$mo1` both point to the *same* MathObject, so changes to one will change the other. Use `$mo2 = $mo1->copy` to obtain a separate copy for `$mo2`.
`$mo->context` `$mo->context($context)`	The first form returns the context in which `$mo` was created. When passed a reference to a Context object, this sets the context for `$mo` to be the given context.
`$mo->inContext($context)`	This calls `context($context)` and then returns `$mo` so that it can be used in a chain of method calls, e.g. `$f->inContext($g->context)->with(limits => $g->{limits});`
`$mo->getFlag("name")` `$mo->getFlag("name",default)`	Returns the object's value for a given context flag. The value can come from a variety of locations, which are searched in the following order (the first that has a property with the given flag name will have that propery's value returned): the object itself, the Formula that created it (if it was the result of an `eval()` call, for example), the AnswerHash associated with the object (if it was the source for an answer checker; this gives access to the flags passed to the `cmp` method), the Context in which the object was created, the currently active Context, or the default value (if given as the second argument), otherwise the return value is `undef`. This is useful in custom answer checkers for finding out the settings of things like the tolerances, the flags passed to the answer checker, and so on.
`$mo->typeMatch($object)`	This determines if the `$object` is "compatible" with the given MathObject for equality or inequality comparisons. For example, a Real will allow equality comparison to an Infinity, and a Complex will allow comparison to a Real. It is best to use `typeMatch` in your own custom answer checkers if you need to check if a student's answer is of the correct type rather than using something like `ref()` to check if the two are the same type of object (which is too restrictive, in general).

`$mo->class`
`$mo->type`
	These are two methods that help you determine what kind of MathObject you are working with. They can be useful in custom answer checkers if you want to know more about what kind of object a student answer is. The `class` method tells you the class of object (like Real, Complex, Point, Formula, etc.), while the `type` method tells you what kind of return value a Formula has (non-Formulas are considered constant-valued Formulas when computing the `type`). The `class` is essentially the package name from the Value package of the MathObject, while the `type` is the package name from the Parser package name for the result of the Formula.
	`Real(5)->class;  # produces "Real"`
	`Real(5)->type;  # produces "Number"`
	`Point(1,2)->class;  # produces "Point"`
	`Point(1,2)->type;  # produces "Point"`
	`Formula("3x+1")->class;  # produces "Formula"`
	`Formula("3x+1")->type;  # produces "Number"`

`$mo->TeX`	Returns a string which represents the object in format. For example
	`$f = Formula("(x+1)/(x-1)");`
	`BEGIN_TEXT`
	`The formula is \(f(x) = \{$f->TeX\}\).`
	`END_TEXT`
	Since it is common to need the expression in the problem's text, there is a special Context setting that tells MathObjects to output their format whenever they are substituted into a string. That is enabled via the `Context()->texStrings` command, and turned off by `Context()->normalStrings`, as in the following example:
	`$f = Formula("(x+1)/(x-1)");`
	`Context()->texStrings;`
	`BEGIN_TEXT`
	`The formula is \(f(x) = $f\)`
	`END_TEXT`
	`Context()->normalStrings;`
	This avoids having to call the `TeX()` method, and prevents the need for using `\{...\}` to get the format.

`$mo->string`	Returns a string similar to that used to create the object, in the form that a student would use to enter the object in an answer blank, or that could be used in `Compute()` to create the object. The string may have more parentheses than the original string used to create the object, and may include explicit multiplication rather than implicit multiplication, and other normalization of the original format.
`$mo->perl`	Returns a string which represents the object as Perl source code. This is used internally, and would rarely be needed in a PG problem.
`$mo->eval` `$mo->reduce`	All MathObjects have these methods, but for most, they just return `$mo`. They are mostly used for the Formula class, but are available on all so that you don't have to check the type before calling them. A few classes, like Unions, implement `reduce()`.

`$mo->ans_rule(width)`
`$mo->named_ans_rule(width)`
`$mo->named_ans_rule_extension(width)`
	These operate like the PG macros of the same names, but are here as the counterpart to the `ans_array` methods below. That way, you can call one or the other in order to get the proper answer rules for `$mo`. It also helps to see what answer rule goes with what answer if you use `\{$mo->ans_rule(10)\}` in your `BEGIN_TEXT/END_TEXT` blocks rather than just `\{ans_rule(10)\}`.

`$mo->ans_array(width)`
`$mo->named_ans_array(width)`
`$mo->named_ans_array_extension(width)`
	For Points, Vectors, and Matrices, these produce multiple answer blanks, one for each coordinate or entry, that are all tied to the one answer checker for `$mo`. This forces (or allows, depending on how you look at it) the student to enter each coordinate separately, while you still only need to handle one answer checker. See the Matrix documentation for an example.

`$mo->isZero`	Returns `1` if `$mo` corresponds to whatever zero means for that type (e.g., for a Vector, it is a zero vector), and `0` otherwise.
`$mo->isOne`	Returns `1` if `$mo` corresponds to whatever one means for that type (e.g., for a Matrix, it is an identity matrix), and `0` otherwise.
`$mo->isSetOfReals`	Returns `1` if `$mo` is an Interval, Set, or Union (or subclasses of those), and `0` otherwise.
`$mo->canBeInUnion`	Returns `1` if `$mo` is an Interval, Set, or Union, or is another object whose string version could look like an Interval (e.g., a Point with two coordinates where the first is less than the right). This is used in contexts where parentheses can produce Points, for example, to promote them to Intervals when appropriate.
`$mo->showClass`	Returns a string representing the class of `$mo` suitable for use in error messages. The string includes "an" or "a", so `Value->Error("You can't take the square root of %s",$mo->showClass)` could produce the message "You can't take the square root of a Vector".
`$mo->showType`	Returns a string representing the type of `$mo` suitable for use in error messages. Note that the class and the type are not the same; e.g., the type of a Formula is the type of it return value, while it's class is Formula (actually `Value::Formula`). So `$f->showClass` might be "a Formula returning a Real Number" for a formula `$f`, while `$f->showType` would produce "a Real Number".
`$mo->inherit($mo1,...)`	Copy the properties that are not already present in `$mo` from `$mo1` (and any other objects passed to it). This is used by things like the binary operations to make sure that the result inherits the attributes of the the two operands.
`$mo->noinherit`	Returns the list of properties that should *not* be copied by `inherit()`.
`$mo->Package($context,"class",$noerrors)`	Returns the name of the package that implements the given class in the given Context. For example, `$mo->Package($mo->context,"Real")` probably will return `"Value::Real"`. The Context could have override the default package for Reals, however, to use a subclass of Reals that implements additional functionality, in which case that subclass package name would be returned. If there is no package for the given class, an error will be produced, unless `$noerrors` is `1`.

Support Functions

These are functions that are part of the Value package, but do not refer to a MathObject directly, so are not methods of any class. You call them from the Value package directly. Mostly they are general utility functions, or functions that test an unknown value to see if it is a MathObject or could be converted to one.

Method	Description
`Value->Error(message,$s1,...)`	Produces an error message by inserting the values of `$s1` and any other arguments into the `message` using `printf`-style substitution, and then throws an error condition (i.e., your program or subroutine will stop at that point). If used in a custom answer checker, the error usually will appear in the messages area of the results table at the top of the page. For example, `Value->Error('You can't take the square root of %s',$mo->showType)` might produce the message "You can't take the square root of a Vector".
`Value->NameForNumber(n)`	Returns a string like "`first`", "`second`", etc. that can be used in an error message. For example, `Value->Error("Your %s point is incorrect",Value->NameForNumber(i))` would produce the message "Your fifth point is incorrect" when `i` is 5. This can be useful in custom answer checkers that deal with lists of objects or coordinates of points or vectors.
`Value::contextSet($context,flags)`	This sets the given flags to the given values in the given context and returns a hash representing the flags and their original settings. This is useful if you want to change the settings of flags temporarily (e.g., in an answer checker), but want to reset them when done. For example, `my $flags = Value::contextSet($correct->context, reduceConstants => 1, reduceConstantFunctions => 0, ); ... do something with Formulas here ... Value::contextSet($correct->context, %{$flags});` would set the value of `reduceConstants` and unset `reduceConstantFunctions` and return the original settings in `$flags`. After the processing that needs these flag settings is complete, the second call to `contestSet()` returns the values to their original settings.
`Value::protectHTML(string)`	Returns a string where HTML special characters have been properly escaped (e.g., `<`, `>` and `&` have been replaced by their corresponding HTML entities.
`Value::preformat(string)`	Returns a string where HTML special characters have been replaced by their HTML entities, and newlines have been replaced by ` `.
`Value::makeValue(data)`	Attempts to convert the `data` into an appropriate MathObject and returns that. For example, a Perl real will be converted to a Real object, and a string will be converted to a String, provided it is in the current Context. It is safe to pass a MathObject to `makeValue()`, as it will be returned without change; thus you can use `$x = makeValue($x)` to guarantee that `$x` is a MathObject.
`Value::matchNumber(string)`	Returns `1` if the contents of the string would be interpreted as a (signed) number in the current Context, `undef` otherwise.
`Value::matchInfinite(string)`	Returns `1` if the contents of the string would be interpreted as an infinity in the current Context, `undef` otherwise.

`Value::classMatch($mo,"class",...)`	Returns `1` if `$mo` is a MathObject belonging to one of the classes specified, and `0` otherwise. To be considered in the given class, `$mo` must satisfy one of the following: 1. Its `class()` method returns the name of the given class. (Note that `class()` usually returns the last component of the object's package name, so an object in package `my::Real` would have `class()` return `"Real"` so could match `Value::matchClass($mo,"Real")`. 2. Its package name is `"Value::`*`class`*`"` for the given class. 3. The value of `$mo->{is`*`Class`*`}` is defined an non-zero (e.g., if `$mo->{isReal} = 1`, then `Value::matchClass($mo,"Real")` is true). 4. The package for `$mo` is the same as the Context's package associated with the given class (e.g., `Value::matchClass($mo,"Real")` will be true if `$mo->context->{value}{Real}` is `ref($mo)`). 5. Its class is a subclass of `Value::`*`Class`* for the given class, and `$mo->{is`*`Class`*`}` is not `0`. These conditions are checked for each class name passed to `classMatch()` until one matches or the end of the list is reached. It is safe to pass a non-MathObject to `classMatch()` as no methods are called directly on `$mo`.
`Value::isContext($x)`	Returns `1` if `$x` is a reference to a Context object, `undef` otherwise.
`Value::isParser($x)`	Returns `1` if `$x` is an instance of a class that is a subclass of `Parser::Item` (i.e., is part of a parse tree for a Formula object), and `undef` otherwise.
`Value::isValue($x)`	Returns `1` if `$x` is a MathObject, and `undef` otherwise. To be a MathObject, `$x` must satisfy one of the following 1. It is in a package whose name begins with `Value::` 2. It has the `{isValue}` set to something other than 0 (e.g., `$mo->{isValue} = 1`, or 3. It is an instance of a subclass of the `Value` class. The latter two are the usual ways to make your own classes that act as MathObjects.
`Value::isFormula($x)`	Shorthand for `Value::classMatch($x,"Formula")`
`Value::isReal($x)`	Shorthand for `Value::classMatch($x,"Real")`
`Value::isComplex($x)`	Shorthand for `Value::classMatch($x,"Complex")`
`Value::isNumber($x)`	Returns `1` if `$x` represents a number, `undef` otherwise. If `$x` is a Formula, this is true when its return value is a number; otherwise it is true when `Value::classMatch($x,"Real","Complex")` or `Value::matchNumber($x)` is true. Note that this is true when `$x` is a Complex number as well as a Real number.
`Value::isRealNumber($x)`	Returns `1` if `$x` represents a real number, `undef` otherwise. If `$x` is a Formula, this is true when its return value is a Real number; otherwise it is true when `Value::classMatch($x,"Real")` or `Value::matchNumber($x)` is true.

References

[1] http://webwork.maa.org/pod/pg_TRUNK/doc/MathObjects/MathObjectsAnswerCheckers.html

Common MathObject Properties

There are a number of properties that are common to all MathObejcts, as described below. Note that these properties *apply* to all MathObjects, but not every MathObject will have all of them defined. Some classes have additional properties, and many of these are listed in the documentation for the individual MathObject Classes. You set or get the value of a property of a MathObject as you would a property for any Perl object, using the `->` operator with the MathObject on the left and the propery name in braces on the right. E.g.,

```
$x = $mo->{property};
$mo->{property} = $x;
```

In addition to the properties listed below, you can also set any of the Context flags listed in the Context flags documentation page as properties of a MathObject. In this case, the value will override that stored in the Context, but for that MathObject only. For example, you could set the `tolerance` on a Real, or the `limits` for a Formula in that way.

Properties Common to All Classes

Option	Description	Default
`$mo->{context}`	A reference to the Context object under which the MathObject was created. Even if the Context is changed, the MathObject will continue to operate within its original Context.	`Context()`
`$mo->{equation}`	A reference to the Formula object from which a MathObject was created (if it was the result of evaluating a Formula).	parent Formula
`$mo->{correct_ans}`	The string to use for the correct answer when students request answers (after the due date). This is set automatically by	`undef`
`$mo->{data}`	A reference to an array containing the information defining the MathObject (e.g., the coordinates of a Point, the elements of a List, the intervals in a Union, or the real and imaginary parts of a Complex number.	object data
`$mo->{format}`	A `printf`-style string indicating how real numbers should be formatted for display. If the format ends in `#`, then trailing zeros are removed after the number is formatted. Example: `"%.4f"` would format numbers using 4-place decimals.	`"%g"`
`$mo->{open}`	For list-type objects (e.g., Points, Vectors, Matrices, Sets), the symbol to use at the left of the object when it is displayed.	See specific class page
`$mo->{close}`	For list-type objects (e.g., Points, Vectors, Matrices, Sets), the symbol to use at the right of the object when it is displayed.	See specific class page
`$mo->{noinherit}`	A reference to an array of property names that will not be included when a copy of a MathObject is made. This includes things like `text_values` and `f` for Formulas, and other values that are maintained automatically by the object.	object specific
`$mo->{isValue}`	When creating subclasses of MathObject classes, you may need to set this value so that the MathObject library will recognize your object class as belonging to the Value package.	`undef`
`$mo->{is`*`Class`*`}` E.g., `$mo->{isVector}`	When creating a subclass of a MathObject class, you may need to set this so that the MathObjects library will properly recognize your object as being of the given type when it does type-matching for things like comparisons with student answers, and so on. The *`Class`* can be any of the MathObject class names, e.g, `$mo->{isReal}`.	`undef`

`$mo->{ans_name}`
`$mo->{ans_rows}`
`$mo->{ans_cols}`
	These are used by the `$mo->ans_array` method to store information about the name used for the answer array, and its size.	`undef`

`$mo->{format_options}`	This is a reference to a hash that specifies options for the answer array layout, including the characters to use for the left and right delimiters and the separator in answer arrays. It provides values for `open`, `close`, and `sep`; the defaults are dependent on the class, and correspond to the open and close delimiters for the object, and either comma or blank as the separator, depending on the class.	object specific

Variables and Constants

Variable	Description	Default
`%Value::Type`	A hash containing type definitions needed for the values of `Parser::Item` objects. Use `TEXT(join(',',lex_sort(keys(%Value::Type))));` in the PG labs [1] to get a list of the available types.	
`$Value::answerPrefix`	The answer name prefix to use for answer arrays created by the `ans_array()` method.	`"MaTrIx"`

`$Value::CMP_ERROR`
`$Value::CMP_WARNING`
`$Value::CMP_MESSAGE`
	Error values used with `$context->setError()` to indicate the severity of the error in answer checkers.	2
		3
		4

References

[1] http://hosted2.webwork.rochester.edu/webwork2/wikiExamples/MathObjectsLabs2/2/?login_practice_user=true

Complex (MathObject Class)

Complex Class

The Complex class implements complex numbers with "fuzzy" comparison (due to the fact that the real and complex parts are handled using MathObject Reals). Complex numbers can be created most readily in the `Complex` Context.

Creation

Complex numbers are created via the `Complex()` function, or by `Compute()`, or through the use of the pre-defined complex number `i`. As with Real objects, you can add, subtract, multiply, and perform the other usual arithmetic operations on Complex objects, and can compute `sin()`, `sqrt()` and the other standard functions on Complex arguments. In addition, there are `arg()` and `mod()`, which return the argument or modulus of a Complex value, and `Re()` and `Im()` that return its real or imaginary part. The `conj()` function returns the complex conjugate of a Complex object. The value `i` represents $\sqrt{-1}$ and can be used in your Perl expressions to produce complex numbers. Note that you must use `-(i)` to obtain $-i$ in Perl expressions (but not in strings parsed by MathObjects, such as student answers or the arguments to `Compute()`).

```
Context("Complex");

$z = Complex(2,3);
$z = Complex([2,3]);
$z = 2 + 3 * i;
$z = Complex("2 + 3i");
$z = Compute("2 + 3i");
```

```
$w = sin($z);       # same as Compute("sin(2+3i)");
$w = conj($z);      # same as Complex("2 - 3i");
$w = $z**2;         # same as Compute("(2+3i)^2");
$w = $z + 5*i;      # same as Complex("2 + 8i");
$w = $z - (i);      # parens needed in Perl expressions
```

Answer Checker

As with all MathObjects, you obtain an answer checker for a Complex object via the `cmp()` method:

```
ANS(Compute("2+3i")->cmp);
```

The Complex class supports the common answer-checker options. There are no additional options.

Methods

The Complex class supports the Common MathObject Methods, and the following additional methods:

Method	Description
`Re($c)` or `$c->Re`	Returns the real part of `$c` as a Real MathObject.
`Im($c)` or `$c->Im`	Returns the imaginary part `$c` as a Real MathObject.
`arg($c)` or `$c->arg`	Returns the argument of `$c` (i.e., the angle in radians counterclockwise from the x-axis) as a Real MathObject.
`mod($c)` or `$c->mod`	Returns the modulus of `$c` (i.e., its distance from the origin) as a Real MathObject. Also available as `abs` or `norm`.
`conj($c)` or `$c->cong`	Returns the complex conjugate of `$c` as a Complex MathObject.

Properties

The Complex class supports the Common MathObject Properties, and the following additional ones:

Property	Description	Default
`$c->{period}`	When set, this value indicates that the complex number is periodic, with period given by this value.	`undef`
`$c->{logPeriodic}`	When `period` is defined, and `logPeriodic` is set to `1` this indicates that the periodicity is logarithmic (i.e., the period refers to the log of the value, not the value itself).	`0`

Context flags

Using Context Flags

Each Context object contains a collection of *flags* that control various aspects of how the Context operates. The main flags are described below, though some contexts may have additional flags. To view the flags for the current Context, enter

```
TEXT( pretty_print(Context()->flags->all));
```

to your PG file. (The interactive PGLabs page is an easy place to do this.)

You can change the value of one or more Context flags as in the following examples:

```
Context()->flags->set(
   reduceContstants => 0,
   reduceConstantFunctions => 0
);

Context()->flags->set(formatStudentAnswers => 'parsed');

Context()->flags->set(limits => [0,1]);
```

Many flags can be applied directly to a MathObject to override the setting in the Context. For example,

```
$f = Formula("sqrt(x-5)")->with(limits => [5,7]);
```

sets the limits for `$f` without changing the default used for other Formulas.

Similarly, some flags can be overridden in the answer checker's option list:

```
ANS($f->cmp(formatStudentAnswers => 'parsed'));
ANS($r->cmp(tolerance => .05, tolType => 'absolute'));
```

In this way, it is not always necessary to change the Context itself in order to affect the flag values.

Further Reading

- Introduction to Contexts
- Common Contexts
- POD documentation [1] -- see the files beginning with `context`
- How to list Context flags

Table of Context Flags

Flags for Reals

Flag	Description	Default
useFuzzyReals	Do/don't use tolerance and toltype to determine when two numbers are equal. When 1, the comparisons are "fuzzy" (meaning allowed to be slightly off), when 0 the native Perl comparison is used. Note that this applies to *all* comparisons of MathObjects, both in Perl code as well as in student answer checking. So when set, Real(1) == Real(1.0000001) will be true (in the default Context settings). This lets you perform comparisons in exactly the same way that the answer checkers do.	1
tolerance	The allowed error when comparing numbers for equality. When tolType is "absolute", the absolute value of the difference must be less than this; when tolType is "relative", then the difference of the two divided by the first must be less than this in absolute value, provided both are sufficiently larger than zero as determined by the zeroLevel (if one is closer to zero than that, then the zeroLevelTol is used instead as an absolute tolerance).	.001
tolType	Determines whether checks for equality use absolute or relative tolerances. See tolerance for details.	"relative"
zeroLevel	When tolType is "relative", then numbers smaller than this in absolute value will cause equality checks to switch to absolute comparisons with tolerance given by zeroLevelTol. This prevents the relative tolerance computations from blowing up.	1E-14
zeroLevelTol	The tolerance to use when tolType is "relative", and one of the numbers being compared is smaller than the zeroLevel. See zeroLevelTol and tolerance for details.	1E-12
useBaseTenLog	Do/don't use base 10 when computing log(). When 0, use base e.	0
NumberCheck	A code reference to a subroutine used to check the format of numbers when a student answer is parsed. It is passed the Parser::Number object to be checked, and can use its {value} or {value_string} properties during the check, and its Error() method to report errors. See the NoDecimals implementation in pg/lib/Parser/Number.pm [2] for an example.	undef

Flags for Formulas

Flag	Description	Default
reduceConstants	Do/don't combine constants when the operands of an operator are constant. E.g., Formula("2 * 3") will become Formula("6") automatically when this is set to 1, but will remain unchanged when it is set to 0.	1
reduceConstantFunctions	Do/don't compute functions when their arguments are all constants. E.g., Formula("sqrt(2)") will become Formula("1.4142135623731") when this is set to 1, but will remain unchanged when it is set to 0.	1
checkUndefinedPoints	Do/don't allow test points where a Formula is undefined. This can be useful if you want to make sure a student's answer is undefined at the same locations as the correct answer. E.g., you want to distinguish between ln(x) and ln(|x|). It often helps to use test_at in this case, and you may want to increase num_points so that sufficient test points are located where the Formula is defined.	0
limits	The range of values to use for variables in determining the test points for Formula comparisons. This the the default used for all variables; the limits for individual variables can be set using Context variables object, as in this example: `Context()->variables->set(x=>{limits=>[0,1]});` See the Introduction to Contexts for more details.	[-2,2]
num_points	The number of random test points to use during function comparisons. This is in addition to any test_at points, but if test_points are given, no random points are given.	5
granularity	This gives the minimum distance between possible test points used to compare two Formulas. The distance is determined by dividing the length of the interval specified by limits by the granularity. For example, with limits=>[0,5] and granularity=>5, the test points will be integers, while with granularity=>20, the test points will be multiples of 1/4.	1000

resolution	An alternative to granularity, this value specifies the spacing between possible test points for Formula comparisons. For example, with limits=>[0,5] and resolution=>1, the test points will be integers, while resolution=>1/4 would mean the test points are multiples of 1/4. If set, this takes precedence over granularity.	undef
max_adapt	This is the largest value allowed for an adaptive parameter. If an adaptive parameter exceeds this limit, a warning will be issued.	1E8
max_undefined	When checkUndefinedPoints is set, this gives the maximum number of test points that can be at locations where the Formula is undefined. If undef, there is no limit, but it may be best to set a limit so that some points where the function actually *is* defined will guaranteed to be used.	undef

Flags for Intervals, Sets, and Unions

Flag	Description	Default
ignoreEndpointTypes	Do/don't ignore the closed/open status of Interval endpoints when comparing Intervals. When set to 1, two intervals whose endpoints match numerically will be equal, regardless of whether they are open or closed at the endpoints. This is set to 1 automatically for student answers if requireParenMatch is set to 0 in the options to the answer checker.	0
reduceSets	Do/don't remove repeated elements from Sets when they are created. For student answers, this is automatically set to the value of studentsMustReduceUnions for the duration of the answer check.	1
reduceSetsForComparison	Do/don't reduce Sets (temporarily) before they are compared. For student answers, this is automatically set to the value of showUnionReduceWarnings for the duration of the answer check when studentsMustReduceUnions is set.	1
reduceUnions	Do/don't combine overlapping intervals or Sets when Unions are created. For student answers, this is automatically set to the value of studentsMustReduceUnions for the duration of the answer check.	1
reduceUnionsForComparison	Do/don't reduce Unions (temporarily) before they are compared. For student answers, this is automatically set to the value of showUnionReduceWarnings for the duration of the answer check when studentsMustReduceUnions is set.	1

Flags for Strings

Flag	Description	Default
allowEmptyStrings	Do/don't allow an empty string to form a String object, even though there is no definition for that. (Useful for if you want to process blank answers.)	1
infiniteWord	The word used by the Infinite object to represent infinity (i.e., the string it will output) when displayed.	"infinity"

Flags Controlling Formatting

Flag	Description	Default
formatStudentAnswer	Determines how the student's answer is shown in the "Entered" box in the results table when the students submits the answers. When set to "evaluated", it is the final result of the expression. When set to "parsed" it is the Formula entered by the student prior to being evaluated. When set to "reduced", it is the result of calling the reduce() method on the student's answer.	"evaluated"
showExtraParens	Determines the level to which extra parentheses are inserted to help make the meaning clear. When set to 0, only parentheses required by the operator precedences are inserted. When set to 1, extra parentheses are added to help make the meaning more clear in situations where students tend not to know the precedence rules very well. When set to 2, extra parentheses are added to make the expression painfully unambiguous.	1
ijk	Do/don't use ijknotation for displaying all vectors. When set to 0, angle brackets are used as delimiters to show vectors in coordinate form.	0
ijkAnyDimension	Do/don't convert expressions involving ijknotation to the proper dimension before comparing two vectors (e.g., Vector context could still be used for vectors in the plane).	1

Flags Controlling Warnings

The following are used by contextTypeset.pl to control when warnings are issued.

Flag	Description	Default
allowBadOperands	Do/don't check whether operator operand types are correct; when 0, warnings are issued.	0
allowMissingOperands	Do/don't allow operators to have missing operands; when 0, warnings are issued.	0
allowBadFunctionInputs	Do/don't check whether function input types are correct; when 0, warnings are issued.	0
allowMissingFunctionInputs	Do/don't allow functions to have missing arguments; when 0, warnings are issued.	0
allowWrongArgCount	Do/don't allow functions to have too many arguments; when 0, warnings are issued.	0

References

[1] http://webwork.maa.org/pod/pg_TRUNK/

[2] https://github.com/openwebwork/pg/blob/master/lib/Parser/Number.pm#L81

Context Function Categories

The functions available to be used in student answers are controlled via the `functions` object of the current Context. See Answer Checkers and the Context for details about how to enable and disable these functions. This can be done either singly or by categories of functions.

The available function categories are the following:

Category	Functions Included
`SimpleTrig`	`sin`, `cos`, `tan`, `sec`, `csc`, `cot`
`InverseTrig`	`asin`, `acos`, `atan`, `asec`, `acsc`, `acot`, `arcsin`, `arccos`, `arctan`, `arcsec`, `arccsc`, `arccot`, `atan2`
`SimpleHyperbolic`	`sinh`, `cosh`, `tanh`, `sech`, `csch`, `coth`
`InverseHyperbolic`	`asinh`, `acosh`, `atanh`, `asech`, `acsch`, `acoth`, `arcsinh`, `arccosh`, `arctanh`, `arcsech`, `arccsch`, `arccoth`
`Numeric`	`log`, `log10`, `exp`, `sqrt`, `abs`, `int`, `sgn`, `ln`, `logten`
`Vector`	`norm`, `unit`
`Complex`	`arg`, `mod`, `Re`, `Im`, `conj`
`Hyperbolic`	all of `SimpleHyperbolic` and `InverseHyperbolic`
`Trig`	all of `SimpleTrig`, `InverseTrig`, and `Hyperbolic`
`All`	all of `Trig`, `Numeric`, `Vector`, and `Complex`

See `pg/lib/Parser/Context/Default.pm` [1] for the definitions of all the functions, and `pg/lib/Parser/Context/Functions.pm` [2] for the code that defines the categories.

References

[1] https://github.com/openwebwork/pg/blob/master/lib/Parser/Context/Default.pm
[2] https://github.com/openwebwork/pg/blob/master/lib/Parser/Context/Functions.pm#L55

Context Operator Table

The operators available to be used in student answers are controlled via the `operators` object of the current Context. See Answer Checkers and the Context or the Introduction to Contexts for details about how to alter the list of operators.

The operators and their precedences are listed below. Those shown in grey are not entered directly, but are generated automatically when the proper situation arises.

Operator	Precedence	Associativity	Description
,	0	left	Separates entries in points, vectors, lists, etc.
+	1	left	Addition
-	1	left	Subtraction
U	1.5	left	Union of intervals and sets
><	2	left	Cross product of vectors (only in `Vector` and `Matrix` contexts)
.	2	left	Dot product of vectors (only in `Vector` and `Matrix` contexts)
*	3	left	Multiplication
/	3	left	Division
//	3	left	Division displayed horizontally
space	3	left	Multiplication (generated automatically by implied multiplication)
u+	6	left	Unary plus (generated automatically when + is used in unary position)
u-	6	left	Unary minus (generated automatically when - is used in unary position)
^	7	right	Exponentiation
**	7	right	Exponentiation
fn	7.5	left	Function call (generated automatically by functions)
!	8	right	Factorial
_	9	left	Element extraction (for vectors, matrices, lists, points)

In addition to these, there are some extra operators used to implement a non-standard precedence arrangement that is designed to allow things like `sin 2x` to be interpreted as `sin(2x)` rather than as `(sin(2)) x`. In the standard precedence settings, these have the precedences or the corresponding operators listed above; in the non-standard precedence, they are as given below. Here, "*space*" represents an actual space.

Operator	Precedence	Associativity	Description
*space**	2.8	left	Multiplication with preceding space
**space*	2.8	left	Multiplication with trailing space
space/	2.8	left	Division with preceding space
/*space*	2.8	left	Division with trailing space
fn	2.9	left	Function call
space	3.1	left	Implied multiplication

See `pg/lib/Parser/Context/Default.pm` [1] for the full definitions of the operators.

Course-Wide Customizations

It is often the case that different courses use different notation for the same concept. For example, some use angle brackets for matrix delimiters, as in `<4,0,-1>`, while others use parentheses. Some use , , and for the coordinate unit vectors, while others use $\hat{\imath}$, $\hat{\jmath}$, and $\hat{k}$, or $\vec{\imath}$, $\vec{\jmath}$, and $\vec{k}$, or some other notation. Some prefer the ijk-notation and others the delimited coordinate form.

It would be nice if you could set your preferences for this globally within your course and have it affect all the problems you use, rather than having to edit them all to use the notation that you would like. Although you don't have perfect control over this, there are some things you can do to move in that direction. The mechanism for doing this is to use the `parserCustomization.pl` file to make customized versions of the Contexts from the problems you are using so that the Context settings match your preferences.

First, make a copy of `pg/macros/parserCustomization.pl` [1] and put it in your course's `templates/macros` folder (using the File Manager). Then edit it to include Context changes that you desire. If you want to make changes to the `Vector` Context, for example, you will need to make a copy of that in the `%context` hash, and then make your modifications, as in the example below. The `Context()` command knows to look in the `%context` hash first, and then to look in the default Context list, so when you put a new Context here, that makes it available via `Context()`. In particular, if you give the new Context the same name as the original one, then `Context()` will find your modified one first rather than the original one. In this way, you are overriding the settings in the original.

```
$context{Vector} = Parser::Context->getCopy("Vector");
$context{Vector}->flags->set(ijk=>1);          # force output in ijk-notation
$context{Vector}->parens->remove('<');         # force entry of vectors in ijk-notation
```

Note that the last command will cause questions that use `Compute("<...>")` to produce error messages, since you have disabled the angle brackets as a means of creating vectors.

To allow students to enter vectors using parentheses rather than angle brackets, use

```
$context{Vector} = Parser::Context->getCopy("Vector");
$context{Vector}->{cmpDefaults}{Vector} = {promotePoints => 1};
$context{Vector}->lists->set(Vector=>{open=>'(', close=>')'});
```

This actually just turns Points into Vectors in the answer checker for Vectors, and displays Vectors using parens rather than angle brackets. The student is really still entering what MathObjects thinks is a Point, but since Points get promoted automatically, that should work. If a problem checks if a student's value is actually a Vector, however, that will not be true.

An alternative would be to use

```
$context{Vector} = Parser::Context->getCopy("Vector");
$context{Vector}->lists->set(Vector=>{open=>'(', close=>')'});
$context{Vector}->parens->set('('=>{type=>'Vector'});
```

which actually *does* force the student's answer to be a Vector object. The problem here is that there is now no way to enter an actual Point, since parentheses now produce Vectors.

Other similar changes can be made, but such changes may cause some problems to fail, since the defaults are no longer what they expect them to be. So if you take this route, be sure to check the problems you use very carefully before assigning them to students.

References

[1] https://github.com/openwebwork/pg/raw/master/macros/parserCustomization.pl

Creating Custom Contexts

Suppose that you have accumulated a nontrivial number of context modifications that you use frequently when writing problems. You can combine all of these modifications into a single file and define your own context.

First, what do you want to call your context? Let's say we will name it `MyContext`. Whatever you end up naming your context, it is recommended practice to begin your filename with `context` and it must end with `.pl` as all macro files do. So your filename might be `contextMyContext.pl`. When we're done we'll put the file in the `/pg/macros/` sudbdirectory of the WebWork installation, or in your course's `templates/macros` directory.

You will then include the file in the `loadMacros()` call at the top of your file along with the other macros you load. You should place your context file after the standard macro files. So, for example,

```
loadMacros(
  "PGstandard.pl",
  "MathObjects.pl",
  "contextMyContext.pl",
  "PGcourse.pl",
);
```

When defining a new context in your file, the code will proceed as follows:

1. copy an existing context which is close to what you want,
2. modify what you want changed.

For example, this line copies the `Numeric` Context into a new `MyContext` Context:

```
$main::context{MyContext} = Parser::Context->getCopy("Numeric");
```

Here, the `%main::context` hash is used to store custom Contexts. The `Context()` command knows to look here first for contexts, and then to look in the default Context list, so when you put a new Context here, that makes it available via `Context("MyContext")`.

The following lines modify some aspects of this context:

```
$main::context{MyContext}->functions->disable("Trig");
$main::context{MyContext}->parens->set('('=>{type=>'Vector'});
etc...
```

The last line in the file should be

```
1;
```

which makes the file return a non-zero value (a zero value would indicate an error within the file).

That's it. Save it, put it in `/pg/macros/` or your course `templates/macros` directory, and load it in your problems with `loadMacros()`.

You can save more complex changes as well, like the added functions or operators listed in the Modifying Contexts (advanced) page.

Custom Answer Checkers

While the built-in MathObject answer checkers do a good job of testing a student response for agreement with a given correct answer, sometimes an answer may require a more sophisticated or customized check. For example, you might ask a student to provide a solution to a given implicit equation for which there are infinitely many solutions and would like to count any of them as correct. You can do this by providing your own answer checker as a parameter to a MathObject answer checker. This lets you get the advantage of the MathObject parsing, syntax checking, type checking, and error messages, while still giving you control over deciding when the student's answer is correct.

You do this by providing a `checker` subroutine, as in the following example:

```
Context("Point");
$a = random(2,10,1);
$x = random(-5,5,1);
$y = $a - $x;

BEGIN_TEXT
Find a point \((x,y)\) that is a solution to \(x+y=$a\).
$PAR
\((x,y)\) = \{ans_rule(15)\}
END_TEXT

ANS(Point($x,$y)->cmp(
  showCoordinateHints => 0,                # doesn't make sense to give hints in this case
  checker => sub {
    my ($correct,$student,$ansHash) = @_;  # get correct and student MathObjects
    my ($sx,$sy) = $student->value;        # get coordinates of student answer
    return ($sx + $sy == $a ? 1 : 0);      # return 1 if correct, 0 otherwise
  }
));
```

The subroutine that is given as the value of the `checker` parameter is called when the student has typed an answer that parses properly and is compatible with a point, so you don't have to worry about type-checking the student answer yourself, and are guaranteed to have a MathObject to work with. Note that your answer checker is tied to a given MathObject, so the type checking and error messages are appropriate for that type of object. Also, this is what will be shown as the correct answer if the student requests answers after the due date, so you should be sure that you provide an actual correct answer, even if you don't use `$correct` within your checker.

Your checker subroutiune is passed three items, the correct answer (as a MathObject), the student's answer (as a MathObject), and a reference to the AnswerHash that is being used to process this answer. In the example above, `$correct` will be the `Point($x,$y)` that was used to create the original MathObject for the answer checker, and `$student` will be the point that the student typed.

The answer hash (`$ansHash`), holds additional data about the answer checker and the student's answer. That data includes all the flags passed to the answer checker; e.g., in the example above, `$ansHash->{showCoordinateHints}` will be `0`, while `$ansHash->{showTypeWarnings}` will be `1` (the default). Other fields that are useful include

Property	Description
`$ansHash->{correct_ans}`	The correct answer string
`$ansHash->{isPreview}`	Whether the "Preview" button was pressed or not (you might want to limit error messages when this is true).
`$ansHash->{original_student_ans}`	The unmodified string originally typed by the student. It has not been processed in any way.
`$ansHash->{student_formula}`	The Formula obtained from parsing the student answer. If it is a constant-valued Formula, then `$student` is the result of evaluating this formula (i.e., it will be a Real or Point or some other MathObject rather than a Formula); if it is not constant, then `$student` will be this Formula.
`$ansHash->{student_value}`	The value passed to `$student`.
`$ansHash->{correct_value}`	The value passed to `$correct`.

The return value of your checker should be a value between 0 and 1 that indicates how much credit the student's answer should receive. Use 1 for full credit, .5 for 50% credit, and so on.

Error messages can be generated via the `Value->Error()` function. For example,

```
Value->Error("Your value should be positive") if $student <= 0;
```

Note that this function will cause the checker to return with an error condition, so there is no need to return a score in this case.

Saving Custom Checkers

If you have written a custom checker that you want to use with more than one problem, or more than one answer in a single problem, then you might want to put that checker into a separate macro file that can be loaded into each problem that needs it. One way to do this is to make a variable that stores the code reference for the subroutine, and then pass that to the `checker` option. For example,

```
$pointChecker = sub {
  my ($correct,$student,$ansHash) = @_;  # get correct and student MathObjects
  my ($sx,$sy) = $student->value;        # get coordinates of student answer
  return ($sx + $sy == $a ? 1 : 0);      # return 1 of correct, 0 otherwise
};

ANS(Point($x,$y)->cmp(showCoordinateHints => 0, checker => $pointChecker));
```

The `$pointChecker` could be placed in a separate macro file placed in the course `templates/macros` directory and loaded via `loadMacros()` at the beginning of the problem along with the other macro files.

Note, however, that this subroutine relies on a constant, `$a`, and it would be better to pass that to the checker. There are several ways to handle that. One would be to pass the value of `$a` to the custom answer checker as an option to the `cmp()` and retrieve it from the `$ansHash` within the point checker subroutine. This is illustrated below.

```
$pointChecker = sub {
  my ($correct,$student,$ansHash) = @_;  # get correct and student MathObjects
  my ($sx,$sy) = $student->value;        # get coordinates of student answer
  my $a = $ansHash->{a};                 # get the parameter value from cmp() call
  return ($sx + $sy == $a ? 1 : 0);      # return 1 of correct, 0 otherwise
};

ANS(Point($x,$y)->cmp(showCoordinateHints => 0, checker => $pointChecker, a => $a));
```

The other approach is to use a *closure* to create a subroutine that depends on an external value. In this case, we use a wrapper function that accepts the value of `$a` as a parameter, and then returns the actual subroutine that is to be used as the `checker`. This subroutine has access to the value of `$a` that was passed to the outer function. Here is an example:

```
sub pointChecker {
  my $a = shift;              # can be referenced by subroutine below
  return sub {
    my ($correct,$student,$ansHash) = @_;  # get correct and student MathObjects
    my ($sx,$sy) = $student->value;        # get coordinates of student answer
    return ($sx + $sy == $a ? 1 : 0);      # return 1 of correct, 0 otherwise
  };
}

ANS(Point($x,$y)->cmp(showCoordinateHints => 0, checker => pointChecker($a)));
```

A more sophisticated point checker could be passed the Formula to be evaluated as well as the value of `$a` and the checker could use the Formula's `eval()` method to get its value at the student's point and compare that to the value of `$a`.

Custom Answer Checkers for Lists

The built-in answer checkers for the List, Union, and Set classes are a bit more complicated than those for the other classes, because the order of elements in the List (or Intervals in a Union, or elements of a Set) usually doesn't matter. If you provide a checker routine for one of these types of MathObjects, it does not refer to the list as a whole, but to the individual elements in the list. Management of the list and determining how many elements are matched, what the partial credit should be, and what error messages to produce is all handled by the main list answer checker. You only provide a routine that determines when an entry in the student's list matches one in the correct list. Your checker will be called repeatedly on the various combinations of student and correct answers to determine if the student list matches the correct list (regardless of order).

When the `checker` is called from the list checker, it has two additional paramters:

```
checker => sub {
  my ($correct,$student,$ansHash,$nth,$value) = @_;
  ...
}
```

Here `$nth` is a word indicating which student answer is being tested (e.g., for the third element in the student's list, `$nth` will be `" third"` (note the leading space). This can be used in error messages, for example, to help the student identify where an error occurred. The `$value` is the name of the type of object that is in the list, e.g., `"point"`. So you could use `"Your$nth $value is not correct"` as an error message in order to get something like `"Your third point is incorrect"` as the output.

As usual, you can use `Value->Error()` to generate an error message. The list checker will trap it and put it in the message area preceded by `"There is a problem with your nth answer:"` (where "nth" is replaced by the proper word for the answer that produced the error message). If you want to produce a message that doesn't include this prefix, use

```
$correct->{context}->setError("message","",undef,undef,$Value::CMP_WARNING);
return 0;
```

where "message" is the error message you want to produce.

If you want to act on the List (or Union or Set) as a whole, then you need to use the list_checker parameter instead. This specifies a subtroutine that will handle checking of the entire list, overriding the default list checking. That means you will be responsible for dealing with the possibly different order of the student answers from the correct answer, providing error messages about individual entries in the list, and so on.

The list_checker gets called with four parameters: a reference to an array containing the list of correct answers, a reference to an array of the student answers, a reference to the AnswerHash, and a string containing the name of the type of elements expected (for use in error messages). So a list checker looks like

```
list_checker => sub {
  my ($correct,$student,$ansHash,$value) = @_;
  ...
}
```

You can refer to the individual entries in $correct or $student as $correct->[$i] or $student->[$i], where $i is an integer representing the position within the list (the first entry is numbered 0 not 1).

Note that in a list_checker for a List object, you will have to do type checking yourself to check if the types of the entries are correct, since a list can consist of any type of elements.

The list_checker should return an array consisting of the number of student entries that are correct followed by any error messages that should be displayed (each on a separate line in the messages section).

An Example List Checker

Here is an example of a list checker:

```
Context("Point");

BEGIN_TEXT
Three distinct points \((x,y)\) that satisfy the equation \(x+y=5\) are:
\{ans_rule(20)\}
END_TEXT

ANS(List("(0,5),(1,4),(2,3)")->cmp(list_checker => sub {
  my ($correct,$student,$ansHash,$value) = @_;
  my $n = scalar(@$student);  # number of student answers
  my $score = 0;              # number of correct student answers
  my @errors = ();            # stores error messages
  my $i, $j;                  # loop counters

  #
  #  Loop though the student answers
  ##
  for ($i = 0; $i < $n; $i++) {
    my $ith = Value::List->NameForNumber($i+1);
    my $p = $student->[$i];   # i-th student answer
    #
    #  Check that the student's answer is a point
    #
```

```
      if ($p->type ne "Point") {
         push(@errors,"Your $ith entry is not a point");
         next;
      }
      #
      #  Check that the point hasn't been given before
      #
      for ($j = 0, $used = 0; $j < $i; $j++) {
        if ($student->[$j]->type eq "Point" && $student->[$j] == $p) {
          push(@errors,"Your $ith point is the same as a previous one") unless $ansHash->{isPreview};
          $used = 1; last;
        }
      }
      #
      #  If not already used, check that it satisfies the equation
      #    and increase the score if so.
      #
      if (!$used) {
        my ($a,$b) = $p->value;
        if ($a + $b == 5) {$score++} else {
          push(@errors,"Your $ith point is not correct") unless $ansHash->{isPreview}
        }
      }
    }
    #
    #  Check that there are the right number of answers
    #
    if (!$ansHash->{isPreview}) {
      push(@errors,"You need to provide more points") if $i < 3;
      push(@errors,"You have given too many points") if $score > 3 && $i != $score;
    }
    return ($score,@errors);
  }));
```

As you can see, list checkers are more complicated than checkers for the other types of data. But for certain situations like the one above, they can be indispensable.

Set and Union Answer Checkers

Since Sets and Unions are unordered lists of Reals or Intervals, they act a lot like lists. If you provide a `checker` for one of these, it works like the List checker: it is applied to the individual entries in the Set or the individual intervals in a Union. You may want to write a checker that works with the entire Set or Union at once, however. In that case, you need to use a `list_checker` in which you obtain the full Set or Union from `$ansHash`. For example:

```
Context("Interval");

BEGIN_TEXT
Find a union of disjoint intervals that properly contains
the numbers 0 and 5:
\{ans_rule(20)\}
END_TEXT

ANS(Union("(-1,1) U (4,infinity)")->cmp(list_checker => sub {
  my ($correct,$student,$ansHash) = @_;
  my $U = $ansHash->{student_value}; # the complete student answer as MathObject
  if ($U->type ne "Union") {return (0,"Your answer should be a union of intervals")}
  my $n = scalar(@$student); # number of intervals
  my $S = Set(0,5);
  return ($U->contains($S) && $U != $S ? $n : 0);  # number of intervals correct
}));
```

Note that the return value is the score followed by any error messages, and that the score is the number of intervals in the student's answer that are correct, which is why this checker returns `$n` when the answer is correct.

More error checking might be desired, here. For example, since a Union can be of Intervals or Sets, we might want to check that all the student's entries in the Union are actually intervals (the current version allows Sets). Note thta because the `studentsMustReduceUnions` parameter is `1` by default, we don't have to check for disjoint intervals. On the other hand, that flag also requires intervals to be merged into one if they could be, e.g., `(0,1] U (1,2)`. We might want to work harder to allow that, since it *is* a disjoint union. We could also give more error messages to help the student work through a wrong answer. For example, we could check which point isn't covered and report that.

Files Defining MathObjects

Files defining MathObjects

The MathObjects library is contained in two Perl packages: the Parser package and the Value package. The first of these implements the parser that converts student answers (or strings passed to `Compute()` or the MathObject constructor functions) into the corresponding MathObjects. The second implements the mathematical operations and functions for the various object types. The files that define the Parser package are pg/lib/Parser.pm [1] and the files in the directory pg/lib/Parser/ [2], while the Value package is defined in pg/lib/Value.pm [3] and the directory pg/lib/Value/ [4]. There are also two macro files pg/macros/Parser.pl [5] and pg/macros/Value.pl [6] that define the object constructors and other values needed by PG problems that use MathObjects. The pg/macros/MathObjects.pl [7] file is a wrapper that loads these two, and is the file that you need to include in your `loadMacros()` call in order to use MathObjects in a problem you are writing.

The pg/lib/Value/ [4] directory includes files for each of the MathObject types (e.g., Real.pm [8], Complex.pm [9], etc.). These define the various object types and how they operate. For example, these implement how operations like addition and multiplication work for each object class, how functions like `sin()` and `sqrt()` work, how to compare two objects for equality and numeric order, how to display the object in TeX and string form, how to covert the object to Perl code, and so on.

The pg/lib/Parser/ [2] directory contains files that make it possible to break up a string into its various tokens and map them to build the parse tree for the expression. For example, there are binary-operator and unary-operator classes that form the nodes of an expression tree. There are also function classes to handle function calls, list classes to handle points, vectors, and so on, and classes for the various values that can be produced, like numbers and strings. The Context stores information that the parser uses to create the parse tree, so the various entities like variables, constants, strings, operators, functions, parens, and so on that you can set in the Context correspond directly to the classes defined in pg/lib/Parser/ [2]. The Parser classes are related to the ones defined in the Value package, but are not the same. The former are used to build the parse tree, while that latter are used to compute specific instances of the objects.

References

[1] https://github.com/openwebwork/pg/blob/master/lib/Parser.pm
[2] https://github.com/openwebwork/pg/tree/master/lib/Parser
[3] https://github.com/openwebwork/pg/blob/master/lib/Value.pm
[4] https://github.com/openwebwork/pg/tree/master/lib/Value
[5] https://github.com/openwebwork/pg/blob/master/macros/Parser.pl
[6] https://github.com/openwebwork/pg/blob/master/macros/Value.pl
[7] https://github.com/openwebwork/pg/blob/master/macros/MathObjects.pl
[8] https://github.com/openwebwork/pg/blob/master/lib/Value/Real.pm
[9] https://github.com/openwebwork/pg/blob/master/lib/Value/Complex.pm

Formula (MathObject Class)

Formula Class

A Formula object represents an expression whose result is one of the MathObject types defined above. These act like functions in that you can evaluate them at different values of the variables that they use, or substitute other expressions for the variables (to form compositions).

Construction

```
$f = Formula("2x^2+3x-5");

$a = random(2,5,1);
$b = random(5,9,1);
$f = Formula("x^2 + $a x + $b");
```

The variables used in a Formula must be declared in the Context. The Numeric Context has `x` pre-defined, but you can add more, as in the following example:

```
Context("Numeric");
Context()->variables->add(y => 'Real');

$f = Formula("x^2 + 2xy + y^2");
```

Operations on Formulas

Formulas can be added, subtracted, multiplied, etc, to obtain new Formula objects, and functions like `sin()`, `sqrt()`, and so on will return Formula objects when their arguments are formulas:

```
$x = Formula("x");
$f = 3* $x**2 - 2 * $x + 5;    # same as Formula("3x^2 - 2x + 5");
$g = $x - 5;
$h = $f / $g;                  # same as Formula("(3x^2 - 2x + 5) / (x - 5)");
$g1 = sin($g)                  # same as Formula("sin(x-5)");
```

Formulas can produce any type of MathObject as its result, including points, vectors, intervals, etc. For example,

```
Context("Vector");
$f1 = Formula("<2x+1,1-x,x^2>");          # a vector-valued formula

Context()->variables->add(t => 'Real');
$f2 = Formula("(1,3,-2) + t <4,-1,2>");    # a parametric line

Context("Interval");
$f3 = Formula("(x,2x+1]");                 # an interval-valued formula
```

Reducing Formulas

If you substitute values as coefficients in a formula, you may end up with things like `1 x^2 + -3 x + 0`, but Formulas have a `reduce` method that can be used to remove coefficients of 1, simplify addition of negatives, remove sums or products of 0, and so forth. E.g.,

```
$f = Formula("1 x^2 + -3 x + 0")->reduce;  # same as Formula("x^2 - 3x");
```

There are a number of reduction rules, and you can enable or disable them individually (**reference needed**). Note, however, that MathObjects is not a full computer algebra system, and the reduction rules are mainly geared toward improving the output, not solving equations or performing algebraic manipulations to simplify the expression.

Evaluating Formulas

Given a Formula, you may want to evaluate the Formula at a particular value of its variable; that can be accomplished by the `eval()` method. The arguments to `eval()` assign values to all the variables of the Formula, and it returns the value of the Formula for those inputs. For example:

```
$f = Formula("x^2 + 2x + 1");
$a = $f->eval(x=>2);             # f at x=2, or Real(9)

$g = Formula("xy + x + y");
$b = $g->eval(x=>2,y=>3);        # g at (x,y) = (2,3), or Real(11);
```

The `substitute()` method is similar to `eval()`, but in this case, the specified variables are replaced by the values that are given. The values need not be constants; they could be Formulas, in which case the result is the composition of the two Formulas. Note that not all the variables need to be substituted (unlike with `eval()`, where all variables must be given a value).

```
$f = Formula("x^2 + 2x + 1");
$g1 = $f->substitute(x => "y");           # same as Formula("y^2 + 2y + 1");
$g2 = $f->substitute(x => "2x-1");        # same as Formula("(2x-1)^2 + 2(2x-1) + 1");

$h = Formula("xy + x + y");
$h1 = $h->substitute(x => "y");           # same as Formula("y*y + y + y") or Formula("y^2 + 2y");
$h2 = $h->substitute(x => 2, y => "3x");  # same as Formula("2(3x) + 2 + 3x");
```

Perl Code from Formulas

If you need to evaluate a Formula multiple times (for example, to produce a graph), it is not very efficient to use `eval()`. Instead, you can use the `perlFunction` method to generate a native Perl function that you can call to evaluate the Formula. If given no arguments, `perlFunction()` returns an anonymous code reference to a subroutine that evaluates the Formula; the parameters to the subroutine are the values of the variables (in alphabetical order). If a single argument is given, it is the name to give to the subroutine; if two arguments are given, the second is an array reference that lists the order of the variables to use for the arguments to the subroutine.

```
$f = Formula("x^2+y");

$F = $f->perlFunction;            # anonymous subroutine reference
$a = &{$F}(2,3);                  # value of $f for x=2 and y=3, i.e., Real(7)
```

```
$f->perlFunction("F");              # subroutine named F
$a = F(2,3);                        # again, Real(y)
```

```
$f->perlFunciton("G",["y","x"]);  # change order of variables in arguments to G
$a = $G(2,3);                     # value of $f for y=2 and x=3, i.e.,  Real(11);
```

Derivatives of Formulas

You can obtain the derivative of a Formula using the `D()` method. If the Formula has more than one variable, then you need to indicate which one to differentiate by; you do this by giving the variable name as an argument to the method. You can get second or third derivatives (or higher) either by using `D()` a second time or third time, or by including more variable names in the call to `D()`. For example:

```
$f = Formula("3x^2-5x+2");
$df = $f->D('x');                    # same as Formula("3*(2x)-5");
$df_4 = $df->eval(x=>4);             # f'(4), i.e., Real(19);
$ddf = $df->D('x');                  # same as Formula("6");
$ddf = $f->D('x','x');               # same as above

$f = Formula("x^2 + 4xy^2 + y^3");
$fx = $f->D('x');                    # same as Formula("2x+4y^2");
$fxy = $fx->D('y');                  # same as Formula("4*(2y)");
$fxy = $f->D('x','y');               # same as above
```

Answer Checker

As with all MathObjects, you obtain an answer checker for a Formula object via the `cmp()` method:

```
ANS(Compute("x^2+2x+1")->cmp);
```

The options for the Formula answer checker is dependent on the type of the result of the Formula. They include all the options for the class of the result, and the following additional options:

Option	Description	Default
`upToConstant => 1` or `0`	For Real-valued formulas, this controls whether the student's answer only needs to match the correct answer up to addition of a constant.	`0`
`showDomainErrors => 1` or `0`	If the test points for the function comparison reveals that the student answer is defined on a different domain from the correct answer (e.g., the student's answer is not defined at one of the test points), then this value determines whether or not a message is issued alerting the student to that fact.	`1`

Methods

The Formula class supports the Common MathObject Methods, and the following additional methods:

Method	Description
`$f->eval(x => value,...)`	Returns the value of `$f` evaluated at the given values of its variables. All variables used in the Formula must be given a value (use `substitute` if you want to set only some of the variables). See the Evaluating Formulas above for examples.
`$f->substitute(x => value,...)`	Returns a copy of `$f` with the given values replaced by their given values. The values can be any appropriate MathObject, including other Formula objects. If the value is a perl string, it will be parsed to form a MathObject. See the Evaluating Formulas above for examples.
`$f->reduce` `$f->reduce(rule => 0 or 1,...)`	Returns a copy of `$f` where the MathObject reduction rules have been applied. You can turn on or off individual rules or collections of rules by giving the name of the rule and setting it to `0` (don't apply) or 1 (apply). See Reducing Formulas above for examples. See the Reduction rules for MathObject Formulas for a list of the reduction rules.

`$f->perlFunction`
`$f->perlFunction("name")`
`$f->perlFunction("name",["x",...])`
	Turns `$f` into executable Perl code that can be called to evaluate the Formula efficiently. The first form returns an anonymous code reference, the second makes a names Perl subroutine, and the third specifies the order of the variables in the subroutine's argument list (the default is alphabetical). See Perl Code from Formulas above for examples.

`$f->isConstant`	Returns `true` if `$f` contains no variables (i.e., is constant-valued), and `undef` otherwise.
`$f->usesOneOf("x",...)`	Returns `1` if any of the variables listed appear in `$f`, and `0` otherwise.

There are additional methods for a Formula that are for internal use during the parsing of the formula. These are in the `pg/lib/Parser.pm` file, and are not documented here.

Properties

The Formula class supports the Common MathObject Properties, and the following additional ones:

Property	Description	Default

`$f->{limits}`	A reference to an array of values that represent the lower and upper limits for the variables used in `$f` when producing random points for comparisons. If there is more than one variable, then either the limits apply to all of them, or the array should contain references to an array of limits for each variable (in alphabetical order). For example
	`$f = Formula("sqrt(x-10)")->with(limits=>[10,12]);`
	`$f = Formula("e^x-y")->with(limits=>[[-1,1],[-2,-1]]);`
	If not set, it will be taken from the answer checker options, or from the Context. The Context default is `[-2,2]`.
		`undef`

`$f->{test_points}`	A reference to an array of values to use for the variables in `$f` when comparing `$f` to another Formula. The value in the array are themselves references to arrays whose values are the values for the variables in `$f` listed in alphabetical order. For example,
	`$f = Formula("ln(|x|)")->with(test_points =>`
	`[[-3],[-2],[-1],[1],[2],[3]]);`
	sets `$f` so that both positive and negative values are used; this means `$f` will not match `Formula("ln(x)")`, for instance.
	If not set, it will be created automatically from random points when `$f` is first compared to another Formula. If set, no random points will be used.
		`undef`

`$f->{test_at}`	A reference to an array of values to use for test points for `$f` in *addition* to the randomly chosen ones. The format is the same as for `test_points` above. E.g., `$f = Formula("ln(|x|)")->with(test_at => [[-1],[1]]);` will cause `$f` to use $x = -1$ and $x = 1$ along with the usual random points when it is compared to another Formula.	`undef`

`$f->{test_values}`	A reference to an array of values produced by `$f` at the `$f->{test_points}`. You should not set this yourself; it is generated automatically when `$f` is compared to another Formula.	produced from `$f`
`$f->{test_adapt}`	When `$f` includes adaptive parameters, this is a reference to the array of values produced by the adapted formula at the test points (where the adaptive parameters have their adapted values). You should not set this yourself; it is generated automatically when `$f` is compared to another Formula.	produced from `$f`
`$f->{parameters}`	When `$f` includes adaptive parameters, this is a reference to the array of the adapted values for those parameters (in alphabetical order). You should not set this yourself; it is generated automatically when `$f` is compared to another Formula.	produced automatically
`$f->{variables}`	A reference to a hash whose keys are the names of the variables used in `$f` (and whose values are 1). You should not set this yourself; it is maintained automatically.	produced automatically
`$f->{values}`	A reference to a hash whose keys are the names of variables whose values are being set during an `eval()` or `substitute()` call, and whose values are the values to which each variable is being set. You should not set this yourself; it is maintained automatically by the `eval()` and `substitute()` methods, but if you implement your own parser objects, you may need to read these values to perform the requires substitutions.	produced automatically
`$f->{f}`	An anonymous code reference used to evaluate `$f` during comparisons to other Formulas. You should not set this value yourself; it is generated automatically when `$f` is compared to another Formula.	`$f->perlFunction`
`$f->{autoFormula}`	When set, this indicates that `$f` was generated implicitly rather than explicitly via `Formula()` or `Compute()` (e.g., by adding two Formulas together, or calling a function like `sin()` on a Formula). You should not set this value yourself; it is managed automatically.	`undef`
`$f->{domainMismatch}`	When `$f` is compared to another Formula, this is set to `1` if the other Formula can't be computed at all the test points for `$f`, and `0` if it can be. You should not set this value yourself; it is generated automatically when `$f` is compared to another Formula.	produced automatically
`$f->{tree}`	This is a reference to the root node of the parse tree for the Formula. The tree is made up of nodes that represent things like binary operators, function calls, variable references, and so on. The various types of nodes are implemented by the files in the pg/lib/Parser [2] directory, with `pg/lib/Parser/Item.pm` being the base class from which others are subclassed.	produced automatically
`$f->{string}`	The original string from which the Formula was generated, when it was created by parsing a string (e.g., via `Compute()` or `Formula()`).	`undef`
`$f->{tokens}`	A reference to the array of "tokens" produced by parsing the string for `$f`. These are themselves references to arrays of four or five elements: the first is a string indicating the type of token (e.g., `"var"`, `"num"`, `"fn"`, etc.), the second is a string representing the instance of that type for this token (e.g., `"x"`, `3.14`, `"sin"`, etc.), and the next two are the starting and ending positions of the token in the original string (used for syntax error highlighting); if present, the fifth is `true` when there are spaces preceding the given token in the string (used for experimental non-standard operator precedences).	produced automatically

How to list Context flags

There is an explanation of the purpose of many of these flags at Context flags.

Listing Context Flags using PGLabs

You can obtain a complete list of contextFlags from one of the current contexts using the PGLabs and this code snippet:

```
Context("Vector");
TEXT(Context()->{name}, $BR);
TEXT(pretty_print(Context()->{flags}));
```

You could also load in one of the customized contexts such as `contextCurrency.pl`

```
loadMacros("contextCurrency.pl");
Context("Currency");
Context()->flags->set(trimTrailingZeros=>1);
TEXT(Context()->{name}, $BR);
TEXT(pretty_print(Context()->{flags}));
```

which will produce the output below. Notice that the trimTrailingZeros flag has been set to 1 (the default is 0).

Currency

HASH(0xc8c1988)

allowBadFunctionInputs	=>	0
allowBadOperands	=>	0
allowEmptyStrings	=>	1
allowMissingFunctionInputs	=>	0
allowMissingOperands	=>	0
allowWrongArgCount	=>	0
checkUndefinedPoints	=>	0
forceCommas	=>	0
forceDecimals	=>	0
formatStudentAnswer	=>	evaluated
granularity	=>	1000
ignoreEndpointTypes	=>	0
ijk	=>	0
ijkAnyDimension	=>	1
infiniteWord	=>	infinity
limits	=>	(-2 , 2 ,)
max_adapt	=>	100000000
max_undefined	=>	
noExtraDecimals	=>	1
num_points	=>	5

promoteReals	=>	1
reduceConstantFunctions	=>	1
reduceConstants	=>	1
reduceSets	=>	1
reduceSetsForComparison	=>	1
reduceUnions	=>	1
reduceUnionsForComparison	=>	1
resolution	=>	
showExtraParens	=>	1
tolType	=>	absolute
tolerance	=>	0.005
trimTrailingZeros	=>	1
useBaseTenLog	=>	0
useFuzzyReals	=>	1
zeroLevel	=>	1e-14
zeroLevelTol	=>	1e-12

Listing Context Flags in an existing WeBWorK question

- Set your Context, e.g.

```
Context("Your Context Here");  #  or don't do anything to see flags from the current context
```

- Enter the text

```
TEXT(pretty_print(Context()->{flags}));
```

- View the problem

Infinity (MathObject Class)

Infinity Class

The Infinity class handles the positive infinity of the extended reals. This value can be negated and used in intervals, but can be added to Reals (or other MathObjects) and can't be used as an argument to functions like `sin()` or `sqrt()`. Infinity can be created in any Context, though `Numeric` context is commonly used.

Creation

In Perl code, you can use `Infinity` to obtain this value, or `-(Infinity)` to obtain its negation. E.g.,

```
Context("Numeric");

$Inf = Infinity;
$MInf = -(Infinity);

Context("Interval");
$I = Interval("(",0,Infinity,"]");   # but easier as Interval("(0,infinity]");
```

Answer Checker

As with all MathObjects, you obtain an answer checker for an Infinity object via the `cmp()` method:

```
ANS(Infinity->cmp);
ANS((-(Infinity))->cmp);
```

The Infinity class supports the common answer-checker options. There are no additional options.

Methods

The Infinity class supports the Common MathObject Methods. There are no additional methods for this class.

Properties

The Infinity class supports the Common MathObject Properties, and the following additional ones:

Property	Description
`$r->{isNegative}`	True when the object represents `-Infinity`, false otherwise. You should not set this value yourself.

Interval (MathObject Class)

Interval Class

The Interval class implements intervals on the real line. They can be open or closed at each end, and can be infinite. For example, `(0,infinity)` is the set of $x$where $x > 0$and `[-1,1]` is the set of $x$where $-1 \le x \le 1$. The interval `(-infinity,infinity)` is the entire real line (and the constant `R` refers to this set in the Interval Context). The individual point $a$on the line can be represented as `[a,a]`, but this is better handled via a Set (i.e., `{a}`). Intervals are most often created in the `Interval` Context.

The answer checker for Intervals can give hints about whether each endpoint is correct or not, and about whether the type of endpoint (open/closed) is correct or not. These features can be controlled through flags for the answer checker.

Construction

Intervals are created via the `Interval()` function, or by `Compute()`.

```
Context("Interval");

$I = Interval(0,1);            # the open interval (0,1)
$I = Interval([0,1]);          # the closed interval [0,1]

$I = Compute("(0,1)");
$I = Compute("[0,1]");

$I = Interval("[",0,Infinity,")");    # half-open the hard way
$I = Interval("[0,infinity)");        # the easy way
$I = Compute("[0,infinity)");
```

Note that you can produce open and closed Intervals using the constructor function without explicit delimiters, but for half-open Intervals, you need to supply the delimiters; it is easiest to do that using a single string that will be parsed rather than the delimiters and endpoints individually.

Operations on Intervals

The union of two Intervals is represent by an upper-case `U` in student answers and parsed strings, and by addition or the dot operator or the `Union()` constructor in Perl code. Differences of intervals can be obtained via subtraction. Intervals can be combined with Sets or Unions in these ways as well.

```
$I1 = Interval("(-infinity,-1]");
$I2 = Interval("[1,infinity)");

$U = $I1 + $I2;
$U = $I1 . $I2;
$U = Union($I1,$I2);
$U = Union("(-infinity,-1] U [1,infinity)");
$U = Compute("(-infinity,-1] U [1,infinity)");

$S = Interval("(-infinity,1]") - Interval("[-1,1)");   # same as Compute("(-infinity,-1) U {1}");
$S = Compute("(-infinity,1] - [-1,1)");                # same as above
```

```
$S = Compute("R - (-1,1)");                          # same as $U above
```

Intersections of Intervals (or Sets or Unions) can be obtained via the `intersect()` method of an Interval. There is no built-in method for students to form intersections (though one could be added to the Context by hand). There are other methods for determining if one Interval is contained in another, or intersects another, or is a subset of another, etc. These methods can be applied to Sets and Unions in addition to Intervals.

```
$I1 = Interval("(-infinity,1]");
$I2 = Interval("(-1,5]");

$I3 = $I1->intersect($I2);                 # same as Interval("(-1,1]");

$I1->contains($I2);                        # returns false
$I3->isSubsetOf($I2);                      # returns true
$I1->intersects($I2);                      # returns true
```

When intervals are compared, they must match not only the endpoints, but also the type of endpoint (open or closed), though an answer checker option can be used to change that for student answers. This condition can be relaxed, however, by setting the `ignoreEndpointTypes` flag in the Context:

```
Context()->flags->set(ignoreEndpointTypes => 1);
```

This will mean that the open/closed type of each endpoint will be ignored, and so intervals only have to match the numbers at the endpoints, not the type.

Answer Checker

As with all MathObjects, you obtain an answer checker for an Interval object via the `cmp()` method:

```
ANS(Compute("[1,infinity)")->cmp);
```

The Interval class supports the common answer-checker options, and the following additional options:

Option	Description	Default
`showEndpointHints => 1` or `0`	Do/don't show messages about which endpoints are correct.	`1`
`showEndTypeHints => 1` or `0`	Do/don't show messages about whether the open/closed status of the end-points are correct (only shown when the end-points themselves are correct).	`1`
`requireParenMatch => 1` or `0`	Do/don't require that the open/closed status of the end-points be correct.	`1`

Methods

The Interval class supports the Common MathObject Methods, and the following additional ones:

Method	Description
`$I1->intersect($I2)`	Returns the intersection of `$I1` with `$I2`. Note that `$I2` can be an Interval, Set, or Union.
`$I1->intersects($I2)`	Returns `true` if `$I1` intersects `$I2`, and `undef` otherwise. Note that `$I2` can be an Interval, Set, or Union.
`$I1->contains($I2)`	Returns `true` if `$I2` is a subset of `$I1`, and `undef` otherwise. Note that `$I2` can be an Interval, Set, or Union.
`$I1->isSubsetOf($I2)`	Returns `true` if `$I1` is a subset of `$I2`, and `undef` otherwise. Note that `$I2` can be an Interval, Set, or Union.

In addition to these, the following are defined so that Intervals have the same methods as Unions and Sets; that way, you can call these on a student's input that might be an Interval, Set, or Union, without having to check the type of object first.

Method	Description
`$I->isEmpty`	Returns `0`.
`$I->isReduced`	Returns `1`.
`$I->reduce`	Returns `$I` itself.
`$I->sort`	Returns `$I` itself.

Properties

The Interval class supports the Common MathObject Properties, and the following additional ones:

Property	Description	Default
`$r->{open}`	`(` for an open left-hand endpoint, `[` for closed.	given when created
`$r->{close}`	`)` for an open right-hand endpoint, `]` for closed.	given when created

Introduction to Contexts

What is a Context?

Every webwork problem has a collection of variables and constants that are part of the problem, and a setting in which these reside. For example, some problems use vector-valued formulas, and others might be about intervals on the real line. Some questions allow students to perform numerical calculations as part of their answers, while others require that they simplify their answers to a single number before entering it. In some questions a less-than and greater-than signs might be used to describe intervals (e.g., `x < 5`) while in others they are used to delimit vectors (as in `<3,-1,2>`). Similarly, in a problem on vectors, the letter `i` might represent the coordinate unit vector along the x-axis, while in a question on complex numbers, `i` might represent the complex number $\sqrt{-1}$.

All of these features represent the *context* of the problem, and MathObjects maintains this information in a special Context object. The Context object (or simply Context) holds information about what variables are defined and their types and domains, what constants are available, what functions can be applied and what operations performed, what words (like `NONE` and `DNE`) are available, and what interpretation to give to the various symbols and delimiters that the student can type.

There are predefined contexts for working with things like real numbers, complex numbers, intervals, vectors, matrices, and so on. There are also a large number of extensions that implement specialized Contexts for particular purposes: scientific notation, currency values, inequalities, factions, polynomials, chemical reactions, and so forth. It is possible to modify an existing Context to suit the needs of your problems, and you can even create your own custom Contexts (though this is an advanced topic).

Selecting a Context

Each Context has a name, and you select a context by adding the command

```
Context("name");
```

at the top of a problem file, where `name` is the name of the desired Context. The pre-defined contexts are listed on the Common Contexts page, and these include `Numeric`, `Complex`, `Vector`, `Matrix`, and `Interval`.

There are a number of Contexts implemented as extensions to the MathObjects library. These are in the `pg/macros/` [1] directory; files whose names begin with `context` implement specialized Contexts. Most of these are listed in the Specialized contexts documentation. In order to use one of these additional Contexts, include the macro file for that Context in the `loadMacros()` call at the top of your file. For example:

```
loadMacros(
  "PGstandard.pl",
  "MathObjects.pl",
  "contextLimitedPolynomial.pl",
  "PGcourse.pl",
);
```

loads the `contextLimitedPolynomial.pl` macro file, which implements the `LimitedPolynomial` Context, which requires students to specify only "simplified" polynomials in the form $a_n x^n + ... + a_0$as their answers. Inspecting files like `contextLimitedPolynomial.pl` is one way to learn how to modify a context to meet your requirements, though that is an advanced topic.

Changing Context Settings

Since the Context stores information about things like the variables and constants for the problem, you need a way to specify what these are. You do this by using methods of the Context object. In order to obtain a reference to the current Context, use the `Contex()` function with no context name; this returns the context that is active.

```
$context = Context();
```

The Context give you access to the various types of information that it stores via methods for each type, such as `Context()->variables` is a reference to an object that holds the information about the variables in the Context, while `Context()->constants` is a reference to the object that manages the constants in the Context. Each of these in turn has methods for adding, modifying, removing, and listing the values that they store. For example,

```
Context()->variables->add(t => 'Real');
```

adds a new variable `t` that is a Real value.

All the types of data listed below have the following methods:

Method	Description
`add(name => data)`	for adding new items
`are(name => data)`	for removing all current items and adding new ones
`set(name => data)`	for modifying existing items
`get(name)`	for getting the data defining a particular item
`replace(name => data)`	for replacing an existing definition by a new one
`undefine(name)`	for making an item be undefined, but still recognized
`redefine(name)`	for making the item be defined again
`remove(name)`	for deleting a particular item
`clear`	for removing all current definitions
`names`	for retrieving the names of all the defined items
`all`	for retrieving the entire collection of data for all items

These are explained in more detail in the Modifying contexts (advanced) documentation. Several of these are used in the examples below, and can serve as an introduction to their use.

In addition to direct calls to the data objects, it is possible to modify the Context in other ways. For example, some function calls modify the Context, e.g.,

```
Parser::Number::NoDecimals(Context());
```

Also, some macro files provide mechanisms for changing the context, as in the following,

```
loadMacros("contextLimitedPowers.pl");
Context()->operators->set(@LimitedPowers::OnlyIntegers);
```

which sets the operators to limit exponents to only integer values.

Variables

Each Context comes with one or more pre-defined variables. For example the `Numeric` Context includes the variable x, while the `Complex` Context has the variable z. The `Point` and `Vector` Contexts have variables x, $y$and z.

To add a new variable to a context, use

```
Context()->variables->add(name => type);
```

where `name` is the name of the variable to be added, and `type` is the kind of MathObject that is to be stored. The valid types are `"Real"`, `"Complex"`, `"Point2D"`, `"Point3D"`, `"Vector2D"`, `"Vector3D"`, `"Parameter"`, or an instance of a MathObject of the appropriate type. You can also add multiple variables at once. For example,

```
Context()->variables->add(y => "Real");
Context()->variables->add(v => Vector(1,2,3,4));
Context()->variables->add(u => "Complex", w => "Complex");
```

adds a real variable named `y`, a variable `v` holding a four-dimensional vector, and two complex variables, `u` and `w`.

You can specify the domain of a variable using the `set()` method, as in the following,

```
Context()->variables->set(x => {limits => [3,5]});
```

which sets the limits for $x$to be from 3 to 5. Again, multiple variables can be set at once.

You can set both the type and the limits at once when you add the variable, as in

```
Context()->variables->add(t => ["Real",limits => [0,1]]);
```

If the type of a variable is `"Parameter"`, then this variable is used as an *adaptive parameter*, meaning WeBWorK will try to pick a value of the variable so that the student's answer matches the professor's answer with that value set. Note that student's can't enter adaptive parameters of their own; only the correct answer can include them. Also note that adaptive parameters only work in a linear setting, e.g., $af + b$, where $a$and $b$are the adaptive parameters, and $f$is the basic correct formula. That is, you can adapt for linear multiples, and constant addition. Once the formulas have been compared, the adapted value for the parameter can be obtained using the `value()` method described below.

A variable can be removed via the `remove()` method:

```
Context()->variables->remove("t");
```

Finally, the names of all the variables in the Context can be displayed by placing

```
TEXT(join(',',Context()->variables->names));
```

into the body of a PG problem.

In addition to the methods shown above, the `variables` object has the following methods:

Method	Description
`Context()->variables->type("name")`	Returns the type of the variable (this is a typeRef hash; see **reference needed**).
`Context()->variables->value("name")`	Returns the value of an adaptive parameter, once it has been used in a Formula comparison.
`Context()->variables->parameters`	Returns an array of the names of all the adaptive parameters in the Context.
`Context()->variables->variables`	Returns an array of the names of all the variables (i.e., non-adaptive-parameters) in the Context.

Constants

The values of variables like π and e are stored in the `constants` data of the Context. The values of `pi` and `e` are defined in all the main Contexts. The `Vector` and `Matrix` Contexts include `i`, `j`, and `k`, which represent the coordinate unit vectors, while the `Complex` Context has `i` defined as the imaginary number $i = \sqrt{-1}$. The `Interval` context has a constant `R` which is equal to the whole real line -- the interval `(-infinity,infinity)`.

Constants are added to the Context via the `add()` method. The data is the value for the constant, and you can set several constants at once. For example:

```
Context()->constants->add(tau => Real(pi/2));
Context()->constants->add(n => Real(3), m => Real(2));
```

Usually when a Formula containing a constant is parsed, the name of the constant is retained, so `Formula("sin(pi)")` would remain as `sin(pi)` rather than `0`. You can use the `set()` method to set the `keepName` parameter for a constant to change this so that the value is substituted into the Formula when it is used. E.g.,

```
Context()->constants->set(pi => {keepName => 0});
$f = Formula("sin(pi)");
```

would be the same as having set `$f = Formula("0")`.

A constant can be removed via the `remove()` method:

```
Context()->constants->remove("pi");
```

Finally, the names of all the constant in the Context can be displayed by placing

```
TEXT(join(',',Context()->constants->names));
```

into the body of a PG problem.

In addition to the methods shown above, the `constants` object has the following methods:

Method	Description
`Context()->variables->value("name")`	Returns the value of the constant.

Strings

Most Contexts include some special words that the students can type, like `NONE` or `DNE` (for "does not exist"). These can be used in problems that ask for lists that might be empty (where `NONE` can be used in that case), or for limit problems where the limit might not exist. These two words are available in all the pre-defined contexts, and also the word `infinity`, which generates the Infinity object, with `inf` as an alias for the complete word. In general, words like these are not case-sensitive, so `INF` would produce `infinity`, as would `InfINIty` or any other capitalization of the word.

You can add your own words to the Context via the `add()` method. Usually, there is no data, so you simply use `{}` as the definition. To make a string case sensitive, use `{caseSensitive =>1}` as the data. To make a word create infinity, use `{intinite => 1}`. To make one word mean the same as another, use `{alias => "name"}`. For example,

```
Context()->strings->add(A => {}, B => {}, C => {}, D => {});
Context()->strings->add(True => {}, False => {}, T => {alias => "True"}, F => {alias => "False"});
Context()->strings->add(WeBWorK => {caseSensitive => 1});
Context()->strings->add(unbounded => {infinite => 1});
```

Here, `A`, `B`, `C`, and `D` are now valid answers for a student to type, but they could also be `a`, `b`, `c`, or `d`. `True` and `False` (with any capitalization) are now available as well, along with `T` as an alternative to `True` and `F` as an alternative for `False`. The word `WeBWorK` can be entered, but only with this capitalization (as it should be). Finally, `unbounded` will produce the positive Infinity object (so `-unbounded` would be the negative Infinity object).

A string can be removed via the `remove()` method:

```
Context()->strings->remove("DNE");
```

Finally, the names of all the constant in the Context can be displayed by placing

```
TEXT(join(',',Context()->strings->names));
```

into the body of a PG problem.

Flags

Many settings for the Context are stored as *flags*, which are parameters that control special functions of the parsing process, the answer checkers, or other features of the MathObjects library. For example, the default tolerance for numeric comparisons is stored as a flag, as are the default limits for variable ranges. Other values include things like whether to display Vectors using ijk-notation rather than coordinate form with angle-bracket delimiters, and how to show student answers in the "Entered" and "Preview" columns of the results table when they submit their answers.

You do not usually add new flags (though you could if you wanted to keep track of information for your own custom MathObject classes). The most common actions are to set or get the values of the flags. To set a flag, use the `set()` with the data being the name of the flag to set and its new value. As with the previous case, you can set several flags at once.

```
Context()->flags->set(tolerance => .005);
Context()->flags->set(
   reduceConstants => 0,
   reduceConstantFunctions => 0,
);
Context()->flags->set(formatStudentAnswers=>"parsed");
```

To get the value of a flag, you could use the `get` method, but since this is such a common operation, there is a shorthand. For example,

```
$tol = Context()->flag("tolerance");
```

gets the current tolerance value from the Context.

Note that MathObject and answer checkers often can override the settings of the Context itself. That means you may need to check the properties of an MathObject and the properties of the active answer checker (if there is one) before resorting to the Context's value. MathObjects gives you an easy way to do that, however, with the `getFlag()` method of a MathObject. For example, if you are writing a custom answer checker, you could use

```
ANS($mo->cmp(checker => sub {
  my ($correct,$student,$ansHash) = @_;
  my $tol = $correct->getFlag("tolerance",.001);
  ... (use $tol here) ...
  return $score;
}));
```

to obtain the value of the `tolerance`. This will be taken from `$correct` if it was set there, otherwise from the current answer checker's flags as passed to `$mo->cmp()`, and then from the Context for `$mo`. If the flag is not set in any of those, the value `.001` is used.

The names of all the constants in the Context can be displayed by placing

```
TEXT(join(',',Context()->flags->names));
```

into the body of a PG problem. Note that different contexts may have different flags, and that some flags that could be set may not currently have any value in the Context. Most of the important flags are described in the Context flags documentation.

Functions

The Context includes information about the functions that are available. For example, the `Complex` Context has `Re()` and `Im()` functions, as well as `arg()`, `mod()`, and `conj()`, and the `Vector` Context has `norm()` and `unit()`. These are not available in the `Numeric` context.

In some problems, you may want to remove some functions so that students can't enter them. For example, if you want to have the student evaluate the sine function at a particular value, you would not want her to be able to use `sin()` in her answer or that would defeat the purpose. To remove a function, use the `disable()` method. For instance,

```
Context()->functions->disable("sin");
```

makes the `sin()` function unavailable to the student. (Note: the function is still recognized by MathObjects, but the student will be told it is not available in this problem if it is used. The `remove()` method would remove the function entirely, making it unknown to MathObjects, so the error message would be less useful to the student.)

To make the function available again, use `enable`, e.g.

```
Context()->functions->enable("sin");
```

You can disable or enable more than one function at a time by listing their names separated by commas. There are also *categories* of functions that you can disable or enable all at once. A full list is available in Answer Checkers and the Context. Some of the common ones are `Trig` to disable all trigonometric functions, `Numeric` to disable things like `ln()` and `sqrt()`, `Complex` to disable the complex functions, and `All` to disable *all* the functions.

For example,

```
Context()->functions->disable("All");
Context()->functions->enable("sqrt");
```

would disable all functions and allow only the square root function.

Note that if you disable the `sqrt()` function, you may want to disable the exponentiation operators so that students can't use `a^(1/2)` or `a**.5` to produce square roots. Similarly, if you disable `abs()`, you would want to remove the absolute value vertical bars so that students can't use `|a|`. See Answer Checkers and the Context for details.

It is possible to add new functions in several ways. The `pg/macros/parserFunctions.pl` [2] file implements an easy way to add functions to a Context using Formula objects. Functions written in Perl code can be added to the context using the techniques outlined in the second example on the AddingFunctions page. Alternatively, functions defined using Perl code can be added to the Context, as described in the Adding New Functions documentation.

The names of all the functions in the Context can be displayed by placing

```
TEXT(join(',',Context()->functions->names));
```

into the body of a PG problem.

In addition to the methods listed above, the `functions` object has the following:

Method	Description
`Context()->functions->disable("name")`	Marks the named function(s) or category of functions so that they can't be used in student answers.
`Context()->functions->enable("name")`	Re-enabled the named function(s) or category of functions so that they *can* be used in student answers.

Operators

The operators that are allowed within a student's answer are controlled by the Context. All the pre-defined Contexts include the standard operators like `+` and `-` for addition and subtraction, `*` and `/` for multiplication and division, `^` or `**` for exponentiation, `!` for factorial, and `,` for forming lists. Some contexts include `U` for taking unions, `.` for taking dot products, and `><` for taking cross products.

In some problems, you may want to remove some operators so that students can't enter them. For example, if you want to have the student compute the value of an expression, you would not want her to be able to include the operations from that expression in her answer. To remove a function, use the `undefine()` method. For instance,

```
Context()->operators->undefine("^","**");
```

makes exponentiation unavailable to the student. (Note: the operations still are recognized by MathObjects, but the student will be told they are not available in this problem if they are used. The `remove()` method would remove the operators entirely, making them unknown to MathObjects, so the error message would be less useful to the student.)

To make the operator available again, use `redefine`, e.g.

```
Context()->operators->redefine("^","**");
```

Note that multiplication and division have several forms (in order to make a non-standard precedence that allows things like `sin(2x)` to be entered as `sin 2x`). So if you want to disable them you need to include all of them.

E.g.,

```
Context()->operators->undefine('*',' *','* ');
Context()->operators->undefine('/',' /','/ ','//');
```

would be required in order to make multiplication and division unavailable.

The names of all the operators in the Context can be displayed by placing

```
TEXT(join(',',Context()->operators->names));
```

into the body of a PG problem.

The pg/macros/ [1] directory contains a number of predefined contexts that limit the operations that can be performed in a student answer. For example, the `contextLimitedNumeric.pl` file defines contexts in which students can enter numbers, but no operations, so they would have to reduce their answer to a single number by hand. There are limited contexts for complex numbers, points, and vectors, and there are also specialized contexts for entering polynomials, or where powers are restricted in various ways.

Reduction Rules

When random numbers are used as coefficients in Formulas, it is possible to get situations like `1 x^2 + -3 x + 0`, and it looks bad to have the coefficient of 1, the `+ -`, and the `+ 0` as part of the formula. For that reason, the Formula class has a `reduce()` method that will normalize the Formula to remove such issues. The reduction rules are controlled through the Context's `reduction` object. This lists the various reduction rules and determines which ones are active. You can turn these on and off using the `set()` method, as in

```
Context()->reductions->set("(-x)-y" => 0);
```

to turn off the reduction that converts `(-x)-y` to `-(x+y)`. As usual, you can set the values for multiple rules in one `set()` call.

Because the `set()` call is rather verbose, there are shorthands for turning reduction rules on and off via the `reduce()` and `noreduce()` methods. For example,

```
Context()->noreduce("(-x)-y");
```

is equivalent to the `set()` above. Again, you can supply multiple rules at once:

```
Context()->noreduce("(-x)-y","(-x)+y");
```

These commands set the reduction rules globally, so they affect all reductions that follow. It is also possible, however, to temporarily suspend certain reduction rules during the reduction process for a specific Formula, as in the following.

```
$f->reduce("(-x)-y" => 0, "(-x)+y" => 0);
```

which turns off the reduction rules only for this one reduction.

There are two reductions that occur during the parsing of any formula: the first is that if an operation occurs between two constants, the operation is performed and the result is put into the Formula instead; the second is that if a function call is made on a constant value, the result of the function call is used instead. For example,

```
$a = 2; $b = 3;
$f = Formula("$a * $b * x + 4 * $a");
$g = Formula("abs(-$b)");
```

would produce the formula `6 * x + 8` rather than `2 * 2 * x + 4 * 2` for `$f`, and `3` rather than `abs(-3)` for `$g`.

These operations are controlled by two Context flags (not reduction rules, since they apply to all parsing, not just `reduce()` calls). These are the `reduceConstants` and `reduceConstantFunctions` flags. You can unset these to prevent those reductions from occurring automatically, as in

```
Context()->flags->set(
  reduceConstants => 0,
  reduceConstantFunctions => 0,
);
$f = Formula("(1+sqrt(5))/2)");
```

in which case `$f` would remain `(1 + sqrt(5))/2` rather than the usual `1.61803`.

The names of all the reduction rules in the Context can be displayed by placing

```
TEXT(join(',',Context()->reductions->names));
```

into the body of a PG problem. There is also an annotated available on the Reduction rules for MathObject Formulas page.

See Also

- Common Contexts
- Context flags
- Specialized contexts
- Modifying contexts (advanced)

References

[1] https://github.com/openwebwork/pg/tree/master/macros
[2] http://webwork.maa.org/pod/pg_TRUNK/macros/parserFunction.pl.html

Introduction to MathObjects

What are MathObjects?

MathObjects are a set of Perl objects that make the manipulation of mathematics within WeBWorK problems more intuitive. They make it possible to define variables in your problems that correspond to common mathematical objects, such as formulas, real number, complex numbers, intervals, vectors, points, and so on. For example:

```
$f = Formula("sin(x^2+6)");
$a = Real("sqrt(pi/6)");
$z = Complex("1 + 5i");
```

These are useful (and powerful) because MathObjects "know" information about themselves; thus, you can add formulas to get new formulas, plug real objects into formulas to get formulas evaluated at those values, calculate derivatives of formulas, add and subtract intervals to form unions and intersections, and much more.

For several reasons it is usually preferable to write the MathObjects above using the `Compute()` function rather than the individual constructors for the various MathObject types.

```
$f = Compute("sin(x^2+6)");
$a = Compute("sqrt(pi/6)");
$z = Compute("1 + 5i");
```

The `Compute()` function determines the kind of MathObject from the Context and from the syntax of its argument, which is usually a string value that is in the form that a student could type. The `Compute` function also sets the "correct answer" to be the exact string that it was given, so that if a student were asked to enter a number that matched `$a` from above and asked to see the correct answer (after the due date), then `sqrt(pi/6)` would be displayed rather than `0.723601`. This gives you more control over the format of correct answers that are shown to students.

Why use MathObjects?

MathObjects are designed to be used in two ways. First, you can use them within your Perl code when writing problems as a means of making it easier to handle formulas and other mathematical items, and in particular to be able to produce numeric values, TeX output, and answer strings from a single formula object. This avoids having to type a function three different ways (as a Perl function, as a TeX string, and as an answer string), making it much easier to maintain a problem, or duplicate and modify it to use a different formula. Since MathObjects also includes vectors, points, matrices, intervals, complex numbers, and a variety of other object types, it is easier to work with these kinds of values as well.

More importantly, using MathObjects improves the processing of student input. This is accomplished through special answer checkers that are part of the MathObjects Parser package rather than the traditional WeBWorK answer checkers. Each of these checkers has error checking customized to the type of input expected from the student and can provide helpful feedback if the syntax of the student's entry is incorrect. Because the MathObject checkers are part of a unified library, students get consistent error messages regardless of the type of answer that is expected or that they provide. For example, if a student enters a formula in an answer blank where the correct answer is actually a number, she will receive a message indicating that what she typed was a formula but that a number was expected. Thus students are given guidance automatically about this type of semantic problem with their answers. Answer checkers for points and vectors can indicate that the number of coordinates are wrong, or can tell the student which coordinates are incorrect (at the problem-author's discretion).

MathObject Contexts

Although a problem may include several answer blanks, the problem generally has a collection of variables that it defines, values that it uses, and so on; these form the "context" of the problem. For example, if the problem is concerned with a function of $x$and $y$and its partial derivatives, then the context of the problem includes variables x and y, and so all the answer blanks within the problem should recognize that `x` and `y` have meaning within the problem, even if the answer blank is only asking for a number. Suppose a student is asked to enter the value of a partial derivative at a particular point and, rather than giving a number, enters the formula for the derivative (not evaluated at the point). He should not be told "x is undefined", as would have been the case with WeBWorK's traditional answer checkers, since $x$actually *is* defined as part of the problem. Such messages serve to confuse the student rather than help him resolve the problem.

MathObjects solves this issue by using a Context object to maintain the context of a problem. That way, all the answer blanks will know about all the variables and values defined within the problem, and can issue appropriate warning messages when a student uses them inappropriately. For example, in the situation described above where the student entered the unevaluated derivative where the value at a point was requested, he will get the message "Your answer is not a number (it seems to be a formula returning a number)", which should help him figure out what he has done wrong.

Thus one of the main purposes for the Context is to maintain the set of variables and values that are part of the problem as a whole. Another key function of the Context is to tell the answer checkers what functions and operations are available within a student answer, and what the various symbols the student can type will mean. For example, if you are asking a student to compute the value of 6 / 3, then you may want to restrict what the student can type so that she can't enter `/` in her answer, and must type an actual number (not an expression that is evaluated to a number). Or if you are asking a student to determine the value of `sin(pi/6)`, you might want to restrict his answer so that he can't include `sin()`, but you do allow arithmetic operations. Such restrictions are also part of the context of the problem, and so are maintained by MathObjects as part of the Context.

In a similar way, some symbols mean different things in different problems. For example in a problem dealing with intervals, `(4,5)` means the interval between 4 and 5, while in a multi-variable calculus setting, it might mean a single point in the xy-plane. Or in a problem on complex numbers, the value $i$means $\sqrt{-1}$, while in vector calculus, it would mean the coordinate unit vector along the x-axis. Or in a problem on inequalities, `5 < x` would use `<` as the "less-than" sign, while in a vector calculus problem, `<5,6,7>` would mean the vector with coordinates 5, 6, and 7. The Context determines how these symbols will be interpreted by the MathObjects library.

The MathObjects library comes with a collection of predefined Contexts that you can call on to set the meanings of symbols like these to be what you need for your problems. The default Context is the `Numeric` Context, which is sufficient for most first-semester calculus problems. There are Contexts for complex numbers, intervals, vectors, and so forth; see the Common Contexts list and the links at the bottom of this document for further information. The pg/macros/ [1] directory includes a number of specialized Contexts as well; the files beginning with `context` define these, and are described on the Specialized contexts page.

Use `$context = Context();` to obtain a reference to the current context. See the Introduction to contexts for information about how to use this to modify an existing context (e.g., to add variables to it, or to restrict the functions that can be used).

It is possible to use more than one context within the same problem. This is discussed **(link needed)**.

Further Reading

- Introduction to Contexts
- Common Contexts
- Specialized contexts
- pg/macros [1] -- files starting with `context` define new Contexts

How to create a MathObject

In order to use MathObjects in a problem you are writing, include `MathObjects.pl` in your `loadMacros()` call. For example:

```
loadMacros(
  "PGstandard.pl",
  "MathObjects.pl",
  "PGcourse.pl",
);
```

Once this is done, there are several ways to create MathObjects in your problem:

- By calling a constructor function for the type of object you want (e.g., `Real(3.5)` or `Complex("3+4i")`)
- By calling `Compute()` to parse a string and return the resulting object (e.g., `Compute("3+4i")`)
- By calling a method of an existing MathObject that returns another object (e.g., `Formula("sin(x)")->eval(x => pi/2)`)
- By combining existing MathObjects via mathematical operations (e.g., `$x = Formula("x"); $f = $x**2 + 2*$x + 1`)

Some examples:

```
$a = Real(3.5);   or
$a = Compute(3.5);   or
$a = Compute("3.5");
```

```
$b = Complex(3, 4);   or
$b = Complex("3 + 4i");   or
$b = Compute("3 + 4i");
```

```
$v = Vector(4,5,8);   or
$v = Vector([4,5,8]);   or
$v = Vector("<4,5,8>");  or
$v = Compute("<4,5,8>");
```

```
$f = Formula("sin(x^2) + 6");  or
$f = Compute("sin(x^2) + 6");
```

Here, `$a` (defined any of the three ways) represents a real number and `$b` (defined by any of the three) represents a complex number; `$v` will be a vector, and `$f` is a formula.

In general, the `Compute()` variant is preferred because you enter what you want the student to type, and `Compute()` generates the proper MathObject to handle that (just as it will when it processes the student answer). Another important feature of `Compute()` is that the input string also serves as the correct answer when a student requests answers after the due date, which lets you put the correct answer in exactly the form you want it to appear. For example if you use

```
Compute("<cos(pi/6),sin(pi/6), pi/6>")->cmp
```

the correct answer will be presented to the student as `<cos(pi/6),sin(pi/6), pi/6>`, while

```
Vector("<cos(pi/6),sin(pi/6), pi/6>")->cmp
```

would show `<0.866025,0.5,0.523599>` as the correct answer.

In addition, `Compute()` can be used to compute the result of a formula at a given value of its inputs. For example,

```
$a = Compute("x+3",x => 2);
$a = Compute("x^2+y^2",x => 1, y => 2);
```

both are equivalent to `$a = Real(5).`

The MathObjects library defines classes for a variety of common mathematical concepts, including real numbers, complex numbers, intervals, points, vectors, matrices, and formulas. These are described in more detail in the MathObject Class documentation.

The pg/macros [1] directory contains a number of extensions to MathObjects, including files that define specialized MathObject types. The ones that begin with `parser` typically define a new object class. For example, `parserParametricLine.pl` defines a constructor `ParametricLine()` for creating a special object that checks if a student's answer is a given parametric line or not, even if it is parameterized differently. These files usually contain documentation within them; see the POD pages [1] for versions that you can read on line, and Specialized parsers for a list of most of the MathObject extensions.

Further Reading

- MathObject Class documentation
- Specialized parsers
- pg/macros [1] -- files starting with `parser` define new MathObejcts

How to display a MathObject

MathObject can be inserted into the text of your problem just as any other variable can be. When this is done, the MathObject will insert its string version (as generated by its `string()` method). Frequently, however, you want to include the MathObject within a mathematical expression, and would prefer to use its form (as generated by its `TeX()` method). One way to do this is to call the `TeX()` method directly, as in

```
$f = Formula("(x+1)/(x-1)");
BEGIN_TEXT
  \[f(x) = \{$f->TeX\}\]
END_TEXT
```

but this requires `\{...\}` and the explicit use of `->TeX` to get the desired output. There is an easier way, however, that involves telling the Context to produce format rather than string format via the `texStrings()` method of the Context object.

```
$f = Formula("(x+1)/(x-1)");
Context()->texStrings;
BEGIN_TEXT
  \[f(x) = $f\]
END_TEXT
Context()->normalStrings;
```

In this case, you just use `$f` within the text of the problem, which is easier to type and easier to read, especially if you have several MathObjects to insert. It is good practice to return the setting to normal using `normalStrings()`, as is done here, so that if you use MathObjects withing strings later on (e.g., `$g = Compute("3*$f + 1")`), you will not be inserting into those strings inadvertently.

MathObject Answer Checkers

Each MathObject type has a corresponding answer checker that can be used to process student responses. You obtain the checker by using the `cmp()` method of the MathObject.

```
ANS($mathObject->cmp);
```

There is no need to select the right kind of answer checker, as the MathObject library handles that for you.

Many of the answer checkers include options that you can specify. For example, the tolerances to use, or the limits to use for the variables, or whether to provide certain kinds of hints to students as they enter incorrect answers. These are passed as `name=>value` to the `cmp()` method, separated by commas if there is more than one option to set, as in the following example:

```
ANS(Real(sqrt(2))->cmp(tolerance => .0001));          # requires answer with roughly four decimal places.

ANS(Formula("sqrt(x-10)")->cmp(limits => [10,12]));   # use x in the interval [10,12] so that the square root is defined.
```

See the Answer Checkers in MathObjects page for details about the MathObject Answer checkers, and the Answer Checker Options page for more on the options you can set for the different MathObject classes.

Further Reading

- Answer Checkers in MathObjects
- Answer Checker Options
- Custom Answer Checkers

MathObject Methods and Properties

Since MathObjects are Perl objects, you call a method on a MathObject as you would a method for any Perl object, using the `->` operator with the MathObject on the left and the method name and its arguments (if any) on the right. E.g.,

```
$mathObject->method;               # when there are no arguments
$mathObject->method($arg);         # for one argument
$mathObject->method($arg1,$arg2); # for two arguments
# and so on
```

Similarly, you access the properties of an object just as you would for a Perl hash, via the `->` operator with the MathObject on the left and the property name in braces on the right. E.g.,

```
$mathObject->{property-name}
```

There are a number of methods and properties that are common to all MathObjects, which are described in the documentation for Common MathObject Methods and Common MathObject Properties. Some classes have additional methods and properties, and many of these are described in the MathObject Class documentation under the individual class links.

Further Reading

- Common MathObject Methods
- Common MathObject Properties
- MathObject Classes

Experimenting with MathObejcts

One way to experiment with MathObjects is to use the on-line PG labs to write example code and see what it produces. You can copy any of the examples above (or elsewhere in the WIki) and paste them into the lab to check the results. This makes it easy to test code without having to write complete problem files and save them in a course. One lab lets you get the output from a line of PG code, while another lets you try out full problems to see how they work.

List (MathObject Class)

List Class

The List class implements arbitrary lists of other MathObjects. There is no restriction on what can be placed in a list, including other lists, other than what can be specified in the Context. For example, `1,<2,3>,5-2i,DNE` is a valid List. Lists can have repeated items, as in `1,1,0,3,0,1`. Lists can be enclosed in delimiters, e.g., `(1,DNE)`, and you can have lists of lists, as in `(1,increases),(3.5,decreases)`. If you include delimiters in a list, then students must include them as well; if you leave them off, students must also leave them off in their answers. The empty list is represented by `()`, though you may prefer to have student enter a value like `NONE` instead. Lists can be created in all the main Contexts.

The answer checker for lists allows you to specify if the order of the list matters or not, and students can enter their answers in any order by default. Lists are useful when you don't want to give away the number of answers a student may have to come up with, for example, or when you don't want to have to worry about the order of the answers. There are flags for the answer checker that you can use to change how students must enter delimiters.

Construction

Lists are created via the `List()` function, or by `Compute()`. Lists can be combined via addition or by the dot operator; these both concatenate two lists. For example:

```
$L = List(1,3+i,"DNE",Vector(3,2,1));
$L = List([1,3+i,"DNE",Vector(3,2,1)]);
$L = List("1,3+i,DNE,<3,2,1>");        # provided i and Vectors are in the context
$L = Compute("1,3+i,DNE,<3,2,1>");     # same caveat

$L1 = List(1,2,3);  $L2 = List(4,5,6);
$L3 = $L1 + $L2;                       # same as List(1,2,3,4,5,6)
$L3 = $L1 . $L2;                       # same as above
```

If you create a list via the `List()` constructor that is passed a list of items rather than a string (as in the first two examples above), then this assumes no delimiters are needed. The third and fourth examples explicitly indicate that no delimiters are needed.

Answer Checker

As with all MathObjects, you obtain an answer checker for a List object via the `cmp()` method:

```
ANS(List("2,pi,sqrt(2)")->cmp);
```

The List class supports the common answer-checker options, and the following additional options:

Option	Description	Default
`showHints => 1` or `0`	Do/don't show messages about which entries are incorrect.	`$showPartialCorrectAnswers`
`showLengthHints => 1` or `0`	Do/don't show messages about having the correct number of entries (only shown when all the student answers are correct but there are more needed, or all the correct answsers are among the ones given, but some extras were given).	`$showPartialCorrectAnswers`
`showParenHints => 1` or `0`	Do/don't show messages about having the correct type of delimiters, or missing delimiters.	`$showPartialCorrectAnswers`
`partialCredit => 1` or `0`	Do/don't give partial credit for when some answers are right, but not all.	`$showPartialCorrectAnswers`
`ordered => 1` or `0`	Give credit only if the student answers are in the same order as the professor's answers.	`0`
`entry_type => "a (name)"`	The string to use in error messages that identifies the type of elements that make up the list.	determined from entries
`list_type => "a (name)"`	The string to use in error messages that identifies the list itself.	determined from list
`extra => $object`	A MathObject to use for comparison to student-provided entries that do not match any of the correct answers in the list. Use this in particular when the correct answer is a list of a single String (like `NONE`), and the expected type of elements in the list are complicated and need special syntax checking (e.g., for size of Vectors).	determined from entries
`typeMatch => $object`	Specifies the type of object that the student should be allowed to enter in the list (determines what constitutes a type mismatch error). Can be either a MathObject or a string that is the class of a MathObject (e.g., `"Value::Vector"`).	determined from entries
`requireParenMatch => 1` or `0`	Do/don't require the parentheses in the student's answer to match those in the professor's answer exactly.	`1`
`removeParens => 1` or `0`	Do/don't remove the parentheses from the professor's list as part of the correct answer string. This is so that if you use List() to create the list (which doesn't allow you to control the parens directly), you can still get a list with no parentheses.	`1`
`implicitList => 1` or `0`	Force/don't force single entry answers to be lists (even if they are already lists). This way, if you are asking for a list of lists, and the student enters a single list, it will be turned into a list of a single list, and will be checked properly against the correct answer.	`1`

Methods

The List class supports the Common MathObject Methods. There are no additional methods for this class.

Properties

The List class supports the Common MathObject Properties, and the following additional ones:

Property	Description	Default
`$r->{open}`	The symbol to use for the open parenthesis	`undef`
`$r->{close}`	The symbol to use for the close parenthesis	`undef`

Matrix (MathObject Class)

Matrix Class

The Matrix class implements matrices (in any dimension) of Real entries, though the most common use is 2-dimensional ones (i.e., $n \times m$matrices). A 1-dimensional Matrix is effectively a Vector. Typically, the rows of a Matrix are comma-separated lists of numbers delimited by square brackets, and a Matrix is a comma separated list of rows delimited by square brackets. E.g., `1,2,3],[4,5,6` is a 2×3matrix whose first row has entries 1, 2 and 3, and whose second row has entries 4, 5 and 6. Matrices can be formed in the `Matrix` Context. It is possible to have Complex entries in a Matrix, but there is not a pre-defined Context that makes that easy to do.

The answer checker for Matrices can give students hints about whether the dimension of the matrix is correct. It is also possible to have an array of answer blanks rather than having the student type the entire matrix in one long answer blank.

Construction

Matrices are created via the `Matrix()` function, or by `Compute()`.

```
Context("Matrix");

$M = Matrix([1,2,3],[4,5,6]);
$M = Matrix(1,2,3],[4,5,6);
$M = Matrix("1,2,3],[4,5,6");
$M = Compute("1,2,3],[4,5,6");
```

Higher-dimensional Matrices can be obtained as a comma-separated list of lower-dimensional matrices delimited by square brackets. For example,

```
$M = Matrix([1,2],[3,4,5,6],[7,8])
```

produces a $2 \times 2 \times 2$matrix (a 3-dimensional matrix).

The identity matrix for any dimension can be obtained from `Value::Matrix->I(n)` where *n* is the dimension of the matrix. For example

```
$I = Value::Matrix->I(2);
```

is a 2×2identity matrix.

Operations on Matrices

Matrices can be combined by addition, subtraction, and multiplication, when the dimensions match up properly. Square matrices can have integer powers taken of them (meaning repeated multiplication). You can perform scalar multiplication and division, and also matrix-vector multiplication.

```
$M1 = Matrix([1,2],[3,4]);
$M2 = Matrix([5,6],[7,8]);
$v = Vector(9,10);

$M3 = $M1 + $M2;      # same as Matrix([6,8],[10,12]);
$A = $M1 * $M2;       # usual matrix multiplication
$B = $M1 ** 2;        # same as $M1 * $M1
$C = 3 * $M1;         # same as Matrix([3,6],[9,12]);
$D = $M1 / 2;         # same as Matrix([.5,1],[1.5,2]);

$u = $M1 * $v         # matrix-vector multiplication
```

The value of `$u` above will be a 2×1matrix, which is effectively a column vector but it is not a Vector object, it is a Matrix object. You can force it to be a Vector object using `$u = Vector($M1 * $v);` or `$u = ColumnVector($m1 * $v)` instead.

Matrices have a number of methods that can be used to compute things like determinants, inverses, transposes, and so on; these are listed in the Methods section below. Some examples are:

```
$a = $M1->det;
$A = $M1->transpose;
$B = $M1->inverse;
$t = $M1->trace;
$v = $M1->row(1);         # the first row of the matrix
$v = $M1->column(1);      # the first column of the matrix
$a = $M1->element(2,1);   # the element in the second row, first column
```

Note that `$M1->row(1)` and `$M1->column(1)` produce Matrix objects, not Vectors. You can coerce them to Vectors via `Vector($M1->row(1))` and `ColumnVector($M1->column(1))`.

Answer Checker

As with all MathObjects, you obtain an answer checker for a Matrix object via the `cmp()` method:

```
ANS(Matrix(4,0,-2],[1,1,5)->cmp);
```

The Matrix class supports the common answer-checker options, and the following additional options:

Option	Description	Default
`showDimensionHints => 1` or `0`	Show/don't show messages about the wrong number of rows or columns.	`1`

If you use a single answer blank for a Matrix answer checker, this means the student must type the entire Matrix, including the brackets and commas, all on one line. This allows the student to enter Matrix-values expressions (like sums or multiplications), but it is hard to type, and difficult to see the Matrix as an array when it is presented linearly in this way. It is more common, therefore, to use the Matrix object's `ans_array()` method rather than PG's `ans_rule()` function to obtain a separate answer box for each entry. The Matrix answer checker will manage this collection of answer boxes for you, so your code doesn't have to change in any other way.

```
Context("Matrix");

$M = Matrix(2,4,1],[-1,0,3);

Context()->texStrings;
BEGIN_TEXT
\($M\) = \{$M->ans_array\}
END_TEXT
Context()->normalStrings;

ANS($M->cmp);
```

If you want students to enter Matrix-valued expressions, you can add Matrix-valued constants to the Context of your problem and have them write answers as matrix expressions in terms of those constants. For example

```
Context("Numeric");        # Numeric prevents students from entering Matrices or Vectors directly


Context()->constants->add(
   M => Matrix([pi**2,1/pi],[sqrt(2),ln(pi)]),    # an arbitrary matrix with no special properties or symmetries
   I => Matrix(1,0],[0,1),                    # identity matrix
);
Context()->flags->set(
  formatStudentAnswer => 'parsed',               # leave student's answer as a formula
);


$f = Formula("M^2 + 3M - I");                    # a Matrix formula


Context()->texStrings;
BEGIN_TEXT
\($f\) = \{ans_rule(20)\}
END_TEXT
Context()->normalStrings;


ANS($f->cmp);
```

Methods

The Matrix class supports the Common MathObject Methods, and the following additional methods:

Method	Description
`$M->det`	Returns the determinant of `$M` as a Real.
`$M->inverse`	Returns the inverse of `$M`. Note that `$M` must be square.
`$M->transpose`	Returns the transpose of `$M`.
`$M->trace`	Returns the trace of `$M`. Note that `$M` must be square.
`$M->proj`	Returns ???
`$M->proj_coeff`	Returns ???
`$M->row(`*n*`)`	Returns the n-th row of `$M` as a 1-dimensional Matrix. To obtain it as a Vector, use `Vector($M->row(n))`.
`$M->column(`*n*`)`	Returns the n-th column of `$M` as a 1-dimensional Matrix. To obtain it as a Vector, use `Vector($M->column(n))` or `ColumnVector($m->column(n))`.
`$M->element(`*i*`,`*j*`)`	Returns the (i,j)-th element of `$M` (i.e., the element in row i and column j) as a Real.
`$M->dimensions`	Returns an array of number of rows and columns of `$M` if it is 2-dimensional. (If 1-dimensional, it is the number of elements in `$M`, if 3-dimensional, it is three number giving the sizes of the three dimension, and so on.)
`$M->isSquare`	Returns 1 if the matrix is $n \times n$ and 0 otherwise.
`$M->isRow`	Returns 1 if the matrix is 1-dimensional, 0 otherwise.
`$M->is_symmetric`	Returns 1 if the matrix is symmetric across the main diagonal, 0 otherwise.
`$M->wwMatrix`	Returns returns an old-style Matrix from the Matrix package. Note that if `$M1` is an old-style Matrix object, `Matrix($M1)` will convert it to a MathObject Matrix.
`$M->norm_one`	Returns the one-norm of `$M` (the maximum of the sums of the columns).
`$M->norm_max`	Returns the maximum-norm of `$M` (the maximum of the sums of the rows).
`$M->kleene`	Returns a copy of `$M` where Kleene's algorithm has been applied. Note that `$M` must be square.
`$M->normalize($v)`	Returns a matrix and vector that have been "normalized" to improve numeric stability, but still represent the same system as the original `$M` and `$v`. Note that `$M` must be square.
`$M->decompose_LR`	Returns a matrix that encodes the L-R decomposition of `$M`. Note that `$M` must be square. (L is a lower-triangular matrix and R is an upper-triangular matrix where $LR = M$, but the matrix returned here is a special combination of the two stored in one square matrix, the so-called LR matrix.)
`$M->L`	Returns the L matrix from the L-R decomposition of `$M`.
`$M->R`	Returns the R matrix from the L-R decomposition of `$M`.
`$M->PL`	Returns the left pivot permutation matrix for the L-R decomposition of `$M`. (Note that $PLLRPR = M$.)
`$M->PR`	Returns the right pivot permuation matrix for the L-R decomposition of `$M`. (Note that $PLLRPR = M$.)
`$A->solve_LR($b)`	Solves the system $A\boldsymbol{x} = \boldsymbol{b}$ using an LR matrix and returns an array consisting of the dimension of the solution space, the solution vector (as a Matrix object), and the *base* matrix (see the MatrixReal1 documenation [1] for details).
`$M->condition($Minv)`	Shorthand for `$M->norm_one*$Minv->norm_one`. Note that `$Minv` should be the inverse of `$M`, and both must be square. The number is a measure of the numerical stability of the matrix; the smaller the number, the better stabiity.
`$M->order_LR`	Returns the order of `$M` (the number of linearly independent columns or rows).

`$A->solve_GSM($x0,$b,$epsilon)`	Solves the system $A\boldsymbol{x} = \boldsymbol{b}$using the Global Step Method fixed-point iterative approach. Here, `$x0` is a starting guess solution (that should be near the actual solution), `$b` is the vector $\boldsymbol{b}$, and `$epsilon` is an error tolerance for determining when to stop. See the MatrixReal1 documenation [1] for details.
`$A->solve_SSM($x0,$b,$epsilon)`	Solves the system $A\boldsymbol{x} = \boldsymbol{b}$using the Single Step Method fixed-point iterative approach. Here, `$x0` is a starting guess solution (that should be near the actual solution), `$b` is the vector $\boldsymbol{b}$, and `$epsilon` is an error tolerance for determining when to stop. See the MatrixReal1 documenation [1] for details.
`$A->solve_RM($x0,$b,$weight,$epsilon)`	Solves the system $A\boldsymbol{x} = \boldsymbol{b}$using the Relaxation Method fixed-point iterative approach. Here, `$x0` is a starting guess solution (that should be near the actual solution), `$b` is the vector $\boldsymbol{b}$, `$weight` is a number between 0 and 2, and `$epsilon` is an error tolerance for determining when to stop. See the MatrixReal1 documenation [1] for details.

Properties

The Matrix class supports the Common MathObject Properties, and the following additional ones:

Property	Description	Default
`$r->{open}`	The symbol to use for the open square bracket	`[`
`$r->{close}`	The symbol to use for the close square bracket	`]`

References

[1] http://webwork.maa.org/pod/pg_TRUNK/lib/MatrixReal1.html

Modifying Contexts (advanced)

Advanced Context Modifications

The Introduction to Contexts describes how to make basic modifications to a Context's variables, constants, strings, flags, functions, operators, and reduction rules. Here we will describe more advanced modifications and techniques involving the Context.

Number Formats

Real numbers are stored using a format that retains about 16 or 17 significant digits, making computations very accurate in most situations. When a number is displayed, you probably don't want to see all 17 digits (that would make a vector in three-space take up around 35 characters, for example). To make answers easier to read, MathObjects usually display only 6 significant digits. You can change the format used, however, to suit your needs. The format is determined by the `Context()->{format}{number}`, which is a `printf`-style string indicating how real numbers should be formatted for display.

The format always should begin with `%` and end with one of `f`, `e`, or `g`, possibly followed by `#`. Here, `f` means fixed-point notation (e.g. `452.116`), `e` means exponential notation (e.g, `3.578E-5`), and `g` means use the form most appropriate for the magnitude of the number. Between the `%` and the letter you can (optionally) include `.n` where `n` is the number of decimal digits to use for the number. If the format ends in `#`, then trailing zeros are removed after the number is formatted. (More sophisticated formats are possible, but this describes the basics.)

```
Context()->{format}{number} = "%.2f";    # format numbers using 2-place decimals (e.g., for currency values).
Context()->{format}{number} = "%.4f#";   # format numbers using 4-place decimals, but remove trailing zeros, if any.
```

The default format is `"%g"`.

The Context also includes information about what should count as a number when an answer is parsed. There are two patterns for this, a signed number and an unsigned number. The latter is what is used in parsing numbers (and the sign is treated as unary minus); former is used in the `Value::matchNumber()` function. These are stored in the `Context()->{pattern}` hash; the default values are:

```
Context()->{pattern}{number} = '(?:\d+(?:\.\d*)?|\.\d+)(?:E[-+]?\d+)?';
Context()->{pattern}{signedNumber} = '[-+]?(?:\d+(?:\.\d*)?|\.\d+)(?:E[-+]?\d+)?';
```

These are fairly complicated regular expressions that match the usual fixe-point and exponential notation for numbers in WeBWorK. It is possible to change these patterns to handle things like commas instead of decimals for European usage, or to allow commas every three digits. Note, however, that you would need to include a `NumberCheck` routine that would translate the special format into the required internal format. For example, this allows you to enter numbers as hexadecimal values:

```
#
#  Numbers in hexadecimal
#
Context()->{pattern}{number} = '[0-9A-F]+';
Context()->{pattern}{signedNumber = '[-+]?[0-9A-F]+';
Context()->flags->set(NumberCheck => sub {
  my $self = shift;                              # the Number object
  $self->{value} = hex($self->{value_string});   # convert hex to decimal via perl hex() function
  $self->{isOne} = ($self->{value} == 1);        # set marker indicating if the value is 1
  $self->{isZero} = ($self->{value} == 0);       # set marker indicating if the value is 0
```

```
});
Context()->update;
```

Note that after changing the `pattern` you must call `Context()->update` to remake the tokenization patterns used by the Context.

Here is an example that lets you use commas in your numbers:

```
#
# Allow commas every three digits in numbers
#
Context()->{pattern}{number} = '(:?(:?\d{1,3}(:?\,\d{3})+|\d+)(?:\.\d*)?|\.\d+)(?:E[-+]?\d+)?';
Context()->{pattern}{signedNumber} = '[-+]?(:?(:?\d{1,3}(:?\,\d{3})+|\d+)(?:\.\d*)?|\.\d+)(?:E[-+]?\d+)?';
Context()->flags->set(NumberCheck => sub {
  my $self = shift;                              # the Number object
  my $value = $self->{value_string};             # the original string
  $value =~ s/,//g;                              # remove commas
  $self->{value} = $value + 0;                   # make sure it is converted to a number
  $self->{isOne} = ($self->{value} == 1);        # set marker indicating if the value is 1
  $self->{isZero} = ($self->{value} == 0);       # set marker indicating if the value is 0
});
Context()->update;
```

If you want to make the numbers display with commas, then you will need to subclass the `Value::Real` object and override the `string()` and `TeX()` methods to insert the commas again, and then tie your new class into the `Context()->{value}{Real}` value. For example, in addition to the changes above, you might do

```
#
#  Subclass the Value::Real class and override its string() and TeX()
#  methods to insert commas back into the output
#
package my::Real;
our @ISA = ('Value::Real');    # subclass of this Value::Real

sub string {
  my $self = shift; my $x = $self->SUPER::string(@_);  # get the original string output
  my ($n,@rest) = split(/([.E])/,$x,1);                # break it into the integer part and the rest
  while ($n =~ m/[0-9]{4}(,|$)/)                       # add commas as needed
    {$n =~ s/([0-9])([0-9]{3})(,|$)/$1,$2$3/}
  return join("",$n,@rest);                            # return the final string
}

sub TeX {
  my $self = shift;
  my $n = $self->SUPER::TeX(@_);     # original TeX uses string(), so commas are already there
  $n =~ s/,/{,}/g;                   # just make sure they have the correct spacing
  return $n;
}

package main;    # end of package my::Real;
```

```
Context()->{value}{Real} = "my::Real";    # make the Context use my::Real rather then Value::Real
Context()->{format}{number} = "%f#";      # format using "f" rather than "g", so no exponential notation
```

This could be put into a separate macro file that you could load into your problems whenever it is needed. See Creating Custom Contexts for details.

Lists and Delimiters

The Context object contains two more collections of data that were not mentioned in the Introduction to Contexts: the `lists` and `parens` objects. These are closely related, and determine what types of objects are created from various delimiters like braces and brackets. For example, in some contexts parentheses form Points, while in others they form Intervals or Lists. This it controlled by the settings in these two objects.

The `lists` object contains the definitions for the various types of list-like objects such as Points, Vectors, and Intervals. Each of the types of list has an entry that tells the parser what class implements the list, and specifies the open and close delimiters and the separators that will be used by default to display an instance of the class. A special case is `AbsoluteValue`, which is treated as a list since it has open and close delimiters, even though the list can only contain one element.

List	Open	Close	Separator
Point	(	)	,
Vector	<	>	,
Matrix	[	]	,
List			,
Interval	(	)	,
Set	{	}	,
Union			U
AbsoluteValue	|	|	

This delimiters listed in this table are used when the object doesn't specify the delimiters explicitly via its `{open}` and `{close}` properties. This is usually the case when objects are created via the class constructors rather than parsing a string. For example, `Vector(4,0,-1)` would not have its `{open}` and `{close}` properties set, so would use the defaults in the `lists` object. Note that the Interval object has parentheses as its default delimiters, but the `Interval()` constructor will set the open and close properties automatically so that you can form open and closed intervals easily:

```
$I1 = Interval(1,2);          # an open interval
$I2 = Interval([1,2]);        # a closed interval
$I3 = Interval("(",1,2,"]");  # a half-open interval
```

It is also possible to put the delimiters at the end of the interval (which is how the Interval's `value()` method returns them): `Interval(0,1,"(","]")`.

On the other hand, instances of these objects created by parsing a string usually save the open and closing delimiters in the object's `{open}` and `{close}` properties, so the default will not be used in those cases. E.g., `$v = Compute("<4,0,-1>")` would produce a Vector object with the `$v->{open} = "<"` and `$v->{close} = ">"`.

To change the `list` settings, use the `set()` method, as usual:

```
Context()->lists->set(Vector => {open => "(", close => ")"};
```

Note that this only affects the case where the Vector object doesn't specify the open and close delimiters explicitly. It also doesn't change the delimiters that the parser uses to identify a vector, since the `list` values are only for output.

To change what delimiter to use for a given object type, you need to change the `parens` object. This associates each open delimiter to one of the `list` types given above, and also gives the close delimiter that is needed to match it, and some other data about how it can be used. So to complete the change for Vectors, we would need to use

```
Context()->parens->set("(" => {type => "Vector", close => ")"});
```

The other possible data for a `paren` object includes:

Name	Description
`type => "`*name*`"`	Specifies the list type that this open delimiter will genarate.
`close => "`*c*`"`	The closing delimiter for this opening one.
`removable => 1` or `0`	Do/don't remove delimiters when used around a single element. When `1`, don't create a list, just return the element.
`formInterval => "`*c*`"`	When present, this indicates that an Interval should be formed when the list is closed by the character *c* rather than the usual `close` character.
`formList => 1` or `0`	Do/don't allow this delimiter to form a List when the entries don't work for its `type`, otherwise produce an error.
`formMatrix => 1` or `0`	Do/don't allow this delimiter to form a Matrix when the entries are appropriate for that.
`emptyOK => 1` or `0`	Do/don't allow empty delimiters. When `0` there must be at least one element between the open and close delimiters.
`function => 1` or `0`	Do/don't allow this delimiter to form a function call when preceded by a function name.

More about Variables

The Introduction to Contexts shows how to add variables to a Context. One thing to keep in mind is that most Contexts come with some variables pre-defined, and when you add new ones, the originals are still available. If you wish to have only the variables that you define, then use `are()` rather than `add()` to add the variables. For example,

```
Context("Numeric");
Context()->variables->add(t => "Real");
```

would add a new real variable t, which would be in addition to the $x$that is already in the `Numeric` context, while

```
Context("Numeric");
Context()->variables->are(t => "Real");
```

would remove any pre-defined variables and leave you with only one variable, t.

More about Constants

The Introduction to Contexts shows how to add constants to a Context. Usually, the output for a constant is its name, but you might want to specify a different value, particular for its output. You can set the constant's `TeX`, `string`, and `perl` values to control the output in those formats. For example,

```
Context("Complex");
Context()->constants->set(i => {TeX=>'\boldsymbol{i}', perl=>'i'});
```

would indicate that output should be $\boldsymbol{i}$, while its Perl form should be just `i`. Similarly,

```
Context("Interval");
Context()->constants->set(R => {TeX=>'\mathbb{R}'});
```

would set the `R` constant to produce $\mathbb{R}$rather than in output.

Adding New Functions

The Introduction to Contexts includes some information about adding new functions that can be used in student Answers. One approach is given in `pg/macros/parserFunctions.pl` [2], which implements an easy way to add functions to a Context using Formula objects. But if your function doesn't just compute the result of a Formula, then you would have to implement a Perl-based function. That process is described here.

To make a new function, you must create a subclass of the `Parser::Function` class or one of its subclasses. It is easiest to do the latter, if possible, since these already handle most of the details. The subclasses are the following:

Subclass	Description
`Parser::Function::numeric`	Functions with one real input producing a real result, or one complex input producing a complex result. Output type is determined by input type. Setting `nocomplex=>1` means complex input not allowed.
`Parser:Function::numeric2`	Functions with two real input returning a real result.
`Parser:Function:complex`	Functions with one complex (or real) input returning a real or complex result. Result is real unless `complex=>1` is set.
`Parser::Function::vector`	Functions with one vector input producing a real or vector result. The result is real unless `vector=>1` is set. Should always set `vectorInput=>1`.

To implement your function, make a package that is a subclass of one of these, and give it a subroutine that computes the function you are interested in. Note that the number and type of inputs will already be checked, so there is not need to check that in your subroutine. Give your subroutine the same name as the function. E.g.,

```
package my::Function::numeric;
our @ISA = ('Parser::Function::numeric');     # subclass of Parser::Function::numeric

my $log2 = CORE::log(2);                      # cached value of log(2)

sub log2 {                                    # the routine for computing log base 2
  shift; my $x = shift;
  return CORE::log($x)/$log2;
}

package main;    #  end my::Function::numeric
```

Now we have to hook the new function into the Context:

```
Context("Numeric");
Context()->functions->add(
  log2 => {class => 'my::Function::numeric', TeX => '\log_2', nocomplex => 1},
);
```

This sets the name of the function to `log2` in the student answer, and uses `my::Function::numeric` to implement it. The output is given as $\log_2$, and we are not allowing complex inputs. Once this is done, you can use things like

```
$r = Compute("(1/3)*log2(5)");
...
ANS($r->cmp);
```

and students can use `log2(5)` in their answers.

If you want to be able to use `log2()` directly in your Perl code, then you should also define

```
sub log2 {Parser::Function->call("log2",@_)}
```

in the `main` package. Then you can do

```
$r = (1/3)*log2(5);
```

directly (i.e., without `Compute()`).

It would be possible to put the required definitions into a macro file that you can load into a problem file whenever it is needed; see Creating Custom Contexts for details.

If your function isn't one that can be obtained by subclassing one of the classes listed above, then you will have to subclass `Parser::Function` directly. In this case, you will have to create `_check`, `_eval` and `_call` methods in your class in addition to the subroutine implementing your function. You can model these after the ones in any of the subclasses listed above (which are in the `pg/lib/Parser/Function` [1] directory). Note that the `checkNumeric()` and other service routines are in `pg/lib/Parser/Function.pm` [2].

Adding New Operators

The Introduction to Contexts gives information about controlling the operators that are part of the Context. To add a new operator you need to make a subclass of `Parser::BOP` or `Parser::UOP` depending on wether the new operator is a binary or unary operator. This should implement the `_check()` and `_eval()` methods to check that its operands are correct and to compute its value. It might also need to implement other methods like `TeX()` or `string()` or `perl()`. There are a number of examples in the `pg/lib/Parser/BOP` [3] and `pg/lib/Parser/UOP` [4] directories.

The example given below implements a binary operator that performs "$n$choose r" so that `n # r` means $\binom{n}{r}$.

```
#
#  A package for computing n choose r
#
package my::BOP::choose;
our @ISA = ('Parser::BOP');            # subclass of Binary OPerator

#
#  Check that the operand types are numbers.
#
```

```
sub _check {
  my $self = shift; my $name = $self->{bop};
  return if $self->checkNumbers();             # method inherited from Parser::BOP
  $self->Error("Operands of '%s' must be Numbers",$name);
}

#
#  Compute the value of n choose r.
#
sub _eval {
  shift; my ($n,$r) = @_; my $C = 1;
  $r = $n-$r if ($r > $n-$r);                  # find the smaller of the two
  for (1..$r) {$C = $C*($n-$_+1)/$_}
  return $C
}

#
#  Non-standard TeX output
#
sub TeX {
  my $self = shift;
  return '{'.$self->{lop}->TeX.' \choose '.$self->{rop}->TeX.'}';
}

#
#  Non-standard perl output
#
sub perl {
  my $self = shift;
  return '(my::BOP::choose->_eval('.$self->{lop}->perl.','.$self->{rop}->perl.'))';
}

package main;
```

This defines the new operator, but now we need to add it into the Context.

```
Context("Numeric");

$prec = Context()->operators->get('+')->{precedence} - .25;

Context()->operators->add(
  '#' => {
     class => 'my::BOP::choose',
     precedence => $prec,         #  just below addition
     associativity => 'left',     #  computed left to right
     type => 'bin',               #  binary operator
     string => ' # ',             #  output string for it (default is the operator name with no spaces)
     TeX => '\mathbin{\#}',       #  TeX version (overridden above, but just an example)
```

```
    }
);
```

Now you can do things like

```
$p = Compute("5 # 3");
```

and students can use `#` to perform the same operation in their answers.

You can combine all of this into a macro file for inclusion into any problem that needs it. See Creating Custom Contexts for more information.

References

[1] https://github.com/openwebwork/pg/tree/master/lib/Parser/Function
[2] https://github.com/openwebwork/pg/blob/master/lib/Parser/Function.pm
[3] https://github.com/openwebwork/pg/tree/master/lib/Parser/BOP
[4] https://github.com/openwebwork/pg/tree/master/lib/Parser/UOP

Point (MathObject Class)

Point Class

The Point class implements points in for arbitrary n. Typically, Points are delimited by parentheses, but that can be controlled by settings in the Context. Points are typically used in the `Point` or `Vector` Contexts, though they are also available in `Matrix` Context. It is possible to create Points in , though there is no pre-defined Context that makes this easy to do.

The answer checker for Points can give students hints about the coordinates that are wrong, and about whether the number of coordinates is correct.

Creation

Points are created via the `Point()` function, or by `Compute()`.

```
Context("Point");

$p = Point(3,0,-2);
$p = Point([3,0,-2]);
$p = Point("(3,0,-2)");
$p = Compute("(3,0,-2)");
```

Operations on Points

Points (of the same dimension) can be added to and subtracted from each other. Points can be multiplied and divided by scalars.

```
$q = $p + Point(1,3,7);    # same as Point(4,3,5);
$q = $p + [1,3,7];         # same as above
$q = 3*$p;                 # same as Point(9,0,-6);
$p = $p/2;                 # same as Point(3/2,0,-1);
```

Answer Checker

As with all MathObjects, you obtain an answer checker for a Point object via the `cmp()` method:

```
ANS(Compute("(4,0,-2)")->cmp);
```

The Point class supports the common answer-checker options, and the following additional options:

Option	Description	Default
`showDimensionHints => 1` or `0`	Show/don't show messages about the wrong number of coordinates.	`1`
`showCoordinateHints => 1` or `0`	Show/don't show message about which coordinates are right.	`1`

By default, the Point answer checker asks the student to type the entire point, including the parentheses and commas. This allows the student to enter point-values formulas (like sums of points, or multiples of points). You may want to restrict the operations that are allowed in the student answer, in which case you might consider using the `LimitedPoint` Context available in the `pg/macros/contextLimitedPoint.pl` file; see the POD documentation [1] for details.

Alternatively, you can use the Point's `ans_array()` method rather than PG's `ans_rule()` function to obtain a separate answer box for each coordinate. The Point answer checker will manage this collection of answer boxes for you, so your code doesn't have to change in any other way.

```
Context("Point");

$P = Point(random(1,10,1),random(1,10,1),-1);

Context()->texStrings;
BEGIN_TEXT
\($P\) = \{$P->ans_array\}
END_TEXT
Context()->normalStrings;

ANS($P->cmp);
```

The value of this approach is that it forces the student to enter individual coordinates, without allowing operations on Points.

Methods

The Point class supports the Common MathObject Methods. There are no additional methods for this class.

Properties

The Point class supports the Common MathObject Properties, and the following additional ones:

Property	Description	Default
`$r->{open}`	The symbol to use for the open parenthesis	(
`$r->{close}`	The symbol to use for the close parenthesis	)

References

[1] http://webwork.maa.org/pod/pg_TRUNK/macros/contextLimitedPoint.pl.html

Presentations on MathObjects

In the summer of 2006 at the MathFest in Knoxville Davide Cervone introduced the MathObjects extension to the PG language. This allows you to write code using objects that more closely mimic mathematics constructs as we usually think about them. This code is easier to read and to maintain and is now the preferred way to write problems. (Of course there are still many problems written without using MathObjects, but when creating new problems use a "MathObjects" problem as a model if at all possible. :-)

Notice that the slide show presentations use WeBWorK itself as a presentation tool.

- Introducing Math Objects [1] : Davide Cervone's 2006 talk at Mathfest 2006 in Knoxville, TN.
- More on math objects [2] : Davide's in depth talk on the properties of MathObjects given at the AIM webwork workshop in August 2007 at the American Institute of Mathematics in Palo Alto, CA. Problems 4, 9, 14, 15, 16, 18, 19, 51, 52, 57, 60, 61, 66, 68, 70, 72, 74, 76, 78, 82 display as webwork problems, the rest of the text shows up if you download the hardcopy.

References

[1] https://courses.webwork.maa.org/webwork2/gage_course/Knoxsville_MAA_2006_cervone/?login_practice_user=true
[2] https://courses.webwork.maa.org/webwork2/cervone_course/setAIM-Talk/?login_practice_user=true

Real (MathObject Class)

Real class

The Real class implements real numbers with "fuzzy" comparison (governed by the same tolerances and settings that control student answer checking). For example, `Real(1.0) == Real(1.0000001)` will be true, while `Real(1.0) < Real(1.0000001)` will be false. Reals can be created in any Context, but the `Numeric` Context is the most frequently used for Reals.

Creation

Reals are created via the `Real()` function, or by `Compute()`. Reals can be added, subtracted, and so on, and the results will still be MathObject Reals. Similarly, `sin()`, `sqrt()`, `ln()`, and the other functions return Real objects if their arguments are Reals. For example:

```
Context("Numeric");

$a = Real(2);
$b = $a + 5;      # same as Real(7);
$c = sqrt($a);    # same as Real(sqrt(2));
```

This allows you to compute with Reals just as you would with native Perl real numbers.

Pre-defined Reals

The value `pi` can be used in your Perl code to represent the value of π. Note that you must use `-(pi)` for $-\pi$ in Perl expressions (but not in strings that will be parsed by MathObjects, such as student answers or arguments to `Compute()`). For instance:

```
$a = pi + 2;              # same as Real("pi + 2");
$b = 2 - (pi);            # same as Real("2 - pi");
$c = sin(pi/2);           # same as Real(1);
$d = Compute("2 - pi");   # parens only needed in Perl expressions
```

The value `e`, for the base of the natural log, e, can be used in student answers and parsed strings.

```
$e = Compute("e");
$p = Compute("e^2");
```

Answer Checker

As with all MathObjects, you obtain an answer checker for a Real object via the `cmp()` method:

```
ANS(Real(2)->cmp);
```

The Real class supports the common answer-checker options, and the following additional options:

Option	Description	Default
`ignoreInfinity => 1` or `0`	Do/don't report type mismatches if the student enters an infinity.	`1`

Methods

The Real class supports the Common MathObject Methods. There are no additional methods for this class.

The command

```
Parser::Number::NoDecimals();
```

will add a check so that a number with decimal places is not allowed. This can be used to require students to enter values in terms of `pi` or `sqrt(2)` for example. You can also supply a Context as an argument to disallow decimals in that context:

```
Parser::Number::NoDecimals($context);
```

Without a parameter, the current context is assumed.

Properties

The Real class supports the Common MathObject Properties, and the following additional ones:

Property	Description	Default

`$r->{period}`	When set, this value indicates that the real is periodic, with period given by this value. So angles might use `period` set to `2*pi`.
	Example:
	`$r = Real(pi/3)->with(period => 2*pi);`
	`$r == 7*pi/3; # will be true`
		`undef`

`$r->{logPeriodic}`	When `period` is defined, and `logPeriodic` is set to `1` this indicates that the periodicity is logarithmic (i.e., the period refers to the log of the value, not the value itself).
	Example:
	`$r = Real(4)->with(period => 10, logPeriodic => 1);`
	`$r == 88105; # true since 88105 is nearly exp(10+log(4))`
		`0`

Reduction rules for MathObject Formulas

When substituting randomly chosen values into an expression, it is possible to obtain situations like `1 x^2 + -3 x + 0`, which do not look good when displayed as part of the text shown to a student. To help remedy this situation, the MathObjects library can reduce Formula objects according to a set of reduction rules that try to eliminate such visual problems. The main reduction rules are shown below.

Sometimes these rules don't improve the look of your formulas, however. In particular, many are designed to try to factor out negatives so that they can be canceled, but this does not always lead to better looking equations when the negatives *don't* cancel. For this reason, you can turn of individual reduction rules so that they will be skipped. See the Reduction Rules section of the Introduction to Contexts for details on how to control the reduction rules.

You can disable one or more reduction rules using a command like the following examples:

```
Context()->noreduce('(-x)-y');
Context()->noreduce('(-x)-y','(-x)+y');
```

It is possible for MathObject extensions to add more reduction rules for the MathObjects they define. See the documentation for the extensions you are using to see if that is the case.

The Reduction Rules and their Actions

Rule	Reduction	Description
`0+x`	`x`	Anything plus zero is itself.
`x+0`	`x`	Anything plus zero is itself.
`0-x`	`-x`	Subtracting from zero gives negative.
`x-0`	`x`	Anything minus zero is itself.
`0*x`	`0`	Zero times anything is zero (or zero vector, etc.).
`x*0`	`0`	Zero times anything is zero (or zero vector, etc.).
`1*x`	`x`	One times anything is itself.
`x*1`	`x`	Anything times one is itself.
`x*n`	`n*x`	Move constants to be in front of formulas.
`0/x`	`0`	Zero divided by anything is zero (since division by zero is already not allowed).
`x/1`	`x`	Anything over one is itself.
`x^0`	`1`	Anything to the first power is itself.
`x^(-1)`	`1/x`	Negative power is reciprocal.
`1^x`	`1`	One to any power is one.
`0.x`	`0`	Zero vector dot any vector is zero.
`x.0`	`0`	Any vector dot zero vector is zero.
`0><x`	`0`	Cross product with a zero vector produces a zero vector.
`x><0`	`0`	Cross product with a zero vector produces a zero vector.
`-n`	`-(n)`	Factor out negatives from real number constants.
`-a-bi`	`-(a+bi)`	Factor out negatives from complex constants.
`(-x)+y`	`y-x`	Reverse order to remove initial negative.
`x+(-y)`	`x-y`	Turn plus-a-negative into minus.

`(-x)-y`	`-(x+y)`	Turn negative-minus-positive to negation of addition.
`x-(-y)`	`x+y`	Turn minus-a-negative to addition.
`(-x)*y`	`-(x*y)`	Factor negation out of multiplication.
`x*(-y)`	`-(x*y)`	Factor negation out of multiplication.
`(-x)/y`	`-(x/y)`	Factor negation out of division.
`x/(-y)`	`-(x/y)`	Factor negation out of division.
`(-x).y`	`-(x.y)`	Factor negative out of dot product.
`x.(-y)`	`-(x.y)`	Factor negative out of dot product.
`(-x)><y`	`-(x><y)`	Factor negative out of cross product.
`x><(-y)`	`-(x><y)`	Factor negative out of cross product.
`-(-x)`	`x`	Cancel double negatives.
`+x`	`x`	Remove unneeded unary plus.
`-x=n`	`x=-n`	Multiply equality by $-1$when equal to a constant.
`-x=-y`	`x=y`	Multiply equality by $-1$when both sides are negative.
`fn*x`	`x*fn`	Move function calls to the right.
`V_n`	n-th item of `V`.	Extract a coordinate from a vector or point.

Set (MathObject Class)

Set Class

The Set class implements finite sets of real numbers. Sets are enclosed in curly braces, and can contain arbitrarily many real numbers, in any order. The empty set is formed by open and close braces with no numbers between them, i.e., `{}`. Set are most often created in the `Interval` context.

The answer checker for Sets reports a warning if an element is entered twice in the set (e.g., `{1,1,2}`), but this can be controlled by an option for the answer checker.

Creation

Sets are created via the `Set()` function, or by `Compute()`.

```
Context("Interval");

$S = Set(0,sqrt(2),pi,-7);
$S = Set([0,sqrt(2),pi,-7]);
$S = Set("{0,sqrt(2),pi,-7}");
$S = Compute("{0,sqrt(2),pi,-7}");
```

When Sets are created as part of the code of a problem, repeated elements are removed automatically so that `{0,1,1,2,0}` will be converted to `{0,1,2}`. In general, however, students must enter Sets without repeated elements (though this can be controlled by answer checker options). You can turn off automatic reduction by setting the flag `reduceSets` to 0 in the Context:

```
Context()->flags->set(reduceSets => 0);
```

This will preserve the Set in whatever form it was originally created (this is set to 0 for student answers so that automatic reductions are not performed). A second flag, `reduceSetsForComparison`, determines whether Sets are reduced (temporarily) before they are compared for equality, or whether the structures must match exactly.

Operations on Sets

Sets can be combined with each other and with Intervals via a Union (represented by an upper-case `U` in student answers and parsed strings, and by addition or the `Union()` constructor in Perl code). Differences of Sets and other Sets, Intervals, or Unions can be obtained via subtraction. (Note that in order to ensure that the result of the union operation has only one copy of each element, you need to pass it through the Set constructor)

```
$U = Set(Set(0,1,2) + Set(2,pi,sqrt(2)));     # same as Set(0,1,2,pi,sqrt(2));
$U = Set("{0,1,2} U {2,pi,sqrt(2)}");
$U = Union("{0,1,2} U {2,pi,sqrt(2)}");
$U = Compute("{0,1,2} U {2,pi,sqrt(2)}");
$W = Set(0,1,2) + Interval("(1,2)");          # same as Compute("{0} U [1,2]");

$S = Set(0,1,2) - Set(2,pi);                  # same as Set(0,1);
$S = Compute("{0,1,2} - {2,pi}");             # same as above

$S = Compute("{0,1,2} - [1,2)");              # same as Set(1,2);
```

Intersections of Sets with other Sets, Intervals, or Unions can be obtained via the `intersect()` method of a Set. There is no built-in method for students to form intersections (though one could be added to the Context by hand). There are other methods for determining if one Set is contained in another, or intersects one, or is a subset of another, etc. These methods can be applied to Intervals and Unions in addition to Sets.

```
$S1 = Set(1,2,3,4);
$S2 = Set(3,4,5);

$S3 = $S1->intersect($S2);           # same as Set(3,4);

$S1->contains($S2);                  # returns false
$S3->isSubsetOf($S2);                # returns true
$S1->intersects($S2);                # returns true
```

Answer Checker

As with all MathObjects, you obtain an answer checker for a Set object via the `cmp()` method:

```
ANS(Compute("{-2.3,0,pi}")->cmp);
```

The Set class supports the common answer-checker options, and the following additional options:

Option	Description	Default
`showHints => 1` or `0`	Do/don't show messages about which entries are incorrect.	`$showPartialCorrectAnswers`
`showLengthHints => 1` or `0`	Do/don't show messages about having the correct number of entries (only shown when all the student answers are correct but there are more needed, or all the correct answsers are among the ones given, but some extras were given).	`$showPartialCorrectAnswers`
`partialCredit => 1` or `0`	Do/don't give partial credit for when some answers are right, but not all.	`$showPartialCorrectAnswers`
`ordered => 1` or `0`	Give credit only if the student answers are in the same order as the professor's answers.	`0`

Note that since a Set is a subclass of List, the other options for the List answer checker also can be used here, though their defaults have been set to appropriate values for a Set and it is unlikely you will need to change them.

Option	Description	Default
`showParenHints => 1` or `0`	Do/don't show messages about having the correct type of delimiters, or missing delimiters.	`1`
`entry_type => "a (name)"`	The string to use in error messages that identifies the type of elements that make up the Set.	`"a number"`
`list_type => "a (name)"`	The string to use in error messages that identifies the Set itself.	`"a set"`
`typeMatch => $object`	Specifies the type of object that the student should be allowed to enter in the Set (determines what constitutes a type mismatch error). Can be either a MathObject or a string that is the class of a MathObject (e.g., `"Value::Vector"`).	`"Value::Real"`
`requireParenMatch => 1` or `0`	Do/don't require the parentheses in the student's answer to match those in the professor's answer exactly.	`1`
`removeParens => 1` or `0`	Do/don't remove the parentheses from the professor's list as part of the correct answer string. This is so that if you use List() to create the list (which doesn't allow you to control the parens directly), you can still get a list with no parentheses.	`0`
`implicitList => 1` or `0`	Force/don't force single entry answers to be lists (even if they are already lists).	`0`

Note also that since Set is a subclass of List, if you supply a custom `checker` option, it operates the same as for a List. You probably want to use a `list_checker` instead. See Custom Answer Checkers for Lists for details.

Methods

The Set class supports the Common MathObject Methods, and the following additional ones:

Method	Description
`$S1->intersect($S2)`	Returns the intersection of `$S1` with `$S2`. Note that `$S2` can be an Interval, Set, or Union.
`$S1->intersects($S2)`	Returns `true` if `$S1` intersects `$S2`, and `undef` otherwise. Note that `$S2` can be an Interval, Set, or Union.
`$S1->contains($S2)`	Returns `true` if `$S2` is a subset of `$S1`, and `undef` otherwise. Note that `$S2` can be an Interval, Set, or Union.
`$S1->isSubsetOf($S2)`	Returns `true` if `$S1` is a subset of `$S2`, and `undef` otherwise. Note that `$S2` can be an Interval, Set, or Union.
`$S->isEmpty`	Returns `true` if `$S` is the empty set, and `undef` otherwise.
`$S->isReduced`	Returns `true` if there are no repeated elements in `$S`, and`undef` otherwise.
`$S->reduce`	Returns a copy of `$S` without any repeated elements.
`$S->sort`	Returns a copy of `$S` with its elements sorted in ascending order.

Properties

The Set class supports the Common MathObject Properties, and the following additional ones:

Property	Description	Default
`$r->{open}`	The symbol to use for the open brace	`{`
`$r->{close}`	The symbol to use for the close brace	`}`

Specialized contexts

Specialized Contexts

Using advanced methods one can customize Contexts to control the possible student responses that are allowed and the feedback given to students if their response has incorrect syntax. These customizations can be placed in a macro file for reuse which by convention is given a name starting with "context" -- e.g. `contextYourContextHere.pl`.

- ModifyingContexts(Advanced) contains information on the techniques and commands for tweaking the contexts for your task.
- SpecializedParsers covers the other advanced method for modifying the behavior of MathObjects. These files modify the parser which translates "TI calculator" formulae into MathObjects.
- Examples are provided in the files listed below.

Specialized Context Macro files

Here is a partial list of contexts.

Check the POD documentation for more examples. [1]

Use `loadMacros("contextABCD.pl");` to make the context available for a WeBWorK question.

- contextABCD.pl [1]
  - `Context("ABCD");`
  - `Context("ABCD-List");`
  - Contexts for matching problems. Adds strings A, B, C, D as allowable entries.
- contextCurrency.pl [2]

  - `Context("Currency");`
  - Context for entering numbers with currency symbols and commas.
  - `$m = Currency(10.99);` or `$m2 = Compute('$10,000.00');`
- contextFraction.pl [3]
  - `Context("Fraction");`
  - `Context("Fraction-NoDecimals");`
  - `Context("LimitedFraction");`
  - `Context("LimitedProperFraction");`
  - Implements a MathObject class for Fractions.
- contextInequalities.pl [4]
  - `Context(`Inequalities');`
  - `Context(`Inequalities-Only');`
  - Provides contexts that allow intervals to be specified as inequalities.
- contextIntegerFunctions.pl [5]
  - `Context("IntegerFunctions");`
  - adds integer related functions C(n,r) and P(n,r).
- contextLimitedComplex.pl [6]
  - `Context("LimitedComplex");`
  - Allow complex numbers but not complex operations.
- contextLimitedNumeric.pl [7]
  - `Context("LimitedNumeric-List");` -- allows lists -- (need contextLimiteNumeric.pl for this)
  - ([DEPRECATED] -- loadMacros("contextLimitedNumeric.pl") is not needed for the these remaining contexts which are now standard and defined in `pg/lib/Parser/Legacy/LimitedNumeric.pm`
  - `Context("LimitedNumeric");`
  - `Context("LimitedNumeric-Fraction")`
  - `Context("LimitedNumeric-StrictFraction");` -- no decimals
  - Allows numeric entry but no operations.
- contextLimitedPoint.pl [1]
  - `Context("LimitedPoint")`
  - Allow point entry but no point operations.
- contextLimitedPolynomial.pl [8]
  - `Context("LimitedPolynomial");`
  - Allow only entry of expanded polynomials.
- contextLimitedPowers.pl [9]
  - `LimitedPowers::NoBaseE();`
  - `LimitedPowers::OnlyIntegers();`
  - `LimitedPowers::OnlyPositiveIntegers();`
  - `LimitedPowers::OnlyNonNegativeIntegers();`
  - Restrict the base or power allowed in exponentials.
- contextLimitedVector.pl [10]
  - `Context("LimitedVector-coordinate");`
  - `Context("LimitedVector-ijk");`
  - `Context("LimitedVector");` # either one
  - Allow vector entry but no vector operations.

- contextOrdering.pl [11]
  - `Context("Ordering")`
  - `Context("Ordering-List")`
  - Create objects which parse answers such as `$ans = Ordering("B > A = D > C");`
- contextPeriodic.pl [12]
  - [DEPRECATED] The features in this file are now standard features of the Real and Complex MathObjects classes.
- contextPiecewiseFunction.pl [13]
  - `Context("PiecewiseFuntion");`
  - Allow usage of piecewise functions.
- contextReaction.pl [14]
  - `Context("Reaction");`
  - Implements the specification and comparison of chemical reactions formulas.
  - `$R = Formula("4P + 5O_2 --> 2P_2O_5");`
- contextScientificNotation.pl [15]
  - `Context("ScientificNotation");`
  - Allows entry of scientific notation. Tries hard to report useful error messages when student's answer is not in the proper format.
  - Experimental: may be renamed to LimitedScientifcNotation
- contextString.pl [16]
  - Use this context to allow restricted short answer (no mathematical symbols) questions.
  - Feedback is provided if the response contains strings that are not allowed.
  - See also parserAutoStrings.pl [17]
- contextTF.pl [18]
  - `Context("TF");`

References

[1] http://webwork.maa.org/pod/pg_TRUNK/macros/contextABCD.pl.html
[2] http://webwork.maa.org/pod/pg_TRUNK/macros/contextCurrency.pl.html
[3] http://webwork.maa.org/pod/pg_TRUNK/macros/contextFraction.pl.html
[4] http://webwork.maa.org/pod/pg_TRUNK/macros/contextInequalities.pl.html
[5] http://webwork.maa.org/pod/pg_TRUNK/macros/contextIntegerFunctions.pl.html
[6] http://webwork.maa.org/pod/pg_TRUNK/macros/contextLimitedComplex.pl.html
[7] http://webwork.maa.org/pod/pg_TRUNK/macros/contextLimitedNumeric.pl.html
[8] http://webwork.maa.org/pod/pg_TRUNK/macros/contextLimitedPolynomial.pl.html
[9] http://webwork.maa.org/pod/pg_TRUNK/macros/contextLimitedPowers.pl.html
[10] http://webwork.maa.org/pod/pg_TRUNK/macros/contextLimitedVector.pl.html
[11] http://webwork.maa.org/pod/pg_TRUNK/macros/contextOrdering.pl.html
[12] http://webwork.maa.org/pod/pg_TRUNK/macros/contextPeriodic.pl.html
[13] http://webwork.maa.org/pod/pg_TRUNK/macros/contextPiecewiseFunction.pl.html
[14] http://webwork.maa.org/pod/pg_TRUNK/macros/contextReaction.pl.html
[15] http://webwork.maa.org/pod/pg_TRUNK/macros/contextScientificNotation.pl.html
[16] http://webwork.maa.org/pod/pg_TRUNK/macros/contextString.pl.html
[17] http://webwork.maa.org/pod/pg_TRUNK/macros/parserAutoStrings.pl.html
[18] http://webwork.maa.org/pod/pg_TRUNK/macros/contextTF.pl.html

Specialized parsers

`parserYourModsHere.pl` Modifications to the Parser change the way that student responses are interpreted. By convention files that modify the parser are named starting with "parser" -- e.g. `parserYourModsHere.pl`. test SpecializedContexts Describes another advanced method for customizing MathObjects by modifying the context.

Examples of modifications are give below.

A description of advanced techniques for customizing the parser are at Modifying Contexts (advanced).

Specialized Parser Macro files

Here is a partial list of the parser modifying files.

Use `loadMacros("parserAssignment.pl");` to make the parser modifications available for a WeBWorK question.

Check the POD documentation for more examples. [1]

- parserAssignment.pl [1]
  - checks answers of the form $y = 3x + 5$with the LHS of the equation required.
  - follow the link above to see the additional statements that must be inserted in the question to use this file.
- parserAutoStrings.pl [17]
  - `parserAutoStrings.pl` - Force String() to accept any string as a potential answer.

```
AutoStrings()
  -- all strings are accepted
DefineStrings("string1", "string2")
DefineStrings(qw(string1 string2))
  -- is a quick way to define "legitimate" strings.
```

- parserCustomization.pl [2]
  - Placeholder for site/course-local customization file.
- parserDifferenceQuotient.pl [3]
  - An answer checker for difference quotients.
- parserFormulaUpToConstant.pl [4]
  - implements formulas ``plus a constant.
  - Students must include the ``+C *as part of their answers*
- parserFormulaWithUnits.pl [5]
  - Implements a formula with units.
  - For example:

```
   FormulaWithUnits("3x+1 ft")->cmp
   FormulaWithUnits($a*$x+1,"ft")->cmp
```

- parserFunction.pl [2]
  - An easy way of adding new functions to the current context.
  - `parserFunction("f(x)" => "sqrt(x+1)-2");`
- parserImplicitEquation.pl [6]
  - An answer checker for implicit equations.

```
Context("ImplicitEquation");
$f = ImplicitEquation("x^2 = cos(y)");
$f = ImplicitEquation("x^2 - 2y^2 = 5",limits=>-3,3],[-2,2);
$f = ImplicitEquation("x=1/y",tolerance=>.0001);
```

- parserImplicitPlane.pl [7]
  - Implement implicit planes.

```
$P = ImplicitPlane(Point(1,0,2),Vector(-1,1,3)); #  -x+y+3z = 5
$P = ImplicitPlane([1,0,2],[-1,1,3]);            #  -x+y+3z = 5
$P = ImplicitPlane([1,0,2],4);                   #  x+2z = 4
$P = ImplicitPlane("x+2y-z=5");
```

- parserMultiAnswer.pl [8]
  - Tie several blanks to a single answer checker.
- parserNumberWithUnits.pl [9]
  - Implements a number with units.

```
ANS(NumberWithUnits("3 ft")->cmp);
ANS(NumberWithUnits("$a*$b ft")->cmp);
ANS(NumberWithUnits($a*$b,"ft")->cmp);
```

- parserParametricLine.pl [10]
  - Implements Formulas that represent parametric lines.
- parserPopUp.pl [11]
  - Pop-up menus compatible with Value objects.
- parserPrime.pl [12]
  - Defines `f'(x)` so that students and authors can use the prime notation in defining equations.
  - After loading file use perl command `parser::Prime->Enable;` to enable the feature.
- parserRadioButtons.pl [13]
  - Radio buttons compatible with Value objects, specifically MultiAnswer objects.
- parserSolutionFor.pl [14]
  - An answer checker that checks if a student's answer satisifies a (possibly non-linear) implicit equation.
- parserVectorUtils.pl [15]
  - Utility macros that are useful in vector problems.

References

[1] http://webwork.maa.org/pod/pg_TRUNK/macros/parserAssignment.pl.html
[2] http://webwork.maa.org/pod/pg_TRUNK/macros/parserCustomization.pl.html
[3] http://webwork.maa.org/pod/pg_TRUNK/macros/parserDifferenceQuotient.pl.html
[4] http://webwork.maa.org/pod/pg_TRUNK/macros/parserFormulaUpToConstant.pl.html
[5] http://webwork.maa.org/pod/pg_TRUNK/macros/parserFormulaWithUnits.pl.html
[6] http://webwork.maa.org/pod/pg_TRUNK/macros/parserImplicitEquation.pl.html
[7] http://webwork.maa.org/pod/pg_TRUNK/macros/parserImplicitPlane.pl.html
[8] http://webwork.maa.org/pod/pg_TRUNK/macros/parserMultiAnswer.pl.html
[9] http://webwork.maa.org/pod/pg_TRUNK/macros/parserNumberWithUnits.pl.html
[10] http://webwork.maa.org/pod/pg_TRUNK/macros/parserParametricLine.pl.html
[11] http://webwork.maa.org/pod/pg_TRUNK/macros/parserPopUp.pl.html
[12] http://webwork.maa.org/pod/pg_TRUNK/macros/parserPrime.pl.html
[13] http://webwork.maa.org/pod/pg_TRUNK/macros/parserRadioButtons.pl.html

[14] http://webwork.maa.org/pod/pg_TRUNK/macros/parserSolutionFor.pl.html
[15] http://webwork.maa.org/pod/pg_TRUNK/macros/parserVectorUtils.pl.html

String (MathObject Class)

String Class

The String class adds the ability to have special words or phrases be recognized by the MathObjects library. The two pre-defined words are `DNE` (for "does not exist") and `NONE`. By default, strings are not case sensitive, so `dne` and `DNE` (and `Dne`, and `dNe`, etc.) are all considered to be the same thing. The allowed strings are part of the Context, and you can add your own (or remove the two pre-defined ones), and you can make case-sensitive strings if you desire. See the String Context changes section for details. Strings can be used in any Context.

Creation

```
$ans = String("DNE");
$ans = Compute("DNE");

$ans = Compute("DNE,NONE");  # a list of strings
```

Answer Checker

As with all MathObjects, you obtain an answer checker for a String object via the `cmp()` method:

```
ANS(String("DNE")->cmp);
```

The String class supports the common answer-checker options, and the following additional options:

Option	Description	Default
`typeMatch => $object`	Specifies the type of object that the student should be allowed to enter (determines what constitutes a type mismatch error). Can be either a MathObject or a string that is the class of a MathObject (e.g., `"Value::Vector"`).	`1`

If a string answer is used where something other than a number could have been the answer (e.g., as a response to "At what points (x, y) is $\sqrt{x^2 + y^2}$ undefined?"), then you should indicate the type of answer that might be expected when you create the answer checker by setting the `typeMatch` parameter to an instance of that type. E.g.,

```
ANS(String("NONE")->cmp(typeMatch => Point(0,0)));
```

This will make sure that syntax checking and error messages are appropriate for the kind of answer that the student might submit.

Methods

The String class supports the Common MathObject Methods. There are no additional methods for this class.

Properties

The String class supports the Common MathObject Properties, and the following additional ones:

Property	Description	Default
`$r->{caseSensitive}`	Determines whether or not comparisons to other strings are case sensitive.	defined in Context

Union (MathObject Class)

Union Class

The Union class implements finite unions of Intervals and Sets. Unions are formed via the `U` symbol in student answers or parsed strings, or by addition or the dot operator or the `Union()` constructor in Perl code. The order of the Sets and Intervals in the Union does not matter. Unions are created most often in the `Interval` Context.

The answer checker for Unions will issue warnings if the Union contains overlapping Intervals or Sets, or if Sets contains repeated elements. These warnings can be controlled by answer checker options.

Creation

```
Context("Interval");

$U = Interval("[-1,1]") + Interval("(5,infinity)");
$U = Union(Interval("[-1,1]"),Interval("(5,infinity)"));
$U = Union("[-1,1] U (5,infinity)");
$U = Compute("[-1,1] U (5,infinity)");
```

When Unions are created as part of the code of a problem, things like overlapping intervals are reduced automatically so that `(-1,2) U [0,3]` will be converted to `(-1,3]`. In general, however, students must enter unions of *non*-overlapping intervals and sets, though this can be controlled by answer checker flags. You can turn off automatic reduction by setting the flag `reduceUnions` to 0 in the Context:

```
Context()->flags->set(reduceUnions => 0);
```

This will preserve the Union in whatever form it was originally created (this is set to 0 for student answers so that automatic reductions are not performed). A second flag, `reduceUnionsForComparison`, determines whether Unions are reduced (temporarily) before they are compared for equality, or whether the structures must match exactly.

Operations on Unions

Differences of Unions with Sets, Intervals, or other Unions can be obtained via subtraction.

```
$W = Union("(0,1) U (2,5)") - Interval("(3,4)");    # same as Union("(0,1) U (2,3] U [4,5)");
$W = Compute("(0,1) U (2,5) - (3,4)");             # same as above
```

Intersections of Unions with other Sets, Intervals, or Unions can be obtained via the `intersect()` method of a Union. There is no built-in method for students to form intersections (though one could be added to the Context by hand). There are other methods for determining if one Union is contained in another, or intersects one, or is a subset of another, etc.

```
$U1 = Union("(0,3] U {4,5}");
$U2 = Union("[3,4] U {0}");

$U3 = $U1->intersect($U2);           # same as Set(3,4);

$U1->contains($U2);                  # returns false
$U3->isSubsetOf($U2);                # returns true
$U1->intersects($U2);                # returns true
```

Answer Checker

As with all MathObjects, you obtain an answer checker for a Union object via the `cmp()` method:

```
ANS(Compute("(-infinity,-1] U [1,infinity)")->cmp);
```

The Union class supports the common answer-checker options, and the following additional options:

Option	Description	Default
`showHints => 1` or `0`	Do/don't show messages about which entries in the Union are incorrect.	`$showPartialCorrectAnswers`
`showLengthHints => 1` or `0`	Do/don't show messages about having the correct number of entries (only shown when all the student answers are correct but there are more needed, or all the correct answsers are among the ones given, but some extras were given).	`$showPartialCorrectAnswers`
`partialCredit => 1` or `0`	Do/don't give partial credit for when some entries are right, but not all.	`$showPartialCorrectAnswers`
`ordered => 1` or `0`	Give credit only if the student answers are in the same order as the professor's answers.	`0`

Note that since a Union is a subclass of List, the other options for the List answer checker also can be used here, though their defaults have been set to appropriate values for a Set and it is unlikely you will need to change them.

Option	Description	Default
`entry_type => "a (name)"`	The string to use in error messages that identifies the type of elements that make up the Union.	`"an interval or set"`
`list_type => "a (name)"`	The string to use in error messages that identifies the Union itself.	`"an interval, set or union"`
`short_type => "a (name)"`	A short string to use in error messages that identifies the Union itself.	`"a union"`
`typeMatch => $object`	Specifies the type of object that the student should be allowed to enter in the union (determines what constitutes a type mismatch error). Can be either a MathObject or a string that is the class of a MathObject (e.g., `"Value::Vector"`).	`Interval("(0,1)")`

Note also that since Union is a subclass of List, if you supply a custom `checker` option, it operates the same as for a List. You probably want to use a `list_checker` instead. See Custom Answer Checkers for Lists for details.

Methods

The Union class supports the Common MathObject Methods, and the following additional ones:

Method	Description
`$U1->intersect($U2)`	Returns the intersection of `$U1` with `$U2`. Note that `$U2` can be an Interval, Set, or Union.
`$U1->intersects($U2)`	Returns `true` if `$U1` intersects `$U2`, and `undef` otherwise. Note that `$U2` can be an Interval, Set, or Union.
`$U1->contains($U2)`	Returns `true` if `$U2` is a subset of `$U1`, and `undef` otherwise. Note that `$U2` can be an Interval, Set, or Union.
`$U1->isSubsetOf($U2)`	Returns `true` if `$U1` is a subset of `$U2`, and `undef` otherwise. Note that `$U2` can be an Interval, Set, or Union.
`$U->isEmpty`	Returns `true` if `$U` is the empty set, and `undef` otherwise.
`$U->isReduced`	Returns `true` if there are no overlapping Intervals or Sets, and no Sets contain repeated elements in `$U`. Returns `undef` otherwise.
`$U->reduce`	Returns a copy of `$U` without any overlapping Intervals or Sets, and where no Sets contain repeated elements.
`$U->sort`	Returns a copy of `$U` with its Intervals and Sets sorted lexicographically.

Properties

The Union class supports the Common MathObject Properties. There are no additional properties for this class.

Vector (MathObject Class)

Vector Class

The Vector class implements vector quantities in for arbitrary n. Typically, vectors are delimited by angle brackets, as in `<-2,4,0>`, but that can be controlled by settings in the Context. Vectors can also be produced using the coordinate unit vectors, `i`, `j`, and `k`, as in the examples below. Vectors typically are created in the `Vector` or `Matrix` Contexts. There is also a `Vector2D` Context which is specifically for vectors in the plane. Here, `i` and `j` are defined as two-dimensional vectors, and `k` is not defined. It is possible to create Vectors in , though there is no pre-defined Context that makes this easy to do.

The answer checker for Vectors can give students hints about the coordinates that are wrong, and about whether the number of coordinates is correct.

The file `pg/macros/parserVectorUtils.pl` includes some useful routines for obtaining things like non-zero points and vectors, or equations of lines and planes; see the POD documentation [15] for details.

Creation

Vectors are created via the `Vector()` function, or by `Compute()`.

```
Context("Vector");

$v = Vector(-2,4,0);
$v = Vector([-2,4,0]);
$v = Vector("<-2,4,0>");
$v = Compute("<-2,4,0>");
$v = Vector("-2i + 4j");
$v = Compute("-2i + 4j");
$v = -2*i + 4*j;
```

There is a special constructor, `ColumnVector()` which will produce a Vector object that displays vertically in TeX output. This is useful for matrix-vector multiplication, or systems of equations, for example. A ColumnVector acts just like a regular vector in terms of the operations and functions described above; its only difference is that it displays vertically.

```
$v = ColumnVector(5,-3,-2);
```

Operations on Vectors

Vectors (of the same dimension) can be added to and subtracted from each other. Vectors can be multiplied and divided by scalars.

```
$u = $v + Vector(3,1,-1);      # same as Vector(1,5,-1);
$u = $v + [3,1,-1];            # same as Vector(1,5,-1);
$u = 3*$v;                     # same as Vector(-6,12,0);
$u = $v/2;                     # same as Vector(-1,2,0);
```

The dot product can be obtained by using a period (`.`) between two Vectors, and the cross product in is formed by an `x` between two Vectors in Perl code, or by `><` in student answers or other strings parsed by MathObjects. The absolute value of a Vector is its length, which can also be obtained from the `norm()` function. A unit vector in the same direction (as a non-zero Vector) can be obtained from the `unit()` function.

```
$a = $u . $v;                  # dot product
$a = Compute("$u . $v");

$w = $u x $v;                  # cross product
$w = Compute("$u >< $v");

$b = norm($v);                 # length of a vector
$b = abs($v);                  # same
$b = Compute("|$v|");
$b = Compute("abs($v)");
$b = Compute("norm($v)");

$w = unit($v);                 # unit vector in direction of $v
$w = Compute("unit($v)");
```

If a Vector is combined with a Point, the Point will first be converted to a Vector, and then combined to form a Vector result:

```
$v = Vector(3,1,-1);
$p = Point(1,-3,5);
$t = Formula("t");

$w = $v + $p;          # same as Vector(4,-2,4);

$f = $p + $t * $v;     # a parametric line through $p in the direction of $v
$f = Formula("(1,-3,5) + t<3,1,-1>");
```

Answer Checker

As with all MathObjects, you obtain an answer checker for a Vector object via the `cmp()` method:

```
ANS(Compute("<1,5,-2>")->cmp);
```

The Vector class supports the common answer-checker options, and the following additional options:

Option	Description	Default
`showDimensionHints => 1` or `0`	Show/don't show messages about the wrong number of coordinates.	`1`
`showCoordinateHints => 1` or `0`	Show/don't show message about which coordinates are right.	`1`
`promotePoints => 1` or `0`	Do/don't allow the student to enter a point rather than a vector.	`1`
`parallel => 1` or `0`	Mark the answer as correct if it is parallel to the professor's answer. Note that a value of `1` forces `showCoordinateHints` to be `0`.	`0`
`sameDirection => 1` or `0`	During a parallel check, mark the answer as correct only if it is in the same (not the opposite) direction as the professor's answer.	`0`

By default, the Vector answer checker asks the student to type the entire vetor, including the angle-braces and commas. This allows the student to enter vector-values expressions (like sums of vectors, or cross products, or scalar multiples of vectors). You may want to restrict the operations that are allowed in the student answer, in which case you might consider using the `LimitedVector` Context available in the

`pg/macros/contextLimitedVector.pl` file; see the POD documentation [10] for details.

Alternatively, you can use the Vector's `ans_array()` method rather than PG's `ans_rule()` function to obtain a separate answer box for each coordinate. The Vector answer checker will manage this collection of answer boxes for you, so your code doesn't have to change in any other way.

```
Context("Vector");

$V = Vector(random(1,10,1),random(1,10,1),-1);

Context()->texStrings;
BEGIN_TEXT
\($V\) = \{$V->ans_array\}
END_TEXT
Context()->normalStrings;

ANS($V->cmp);
```

The value of this approach is that it forces the student to enter individual coordinates, without allowing operations on Vectors.

Methods

The Vector class supports the Common MathObject Methods, and the following additional methods:

Method	Description
`$v . $w` or `$v->dot($w)`	Returns the dot product of the two vectors as a Real.
`$v x $w` or `$v->cross($w)`	Returns the cross product of the two vectors (in) as a Vector.
`norm($v)` or `$v->norm`	Returns the magnitude of `$c` as a Real. Also available as `$v->abs` or `abs($v)`.
`unit($v)` or `$v->unit`	Returns the unit vector in the same direction as `$v` (or `$v` if `$v` is the zero vector) as a Vector.
`areParallel $v $w` or `$v->isParallel($w)`	Returns 1 if the two vectors are parallel (and non-zero), and 0 otherwise.

Properties

The Vector class supports the Common MathObject Properties, and the following additional ones:

Property	Description	Default
`$r->{ColumnVector}`	When set, the vector will display as vertically in TeX output, horizontally otherwise	`undef`
`$r->{open}`	The symbol to use for the open angle bracket	<
`$r->{close}`	The symbol to use for the close angle bracket	>

Article Sources and Contributors

Additional MathObjects documentation *Source*: http://webwork.maa.org/w/index.php?oldid=1707 *Contributors*: Gage

Answer Checker Options (MathObjects) *Source*: http://webwork.maa.org/w/index.php?oldid=13427 *Contributors*: Dpvc

Answer Checkers (MathObjects) *Source*: http://webwork.maa.org/w/index.php?oldid=12820 *Contributors*: Dpvc

Answer Checkers and the Context *Source*: http://webwork.maa.org/w/index.php?oldid=13256 *Contributors*: Dpvc

Common Contexts *Source*: http://webwork.maa.org/w/index.php?oldid=13293 *Contributors*: Dpvc

Common MathObject Methods *Source*: http://webwork.maa.org/w/index.php?oldid=13289 *Contributors*: Dpvc

Common MathObject Properties *Source*: http://webwork.maa.org/w/index.php?oldid=13294 *Contributors*: Dpvc

Complex (MathObject Class) *Source*: http://webwork.maa.org/w/index.php?oldid=12840 *Contributors*: Dpvc

Context flags *Source*: http://webwork.maa.org/w/index.php?oldid=13295 *Contributors*: Aweyhau, Dpvc, Gage, Sam

Context Function Categories *Source*: http://webwork.maa.org/w/index.php?oldid=13296 *Contributors*: Dpvc

Context Operator Table *Source*: http://webwork.maa.org/w/index.php?oldid=13945 *Contributors*: Dpvc

Course-Wide Customizations *Source*: http://webwork.maa.org/w/index.php?oldid=14191 *Contributors*: Dpvc

Creating Custom Contexts *Source*: http://webwork.maa.org/w/index.php?oldid=14190 *Contributors*: Dpvc

Custom Answer Checkers *Source*: http://webwork.maa.org/w/index.php?oldid=14188 *Contributors*: Dpvc, Yoavfreund

Custom Answer Checkers for Lists *Source*: http://webwork.maa.org/w/index.php?oldid=12700 *Contributors*: Dpvc

Files Defining MathObjects *Source*: http://webwork.maa.org/w/index.php?oldid=12670 *Contributors*: Dpvc

Formula (MathObject Class) *Source*: http://webwork.maa.org/w/index.php?oldid=13230 *Contributors*: Dpvc

How to list Context flags *Source*: http://webwork.maa.org/w/index.php?oldid=13304 *Contributors*: Aweyhau, Dpvc, Gage

Infinity (MathObject Class) *Source*: http://webwork.maa.org/w/index.php?oldid=12806 *Contributors*: Dpvc

Interval (MathObject Class) *Source*: http://webwork.maa.org/w/index.php?oldid=13269 *Contributors*: Dpvc

Introduction to Contexts *Source*: http://webwork.maa.org/w/index.php?oldid=14205 *Contributors*: Chris Wingard, Darnold, Dpvc, Gage, Paultpearson, Sam, Travis

Introduction to MathObjects *Source*: http://webwork.maa.org/w/index.php?oldid=13222 *Contributors*: Dpvc, Gage, Jipsen, Pearson, Pjvpjv, Sam

List (MathObject Class) *Source*: http://webwork.maa.org/w/index.php?oldid=12811 *Contributors*: Dpvc

Matrix (MathObject Class) *Source*: http://webwork.maa.org/w/index.php?oldid=12835 *Contributors*: Dpvc

Modifying Contexts (advanced) *Source*: http://webwork.maa.org/w/index.php?oldid=13579 *Contributors*: Dpvc, Gage, Sam

Point (MathObject Class) *Source*: http://webwork.maa.org/w/index.php?oldid=12808 *Contributors*: Dpvc

Presentations on MathObjects *Source*: http://webwork.maa.org/w/index.php?oldid=11569 *Contributors*: Aubreyja, Gage, Gjennings, Sam

Real (MathObject Class) *Source*: http://webwork.maa.org/w/index.php?oldid=13642 *Contributors*: Dpvc

Reduction rules for MathObject Formulas *Source*: http://webwork.maa.org/w/index.php?oldid=13298 *Contributors*: Dpvc, Gage, Sam

Set (MathObject Class) *Source*: http://webwork.maa.org/w/index.php?oldid=13316 *Contributors*: Dpvc, Yoavfreund

Specialized contexts *Source*: http://webwork.maa.org/w/index.php?oldid=13299 *Contributors*: Dpvc, Gage, Sam, Xiong Chiamiov

Specialized parsers *Source*: http://webwork.maa.org/w/index.php?oldid=13944 *Contributors*: Aubreyja, Dpvc, Gage, Gjennings, Sam

String (MathObject Class) *Source*: http://webwork.maa.org/w/index.php?oldid=12815 *Contributors*: Dpvc

Union (MathObject Class) *Source*: http://webwork.maa.org/w/index.php?oldid=12814 *Contributors*: Dpvc

Vector (MathObject Class) *Source*: http://webwork.maa.org/w/index.php?oldid=14854 *Contributors*: Dpvc

License