As part of the Advanced Rating project, a mini (lisp-like) language processor was introduced to allow users to safely input formulas to be executed by tiki. In the case of the rating project, the formulas are used to calculate the score of objects based on ratings and other environmental values.

Sample syntax

```lisp
(rating-average
  (object type object-id)
  (revote 1000)
  (keep latest)
  (range (mul 24 36 30))
)
```

The language and the language processor were build to be extensible, for both local customizations and re-use within tiki or other applications as needed.

Execution Model

In the sample above, rating-average and mul are functions. All other tokens are parameters to rating-average. This section will attempt to explain how this is possible.

The language's execution model is very simple. The provided input string is first tokenized, a process by which parentheses are isolated from the rest of the data and where spaces get tossed out. A parser then walks through the list of tokens, matching parentheses and building the syntax tree. The first token in a set of parenthesis is known as the operation type and all other tokens are placed in a list. Any opening parenthesis encountered cause an other node to be nested following the same rules.

The tree must have a single root node, which will be sent for execution. When executing, the operation type is used to identify the class that will handle the node. The operation class is then provided with the node and a callback. The execution is responsible of checking if the content satisfies the requirements in terms of parameter counts.

If everything is in order, it can use the callback to evaluate child nodes, which will resolve variables or pass the control down to child nodes and return once completed for the parent to aggregate the results.

Exceptions are used to flag errors and terminate the execution. The should provide a significant error message to help in debugging.

For development purposes, it's possible to run a formula using a fake callback. It will send down the evaluation calls but skip variable resolving. This is used by the admin panel to validate that the formula parses and is otherwise valid. Instead of resolving variables, it collects them.
Comments can be added anywhere under the root node. Comments use the reserved `comment` function and automatically discard them at parse time.

```
(mul
 (comment 1 day)
 3600 24)
```

Comments can span across multiple lines, but their content is subject to parsing rules as they are a function too.

### Adding functions

As mentioned above, the expression runner uses the operation type to identify the operations to execute. Each operation is coded as a class. As part of the runner's constructor, a list of folders to look in for operations and the matching prefix is provided. By providing different folders to the runner, separate execution environments can be created with specific operations or levels of security.

At this time, operations are divided in two categories. General functions part of the core, such as `mul` and rating-specific functions for score aggregation.

Assuming the rating execution environment, operations that are pure-functional should go in `lib/core/lib/Math/Formula/Function/` and be prefixed with `with Math_Formula_Function_` and become available to all environments. Operations that are task-specific and hit the database should go under `lib/rating/formula/` and be prefixed with `Tiki_Rating_Function_`

Operation names convert dashes to camel case and use upper case for the first letter.

- `mul` becomes `Math_Formula_Function_Mul`
- `rating-average` becomes `Tiki_Rating_Function_RatingAverage`

All operations must extend `Math_Formula_Function` or a derivative.

As part of the execution process, `evaluate( $element )` will be called on the operation with the node to process. The operation will very likely need to access the child nodes of the element sent to it. In the most simple cases, it only needs to iterate over the child nodes. As `Math_Formula_Element` supports most common operators, iterating is possible naturally. It also supports `ArrayAccess` and `Countable` to access individual items by position.

#### Add operation

```php
<?php
require_once 'Math/Formula/Function.php';
class Math_Formula_Function_Add extends Math_Formula_Function {
    function evaluate( $element ) {
        $out = 0;
        foreach( $element as $child ) {
            $out += $this->evaluateChild( $child );
        }
    }
}
```
evaluateChild() is called on each child node to handle cases where the value would be an other function call or a variable.

In other cases, the input to the operation is more complex and the child nodes are actually configuration flags, not functions. In this case, the operation can access them by name. The first example of rating-average is such a case. In (range (mul 3600 24 30)), range is not a function. It merely is a key looked up by the rating-average operation. However, it's value has to be evaluated. Here is a snippet of the operation's code to handle the range attribute.

Snippet from the rating-average operation

```php
<?php
  // ...
  if( $range = $element->range ) {
    if( count($range) == 1 ) {
      $params['range'] = $this->evaluateChild( $range[0] );
    } else {
      $this->error( tra('Invalid range.') );
    }
  }
  // ...
```

The operation first validates that the property is present by accessing it by name on the element. null will be returned if not present. It then validates that the range element contains a single argument and evaluates it. The result is placed in an array for further processing.

The range is not mandatory, so no error is triggered when it is absent. However, if it is present but does not contain a single value, error() is called, which will throw an exception to interrupt the processing.

To prevent typos that would prevent from expected evaluation, the rating-average function also verifies that only the expected keys are present.

Snipper from the rating-average operation

```php
<?php
  // ...
  function evaluate( $element ) {
    $allowed = array( 'object', 'range', 'ignore', 'keep', 'revote' );
    if( $extra = $element->getExtraValues( $allowed ) ) {
      $this->error( tr('Unexpected values: %0', implode( ', ', $extra ) ) );
    }
  }
  // ...
```
Embedding and Internals

The language can be used in other contexts where accepting user input for a calculation is required. Because they are executed in a controlled environment that does not rely on eval(), they can be used safely, as long as the operations made available remain safe.

The most simple usage would be the following:

```php
<?php
// Would technically come from the user
$function = '(mul hello world 2)';

require_once 'Math/Formula/Runner.php';
$runner = new Math_Formula_Runner( array(
    'Math_Formula_Function_' => '/path/to/function/folder',
) );
$runner->setFormula( $function );

// Available variables mostly depend on the context
$runner->setVariables( array(
    'hello' => 2,
    'world' => 3,
) );

echo $runner->evaluate(); // print 12
```

The code above is not safe as it may throw exceptions if the user provides an erroneous formula, like `(mul hello world 2), (mul helo world)` or `(hello world)`. Exceptions should always be handled. Whether they are reported or inhibited is specific to the use case. In the rating use case, they are reported live as the formula is typed in, but discarded at execution. The only case in which an error would occur at execution is if after resolving a variable, an operation becomes impossible and throws an exception, which is likely to be a content-related object-specific situation.

When provided with a string, the runner will instantiate a parser to resolve the formula. The runner can handle a single formula at a time. If multiple calculations must be performed with different formulas, it's better to pre-parse the formulas to avoid re-parsing them every time. It's also possible to create one runner per formula, but some functions may cache data internally and that would be inefficient.

Parsing formulas is trivial (but not recommended unless batch-processing as it adds a significant amount of code).

```php
<?php
// Probably comes from the database
$functions = array(
    array( 'id' => 1, 'formula' => '(mul hello world 2)' ),
    array( 'id' => 2, 'formula' => '(mul hello world 3)' ),
    array( 'id' => 3, 'formula' => '(mul hello world 4)' ),
```

// Some dataset to process
$data = array(
    array( 'hello' => 1, 'world' => 2 ),
    array( 'hello' => 3, 'world' => 4 ),
    array( 'hello' => 5, 'world' => 6 ),
);)

require_once 'Math/Formula/Parser.php';
$parser = new Math_Formula_Parser;
$parsed = array();
foreach( $functions as $f ) {
    $f['formula'] = $parser->parse( $f );
    $parsed[] = $f;
}

require_once 'Math/Formula/Runner.php';
$runner = new Math_Formula_Runner( array(  
    'Math_Formula_Function_' => '/path/to/function/folder',
) );

$out = array();
foreach( $data as $set ) {
    foreach( $parsed as $f ) {
        $runner->setFormula( $f['formula'] );
        $runner->setVariables( $set );
        $out[] = "{$f['id']}-" . $runner->evaluate();
    }
}

echo implode( ', ', $out ); // 1-2, 2-6, 3-8, 1-24, 2-36, ...

To simply test if a function can parse without considering variables, evaluate() can be replaced with inspect(), which will use an alternate callback internally.