1. {Real(2)->with(period=>10)}

```
TEXT(beginproblem());

BEGIN_PGML
The number twelve is [_______]{12}
Type the  formula [`1+\frac{x}{2}`]
[_________]{"1+x/2"}

Twelve is [_____]{Real(12)}
2 mod 10 is
[_____]{Real(2)->with(period=>10)}
[`[$f]`] is equal to [_______]{Real(1)}
Twelve is [_____]{num_cmp(12)}

The number 12 is
[____]{answer=>12,width=>10}
END_PGML
```

The TEXT(beginproblem()); line displays a **header** for the problem.

Everything between BEGIN_PGML and END_PGML (each of which must appear alone on a line) is shown to the student.
BEGIN_PGML and END_PGML replace the BEGIN_TEXT/END_TEXT structure used in older-style template samples.
The Context()->texStrings seen in those samples line is **not needed when using PGML**.

Answer blanks are indicated by [_______] where **the number of blanks indicates the width of the answer blank**. The **correct answer** can be given in curly braces immediately afterward {"1+x/2"}.

TeX formulas within the text of the problem can be entered as [`1+\frac{x}{2}`]. A **variable** substitution would be given as [$a], while [`[$f]`] typesets the formula for $f in inline math mode.

The statement `{Real(2)->with(period=>10)}` in the PGML code **configures the answer checker** for a **periodic (modular) equivalence** context.

A. `Real(2)`:

Creates a MathObject expecting a **real number answer** of **2**.

B. The `with()` Method:

```
# If you wanted 'true' (lowercase) to also be accepted as 'True',
you could add:
Context()->strings->add(
    True => {caseSensitive => 0}, # Allow case-insensitive
    False => {alias => 'F'}       # Allow "F" as an alias for "False"
);
```

The `with()` method is a **MathObject method** used to **customize how an answer is checked, formatted, or interpreted**. It allows you to **attach additional properties or flags** to a **MathObject** to modify its behavior.

When you call `->with`, you are **modifying the answer checker** associated with the **MathObject**, **not the MathObject itself**!

It also accepts **Key-Value Pairs**, we can pass multiple **settings** as a **hash** (e.g., `->with(tolType => 'absolute', tolerance => 0.1, period => 10)`). **Common parameters** include `period`, `tolerance`, `tolType`, `limits`, `checker`, etc.

Used with **Real**, **Formula**, **Complex**, **Interval**, and other **MathObject types**.

C. The `period` Parameter

The `period` parameter is a specialized setting used to enforce **modular equivalence** for numerical answers. It tells WeBWorK to accept any answer that is congruent to the given value modulo the specified period.

Mathematical Meaning:

If `period => N` is set, the **answer checker** will accept any number x such that:

$$x \equiv < answer > \ mod\ N$$

For `Real(2)->with(period => 10)`, valid answers include **2**, **12**, **-8**, **102**, etc., because they all satisfy $x \equiv 2 \ mod\ 10$.

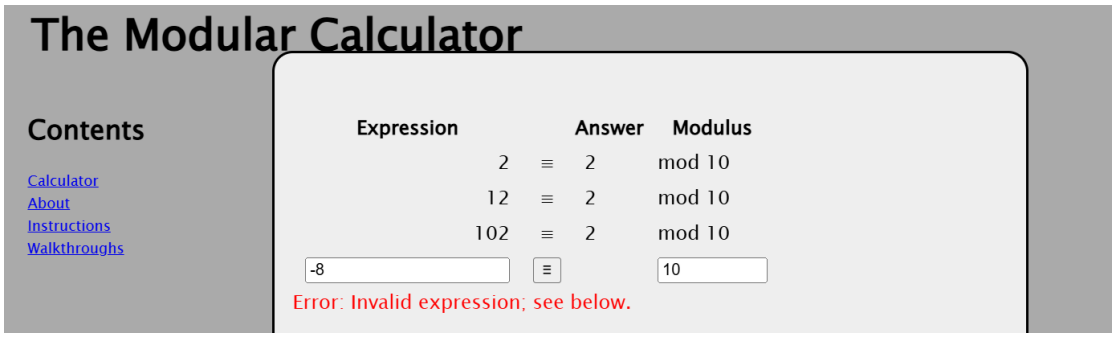


Figure 1 The Modular Calculator

*** How can $2 \bmod 10$ be 12?**

This is about **modular equivalence**, **not** the **remainder operation**.

Mathematical Definition:

- ✧ $a \equiv b \bmod N$ means $a - b$ is divisible by N.
- ✧ Example: $12 \equiv 2 \ mod\ 10$ because $12 - 2 = 10$, which is divisible by 10.

Correct answers: **2**, **12**, **22**, **-8**, **-18**, etc.

Remainder vs. Modular Equivalence:

- ✧ When you compute $2 \div 10$, the remainder is 2 (**always non-negative** and less than 10).

2. From PG to PGML: Have any features been lost?

PGML sacrifices some low-level PG flexibility for improved readability and maintainability. However, **no critical functionality is truly "lost"**—PGML problems can always fall back to embedded PG code for edge cases.

3. `Operators` method in `Context()`

The **Operators()** method in **Context()** of WeBWorK's MathObjects framework allows programmers to **modify the set of mathematical operators** (e.g., **+**, **-**, *****, **/**, **^**, etc.) that are recognized and allowed in student answers. This is critical for **controlling how expressions are parsed and evaluated**.

Operators include:

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)

REF: Context Operator Table - WeBWorK_wiki

The way the operators are defined in the code:

```
$operators = {
    ',' => {
        precedence    => 0,
        associativity => 'left',
        type          => 'bin',
        string        => ',',
        class         => 'Parser::BOP::comma',
        isComma       => 1
    },

    '+' => {
        precedence    => 1,
        associativity => 'left',
        type          => 'both',
        string        => '+',
        class         => 'Parser::BOP::add'
    },

    '-' => {
        precedence    => 1,
        associativity => 'left',
        type          => 'both',
        string        => '-',
        class         => 'Parser::BOP::subtract',
        rightparens   => 'same',
        alternatives  => ["\x{2212}"]
    },

    'U' => {
        precedence    => 1.5,
        associativity => 'left',
        type          => 'bin',
        isUnion       => 1,
        string        => ' U ',
        TeX           => '\cup ',
```

REF: "···\pg-main\lib\Parser\Context\Default.pm"

Two Methods in Operators

```
sub undefine {
    my $self = shift;
    my @data = ();
    foreach my $x (@_) {
        if ($self->{context}{operators}{$x}{type} eq 'unary') {
            push(
                @data,
                $x => {
                    class    => 'Parser::UOP::undefined',
                    oldClass => $self->get($x)->{class},
                }
            );
        } else {
            push(
                @data,
                $x => {
                    class    => 'Parser::BOP::undefined',
                    oldClass => $self->get($x)->{class},
                }
            );
        }
    }
    $self->set(@data);
}
```

```
49  sub redefine {
50      my $self = shift;
51      my $X    = shift;
52      return $self->SUPER::redefine($X, @_) if scalar(@_) > 0;
53      $X = [$X] unless ref($X) eq 'ARRAY';
54      my @data = ();
55      foreach my $x (@{$X}) {
56          my $oldClass = $self->get($x)->{oldClass};
57          push(@data, $x => { class => $oldClass, oldClass => undef })
58              if $oldClass;
59      }
60      $self->set(@data);
61  }
62
```

REF: "…\pg-main\lib\Parser\Context\Operators.pm"

A. `undefine()`

In some problems, you may want to **remove some operators** so that students can't enter them. For example, if you want the student to 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("^","**");
```

which **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.

B. `redefine()`

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('/',' /','/ ','//');
```

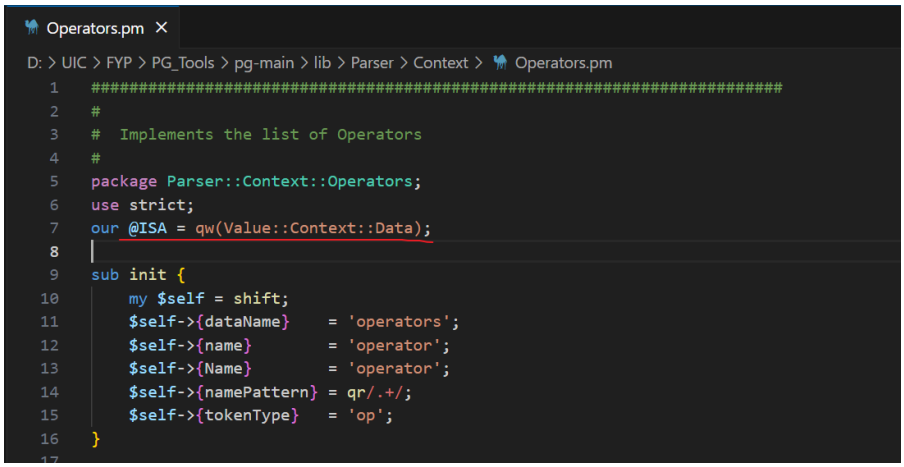
which would be required in order to **make multiplication and division unavailable**.

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.

C. Does Operators.pm only have undefine and redefine methods?



The `Operators` class inherits from `Value::Context::Data`
(via `@ISA = qw(Value::Context::Data)`).

This **parent class** provides **additional methods** like:

- `add`: **Adds new operators** (not explicitly defined in **Operators.pm** but inherited).
- `set`: **Modifies operator properties**.
- `get`: **Retrieves operator definitions**.
- `exists`: **Checks if an operator exists**.
- `remove`: **Removes operators entirely** (different from **undefine**).

For example, the `add` method (used in earlier examples like `Context()->operators->add(...)`):

```
Context()->operators->add(
  '!!' => {
    precedence => 6,          # Higher than * and /
    associativity => 'left',
    type => 'unary',          # Acts on a single operand
    perl => sub { factorial($_[0]) }  # Assume factorial() is defined
  }
);
```

`perl`: A Perl subroutine defining **how the operator is evaluated**.

Data.pm

D: > UIC > FYP > PG_Tools > pg-main > lib > Value > Context > Data.pm

```
117  #
118  #  Add one or more new items to the list
119  #
120  sub add {
121      my $self = shift;
122      my %D    = (@_);
123      return if scalar(@_) == 0;
124      my $data = $self->{context}{ $self->{dataName} };
125      foreach my $x (keys %D) {
126          Value::Error("Illegal %s name '%s'", $self->{name}, $x) unless $x =~ m/^$self->{namePattern}$/;
127          warn "$self->{Name} '$x' already exists" if defined($data->{$x});
128          if ($data->{$x}) {
129              delete $self->{tokens}{$x};
130              delete $self->{patterns}{ Parser::Context::protectRegexp($x) };
131          }
132          $data->{$x} = $self->create($D{$x});
133          $self->addToken($x);
134          if (ref($data->{$x}) eq 'HASH' && $data->{$x}{alias}) {
135              Value::Error("Alias not allowed with %s objects", $self->{name}) unless $self->{allowAlias};
136              my $alias = $data->{$x}{alias};
137              Value::Error("Alias '%s' doesn't exist for %s '%s'", $alias, $self->{name}, $x)
138                  if !(defined($data->{$alias}) || defined($D{$alias}));
139          }
140      }
141      $self->update;
142  }
143
144  #
145  #  Remove one or more items
146  #
147  sub remove {
148      my $self = shift;
149      my $data = $self->{context}{ $self->{dataName} };
150      foreach my $x (@_) {
151          warn "$self->{Name} '$x' doesn't exist" unless defined($data->{$x});
152          $self->removeToken($x);
153          delete $data->{$x};
154      }
155      $self->update;
156  }
```

See "···\pg-main\lib\Value\Context\Data.pm"

D. Why there's a 1 at the end of the file?

E.g. in **Data.pm**:

Data.pm

D: > UIC > FYP > PG_Tools > pg-main > lib > Value > Context > Data.pm

```
304  END {
305      use Value::Context::Flags;
306      use Value::Context::Lists;
307      use Value::Context::Diagnostics;
308  }
309
310  ##################################################################################
311
312  1;
313
```

Also, in **Operators.pm**:

Operators.pm

D: > UIC > FYP > PG_Tools > pg-main > lib > Parser > Context > Operators.pm

```
49   sub redefine {
59       }
60       $self->set(@data);
61   }
62
63   ##################################################################################
64
65   1;
66
```

In **Perl**, a **module must return a true value** to **indicate successful loading**. The **1** at the end of the file serves this purpose. It is a **Perl convention**, not part of the class logic.

- If a **.pm** file **doesn't** end with a true value (like **1**), Perl will **throw an error**: **Module did not return a true value**.
- The **1** is a simple way to satisfy this requirement. **Other truthy values** (e.g., **42;**) would also work, but **1** is standard.