The `luacas` package

Evan Cochrane[*], Timothy All[†]

v1.0.2
May 26, 2023

Abstract

The `luacas` package is a portable Computer Algebra System capable of symbolic computation, written entirely in Lua, designed for use in LuaLaTeX.

Contents

I Introduction

---

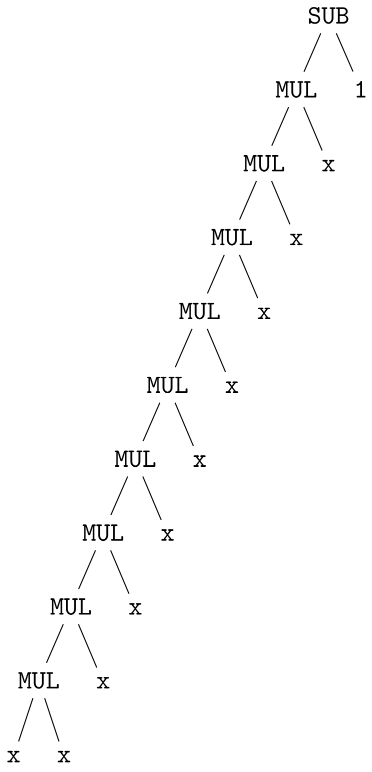
[*] `cochraef@rose-hulman.edu`
[†] `timothy.all@rose-hulman.edu`

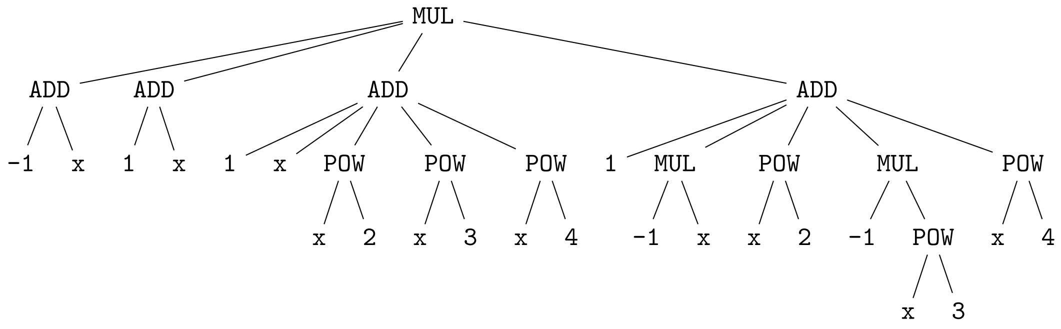
Part I

Introduction

```
\begin{CAS}
    vars('x')
    f = x
    for i in range(1,9) do
        f = f*x
    end
    f = f-1
\end{CAS}
\parseforest{f}
\bracketset{action character = @}
\begin{center}
\begin{forest}
    for tree = {font = \ttfamily}
    @\forestresult
\end{forest}
\end{center}

\begin{CAS}
    f = factor(f)
\end{CAS}
\parseforest{f}
\begin{center}
\bracketset{action character = @}
\begin{forest}
    for tree = {font = \ttfamily}
    @\forestresult
\end{forest}
\end{center}
```





1 What is `luacas`?

The package `luacas` allows for symbolic computation within LaTeX. For example:

```
\begin{CAS}
    vars('x','y')
    f = 3*x*y - x^2*y
    fxy = diff(f,x,y)
\end{CAS}
```

The above code will compute the mixed partial derivative f_{xy} of the function f defined by

$$f(x, y) = 3xy - x^2y.$$

There are various methods for fetching and/or printing results from the CAS within your LaTeX document:

```
\[ \print{fxy} = \print*{fxy} \]
```

$$\frac{\partial^2}{\partial y \partial x}\left(3xy - x^2y\right) = 3 - 2x$$

1.1 About

The core CAS program is written purely in Lua and integrated into LaTeX via LuaLaTeX. Currently, most existing computer algebra systems such as Maple and Mathematica allow for converting their stored expressions to LaTeX code, but this still requires exporting code from LaTeX to another program and importing it back, which can be tedious.

The target audience for this package are mathematics students, instructors, and professionals who would like some ability to perform basic symbolic computations within LaTeX without the need for laborious and technical setup. But truly, this package was born out of a desire from the authors to learn more about symbolic computation. What you're looking at here is the proverbial "carrot at the end of the stick" to keep our learning moving forward.

Using a scripting language (like Lua) as opposed to a compiled language for the core CAS reduces performance dramatically, but the following considerations make it a good option for our intentions:

- Compiled languages that can communicate with LaTeX in some way (such as C through Lua) require compiling the code on each machine before running, reducing portability.
- Our target usage would generally not involve computations that take longer than a second, such as factoring large primes or polynomials.
- Lua is a fast scripting language, especially when compared to Python, and is designed to be compact and portable.
- If C code could be used, we could tie into one of many open-source C symbolic calculators, but the point of this project was (and continues to be) to learn the mathematics of symbolic computation. The barebones but friendly nature of Lua made it an ideal language for those intents.

1.2 Features

Currently, `luacas` includes the following functionality:

- Arbitrary-precision integer and rational arithmetic
- Number-theoretic algorithms for factoring integers and determining primality
- Constructors for arbitrary polynomial rings and integer mod rings, and arithmetic algorithms for both
- Factoring univariate polynomials over the rationals and over finite fields
- Polynomial decomposition and some multivariate functionality, such as pseudodivision

- Basic symbolic root finding and equation solving
- Symbolic expression manipulations such as expansion, substitution, and simplification
- Symbolic differentiation and integration

The CAS is written using object-oriented Lua, so it is modular and would be easy to extend its functionality.

1.3 Acknowledgements

We'd like to thank the faculty of the Department of Mathematics at Rose-Hulman Institute of Technology for offering constructive feedback as we worked on this project. A special thanks goes to Dr. Joseph Eichholz for his invaluable input and helpful suggestions.

2 Installation

2.1 Requirements

The `luacas` package (naturally) requires you to compile with LuaLaTeX. Lua 5.3 or higher is also required. Beyond that, the following packages are needed:

- `xparse`
- `pgfkeys`
- `verbatim`
- `mathtools`
- `luacode`
- `iftex`
- `tikz/forest`
- `xcolor`

The packages `tikz`, `forest`, and `xcolor` aren't strictly required, but they are needed for drawing expression trees.

2.2 Installing `luacas`

The package manager for your local TeX distribution ought to install the package fine on its own. But for those who like to take matters into their own hands: unpack `luacas.zip` in the current working directory (or in a directory visible to TeX, like your local texmf directory), and in the preamble of your document, put:

```
\usepackage{luacas}
```

That's it, you're ready to go.

2.3 Todo

Beyond squashing bugs that inevitably exist in any new piece of software, future enhancements to `luacas` may include:

- Improvements to existing functionality, e.g., a more powerful `simplify()` command and more powerful expression manipulation tools in general, particularly in relation to complex numbers, a designated class for multivariable polynomial rings, irreducible factorization over multivariable polynomial rings, and performance improvements;
- New features in the existing packages, such as sum and product expressions & symbolic evaluation of both, and symbolic differential equation solving;
- New packages, such as for logic (boolean expressions), set theory (sets), and linear algebra (vectors and matrices), and autosimplification rules and algorithms for all of them;
- Numeric functionality, such as numeric root-finding, linear algebra, integration, and differentiation;
- A parser capable of evaluating arbitrary LaTeX code and turning it into CAS expressions.

3 Tutorials

Taking a cue from the phenomenal TikZ documentation, we introduce basic usage of the `luacas` package through a few informal tutorials. In the subsections that follow, we'll walk through how each of the outputs below are made using `luacas`. **Crucially**, none of the computations below are "hardcoded"; all computations are performed and printed using `luacas` to maximize portability and code reuse.

Tutorial 1: *A limit definition of the derivative for Alice.*

Let $f(x) = 2x^3 - x$. We wish to compute the derivative of $f(x)$ at x using the limit definition of the derivative. Toward that end, we start with the appropriate difference quotient:

$$\begin{aligned} \frac{2\,(x+h)^3 - (x+h) - \left(2x^3 - x\right)}{h} &= -1 + 2h^2 + 6hx + 6x^2 && \text{expand/simplify}\\ &\xrightarrow{h\to 0} -1 + 2\cdot 0^2 + 6\cdot 0\cdot x + 6x^2 && \text{take limit}\\ &= -1 + 6x^2 && \text{simplify.} \end{aligned}$$

Tutorial 2: *A local max/min diagram for Bob.*

Consider the function $f(x)$ defined by: $f(x) = 10 + 3x - 4x^2 + x^4 + \dfrac{x^5}{5}$.

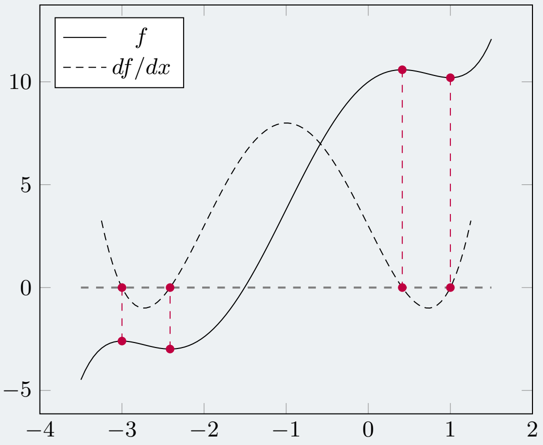
Note that:

$$f'(x) = 3 - 8x + 4x^3 + x^4.$$

The roots to $f'(x) = 0$ equation are:

$$1, \quad -3, \quad -1+\sqrt{2}, \quad -1-\sqrt{2}.$$

Recall: $f'(x_0)$ measures the slope of the tangent line to $y = f(x)$ at $x = x_0$. The values r where $f'(r) = 0$ correspond to places where the slope of the tangent line to $y = f(x)$ is horizontal (see the illustration). This gives us a method for identifying locations where the graph $y = f(x)$ attains a peak (local maximum) or a valley (local minimum).



Tutorial 3: *A limit definition of the derivative for Charlie.*

Let $f(x) = \frac{x}{x^2+1}$. We wish to compute the derivative of $f(x)$ at x using the limit definition of the derivative. Toward that end, we start with the appropriate difference quotient:

$$\begin{aligned} \frac{\frac{x+h}{(x+h)^2+1} - \frac{x}{x^2+1}}{h} &= \frac{(x+h)\left(x^2+1\right) - \left((x+h)^2+1\right)x}{h\left((x+h)^2+1\right)\left(x^2+1\right)} && \text{get a common denominator}\\ &= \frac{h - h^2x - hx^2}{h\left((x+h)^2+1\right)\left(x^2+1\right)} && \text{simplify the numerator}\\ &= \frac{h\left(1 - hx - x^2\right)}{h\left((x+h)^2+1\right)\left(x^2+1\right)} && \text{factor numerator}\\ &= \frac{\left(1 - hx - x^2\right)}{\left(1+x^2\right)\left(1+(h+x)^2\right)} && \text{cancel the } hs\\ &\xrightarrow{h\to 0} \frac{\left(1-x^2\right)}{\left(1+x^2\right)^2} && \text{take limit.} \end{aligned}$$

3.1 Tutorial 1: Limit Definition of the Derivative

Alice is teaching calculus, and she wants to give her students many examples of the dreaded *limit definition of the derivative*. On the other hand, she'd like to avoid working out many examples by-hand. She decides to give `luacas` a try.

Alice can access the `luacas` program using a custom environment: `\begin{CAS}..\end{CAS}`. The first thing Alice must do is declare variables that will be used going forward:

```
\begin{CAS}
    vars('x','h')
\end{CAS}
```

Alice decides that f, the function to be differentiated, should be x^2. So Alice makes this assignment with:

```
\begin{CAS}
    vars('x','h')
    f = x^2
\end{CAS}
```

Now, Alice wants to use the variable q to store the appropriate *difference quotient* of f. Alice could hardcode this into q, but that seems to defeat the oft sought after goal of reusable code. So Alice decides to use the `substitute` command of `luacas`:

```
\begin{CAS}
    vars('x','h')
    f = x^2
    subs = {[x]=x+h}
    q = (substitute(subs,f) - f)/h
\end{CAS}
```

Alice is curious to know if q is what she thinks it is. So Alice decides to have LaTeX print out the contents of q within her document. For this, she uses the `\print` command.

```
\[ \print{q} \]
```

$$\frac{(x+h)^2 - x^2}{h}$$

So far so good! Alice wants to expand the numerator of q; she finds the aptly named `expand` method helpful in this regard. Alice redefines `q` to be `q=expand(q)`, and prints the result to see if things worked as expected:

```
\begin{CAS}
    vars('x','h')
    f = x^2
    subs = {[x]=x+h}
    q = (substitute(subs,f)-f)/h
    q = expand(q)
\end{CAS}
\[ \print{q} \]
```

$$h + 2x$$

Alice is pleasantly surprised that the result of the expansion has been *simplified*, i.e., the factors of x^2 and $-x^2$ cancelled each other out, and the resulting extra factor of h has been cancelled out of the numerator and denominator.

Finally, Alice wants to take the limit as $h \to 0$. Now that our difference quotient has been expanded and simplified, this amounts to another substitution:

```
\begin{CAS}
    vars('x','h')
    f = x^2
    subs = {[x]=x+h}
    q = (substitute(subs,f)-f)/h
    q = expand(q)
    subs = {[h] = 0}
    q = substitute(subs,q)
\end{CAS}
\[ \print{q} \]
```

$$0 + 2x$$

Alice is slightly disappointed that $0 + 2x$ is returned and not $2x$. Alice takes a guess that there's a `simplify` command. This does the trick: adding the line `q = simplify(q)` before leaving the `CAS` environment returns the expected $2x$:

```
\begin{CAS}
    vars('x','h')
    f = x^2
    subs = {[x]=x+h}
    q = (substitute(subs,f)-f)/h
    q = expand(q)
    subs = {[h] = 0}
    q = substitute(subs,q)
    q = simplify(q)
\end{CAS}
\[ \print{q} \]
```

$$2x$$

Alternatively, Alice could have used the `\print*` command instead of `\print` – the essential difference is that `\print*`, unlike `\print`, automatically simplifies the content of the argument.

Alice is pretty happy with how everything is working, but she wants to be able to typeset the individual steps of this process. Alice is therefore thrilled to learn that the `\begin{CAS}..\end{CAS}` environment is very robust – it can:

- Be entered into and exited out of essentially anywhere within her LaTeX document, for example, within `\begin{aligned}..\end{aligned}`; and
- CAS variables persist – if Alice assigns `f = x^2` within `\begin{CAS}..\end{CAS}`, then the CAS remembers that `f = x^2` the next time Alice enters the CAS environment.

Here's Alice's completed code:

```
\begin{CAS}
    vars('x','h')
    f = x^2
\end{CAS}
Let $f(x) = \print{f}$. We wish to compute the derivative of $f(x)$ at $x$ using thelimit
↪   definition of the derivative. Toward that end, we start with the appopriatedifference
↪   quotient:
\begin{CAS}
    subs = {[x]=x+h}
    q = (substitute(subs,f) - f)/h
\end{CAS}
\[ \begin{aligned}
    \print{q} &=
    \begin{CAS}
```

```
        q = expand(q)
    \end{CAS}
    \print{q}& &\text{expand/simplify} \\
    \begin{CAS}
        subs = {[h]=0}
        q = substitute(subs,q)
    \end{CAS}
    &\xrightarrow{h\to 0} \print{q}& &\text{take limit}\\
    &=
    \begin{CAS}
        q = simplify(q)
    \end{CAS}
    \print{q} & &\text{simplify.}
\end{aligned} \]
So $\print{diff(f,x)} = \print*{diff(f,x)}$.
```

Alice can produce another example merely by changing the definition of f on the third line to another polynomial:

```
\begin{CAS}
    vars('x','h')
    f = 2*x^3-x
\end{CAS}
```

And here is Alice's completed project:

Tutorial 1: *A limit definition of the derivative for Alice.*

Let $f(x) = 2x^3 - x$. We wish to compute the derivative of $f(x)$ at x using the limit definition of the derivative. Toward that end, we start with the appropriate difference quotient:

$$\begin{aligned}
\frac{2\left(x+h\right)^3-\left(x+h\right)-\left(2x^3-x\right)}{h} &= -1+2h^2+6hx+6x^2 & &\text{expand/simplify} \\
&\xrightarrow{h\to 0} -1+2\cdot 0^2+6\cdot 0\cdot x+6x^2 & &\text{take limit} \\
&= -1+6x^2 & &\text{simplify.}
\end{aligned}$$

3.2 Tutorial 2: Finding Maxima/Minima

Bob is teaching calculus too, and he wants to give his students many examples of the process of *finding the local max/min of a given function*. But, like Alice, Bob doesn't want to work out a bunch of examples by-hand. Bob decides to try his hand with `luacas` after having been taught the basics by Alice.

Bob decides to stick with polynomials for these examples; if anything because those functions are in the wheel-house of `luacas`. In particular, Bob decides that the *derivative* of the function he wants to use should be a composition of quadratics. This ought to ensure that the roots of that derivative are expressible in a nice way.

Accordingly, Bob declares variables and chooses two quadratic polynomials to compose, say f and g, and sets $dh = g \circ f$:

```
\begin{CAS}
    vars('x')
    f = x^2+2*x-2
    g = x^2-1
    subs = {[x] = f}
    dh = substitute(subs,g)
\end{CAS}
```

Bob wants to compute h, the integral of dh. Bob could certainly compute this quantity by-hand, but why hardcode that information into the document when `luacas` can do this for you? So Bob uses the `int` command and shifts the result (with some malice aforethought):

```
\begin{CAS}
    h = int(dh,x) + 10
\end{CAS}
```

Bob is curious to know the value of h. So he uses `\print{h}` to produce:

```
\[ \print{h} \]
```

$$\int \left(x^2+2x-2\right)^2 - 1\, dx + 10$$

This isn't exactly what Bob had in mind. It occurs to Bob that he may need to simplify the expression h, so he tries:

```
\begin{CAS}
    h = simplify(int(dh,x)+10)
\end{CAS}
\[ \print{h} \]
```

$$10 + 3x - 4x^2 + x^4 + \frac{x^5}{5}$$

That's more like it! Now, Bob wants to find the roots to dh. Bob uses the `roots` command to do this:

```
\begin{CAS}
    r = roots(dh)
\end{CAS}
```

But then Bob wonders to himself, "How do I actually retrieve the roots of dh from `luacas`?" The assignment `r = roots(dh)` stores the roots of the polynomial dh in a table named `r`:

```
\[ \print{r[1]}, \quad \print{r[2]}, \quad \print{r[3]}, \quad\print{r[4]} \]
```

$$1, \quad -3, \quad -1+\sqrt{2}, \quad -1-\sqrt{2}$$

If Bob truly wants to print the entire list `r`, Bob can use the `\lprint` (list **print**) command:

```
\[ \left\{ \lprint{r} \right\} \]
```

$$\left\{1, -3, -1+\sqrt{2}, -1-\sqrt{2}\right\}$$

Splendid! Bob would now like to evaluate the function h at these roots (for these are the local max/min values of h). Here's Bob's first thought:

```
\begin{CAS}
  v = simplify(substitute({[x]=r[1]},h))
\end{CAS}
\[ \print{v} \]
```

$$10+3x-4x^2+x^4+\frac{x^5}{5}$$

What the heck?! Bob is (understandably) confused. But here's where Bob learns a valuable lesson...

3.2.1 A brief interlude: Lua numbers vs `luacas` `Integers`

The LaTeX environment `\begin{CAS}..\end{CAS}` is really a glorified Lua environment. The "glory" comes in how the contents of the environment are parsed in a special manner to make interacting with the CAS (mostly) easy. Bob has encountered a situation where that interaction is not as easy as we'd like.

For comparison, consider the following:

Here's some code using the `\begin{CAS}..\end{CAS}`:

```
\begin{CAS}
    vars('y')
    a = 1
    b = y+a
\end{CAS}
\[ \print{b} \]
```

$$y+1$$

Here's that same code but using `\directlua` instead:

```
\directlua{
    vars('y')
    a = Integer(1)
    b = y+a
}
\[ \print{b} \]
```

$$y+1$$

The essential difference being:

- Using `\begin{CAS}..\end{CAS}`, a parser automatically interprets any digit strings as an `Integer`; this is a special class defined within the bowels of `luacas`. Ultimately, it allows for us to define things like the addition of an `Integer` and an `Expression` (in this case, the result is a new `Expression`) as well as arbitrary precision arithmetic.
- Using `\directlua`, there is no parsing, so the user (aka Bob) is responsible for telling `luacas` what to interpret as an `Integer` versus what to interpret as a normal Lua `number`.

Generally speaking, we like what the parser in `\begin{CAS}..\end{CAS}` does: it keeps us from having to wrap all integers in `Integer(..)` (among other things). But the price we pay is that the parser indiscriminately wraps *all* (or rather, most) digit strings in `Integer(..)`. This causes a problem in the following line in Bob's code:

```
v = simplify(substitute({[x]=r[1]},h))
```

The parser sees `r[1]` and interprets `1` as `Integer(1)` – but `r[Integer(1)]` is `nil`, so no substitution is performed.

The good news is that, excluding the annoyance between `Integer` and Lua number, interacting with the CAS via `\directlua` is not much different than interacting with it via `\begin{CAS}..\end{CAS}`.

Back to the tutorial...

After that enlightening interlude, Bob realizes that some care needs to be taken when constructing tables. Here's a solution from within `\begin{CAS}..\end{CAS}`:

```
\begin{CAS}
    r = ZTable(r)
    v = ZTable()
    for i in range(1, 4) do
        v[i] = simplify(substitute({[x]=r[i]},h))
    end
\end{CAS}
\[ \left\{ \lprint{v} \right\} \]
```

$$\left\{ \frac{51}{5}, -\frac{13}{5}, \frac{19}{5} + \frac{24\sqrt{2}}{5}, \frac{19}{5} - \frac{24\sqrt{2}}{5} \right\}$$

The function `ZTable()` sets indices appropriately for use within `\begin{CAS}..\end{CAS}` while the function `range()` protects the bounds of the for-loop. Alternatively, Bob can make tables directly within `\directlua` (or `\luaexec` from the `luacode` package) using whatever Lua syntax pleases him:

```
\directlua{
    v = {}
    for i=1,4 do
        table.insert(v,simplify(substitute({[x]=r[i]},h)))
    end}
\[ \left\{ \lprint{v} \right\} \]
```

$$\left\{ \frac{51}{5}, -\frac{13}{5}, \frac{19}{5} + \frac{24\sqrt{2}}{5}, \frac{19}{5} - \frac{24\sqrt{2}}{5} \right\}$$

Great! But still; Bob doesn't want to just pretty-print the roots of dh (or the values that h takes at those roots). Bob is determined to plot the results – he wants to hammer home the point that the roots of dh point to the local extrema of h.

Luckily, Bob is familiar with some of the fantastic graphics tools in the LaTeX ecosystem, like `pgfplots` and `asymptote`. But then Bob begins to wonder, "How can I yoink results out of `luacas` so that I may yeet them into something like `pgfplots`?" Bob is delighted to find the following commands: `\fetch` and `\store`.

Whereas the `\print` command relies on the `luacas` method `tolatex()`, the commands `\fetch` and `\store` rely on the `luacas` function `tostring()`. Bob can view the output of `tostring()` using the `\vprint` command (**v**erbatim **print**). For example, `\vprint{h}` produces:

```
10 + (3 * x) + (-4 * (x ^ 2)) + (x ^ 4) + (1/5 * (x ^ 5))
```

This is more-or-less what Bob wants – but he doesn't want the verbatim output printed to his document, Bob just wants the contents of `tostring(h)`. Here's where `\fetch` comes in. The command `\fetch{h}` is equivalent to:

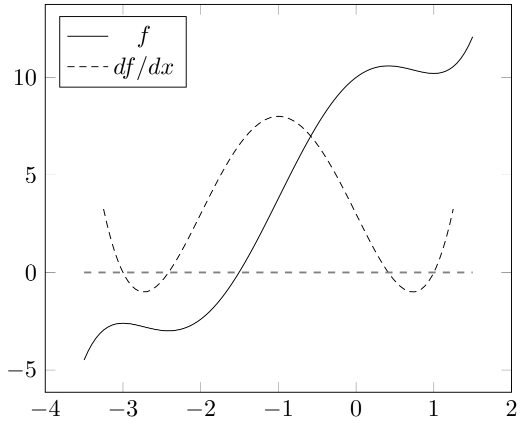
```
\directlua{
    tex.print(tostring(h))
}
```

For comparison, the command `\print{h}` is equivalent to:

```
\directlua{
    tex.print(h:tolatex())
}
```

For Bob's purposes, `\fetch{h}` is exactly what he needs:

```
\begin{tikzpicture}[scale=0.9]
  \begin{axis}[legend pos = north west]
    \addplot [domain=-3.5:1.5,samples=100]
      {\fetch{h}};
    \addlegendentry{$f$};
    \addplot[densely dashed]
      [domain=-3.25:1.25,samples=100]
      {\fetch{dh}};
    \addlegendentry{$df/dx$};
    \addplot[gray,dashed,thick]
      [domain=-3.5:1.5] {0};
  \end{axis}
\end{tikzpicture}
```



Alternatively, Bob could use `\store`. The `\store` command will *fetch* the contents of its mandatory argument and store it in a macro of the same name.

```
\store{h}
\store{dh}
```

Now the macros `\h` and `\dh` can be used in place of `\fetch{h}` and `\fetch{dh}`, respectively. An optional argument can be used to store contents in a macro under a different name. This is useful for situations like the following:

```
\store{r[1]}[rootone]
```

Now `\rootone` can be used in place of `\fetch{r[1]}`. But Bob wants to fetch all the values stored in `r` (and `v`, for that matter). In this case, Bob can use:

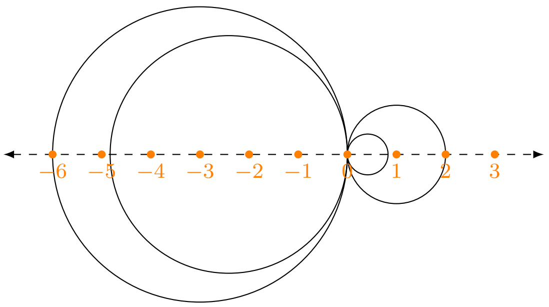
```
\store{r}
\store{v}
```

The command `\store{r}` is equivalent to:

```
\def\r{{ \fetch{r[1]}, \fetch{r[2]}, \fetch{r[3]}, \fetch{r[4]} }}
```

The contents of the LaTeX macro `\r` can be accessed with `\pgfmathsetmacro`. For example:

```
\begin{tikzpicture}[scale=0.6]
  \draw [dashed,latex-latex]
    (-7,0) -- (4,0);
  \foreach \k in {0,1,2,3}{
    \pgfmathsetmacro\a{\r[\k]}
    \draw (\a,0) circle (\a);
  }
  \foreach \x in {-6,...,3}{
    \draw[fill,orange]
      (\x,0) circle (2pt)
      node[below] {\footnotesize$\x$};
  }
\end{tikzpicture}
```



Alternatively, Bob could avoid the call to `\pgfmathsetmacro` by replacing lines 5-6 in the above code with the slightly more verbose:

```
\draw ({\fetch{r[\k]}},0) circle (\fetch{r[\k]});
```

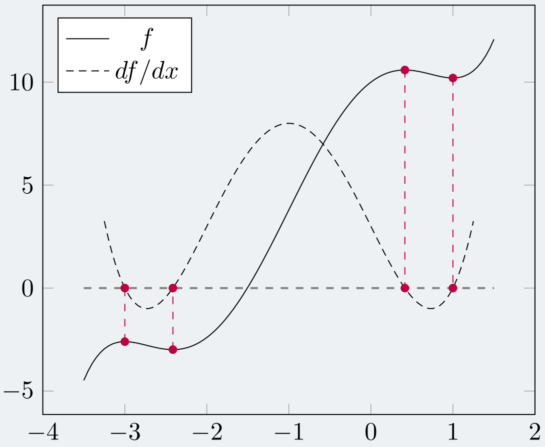
Alternatively still, Bob could appeal directly to the `tostring()` function in `luacas` and iterate over tables like `r` using Lua itself. This can often be a simpler solution (particularly when working within `\begin{axis}..\end{axis}`), and it is exactly what Bob does in his complete project shared below:

```
Consider the function $f(x)$ defined by:
\begin{CAS}
  vars('x')
  f = x^2+2*x-2
  g = x^2-1
  subs = {[x] = f}
  dh = expand(substitute(subs,g))
  h = simplify(int(dh,x)+10)
\end{CAS}
$\displaystyle f(x) = \print{h}$.
\begin{multicols}{2}
  Note that:
  \[ f'(x) = \print{dh}.\]
  The roots to $f'(x)=0$ equation are:
  \begin{CAS}
      r = roots(dh)
  \end{CAS}
  \[ \left\{ \lprint{r} \right\} \]
  Recall: $f'(x_0)$ measures the slope of the tangent line to $y=  (x)$ at $x=x_0$. The
  ↪  values $r$ where $f'(r)=0$ correspond to  places where the slope of the tangent line to
  ↪  $y=f(x)$ is horizontal (see the illustration). This gives us a method for identifying
  ↪  locations where the graph $y=f(x)$ attains a peak  (local maximum) or a valley (local
  ↪  minimum).
  \begin{CAS}
    r = ZTable(r)
    v = ZTable()
    for i in range(1, 4) do
        v[i] = simplify(substitute({[x]=r[i]},h))
    end
  \end{CAS}
  \columnbreak
  \store{h}\store{dh}
  \begin{tikzpicture}[scale=0.95]
    \begin{axis}[legend pos = north west]
      \addplot [domain=-3.5:1.5,samples=100] {\h};
      \addlegendentry{$f$};
      \addplot[densely dashed] [domain=-3.25:1.25,samples=100] {\dh};
      \addlegendentry{$df/dx$};
      \addplot[gray,dashed,thick] [domain=-3.5:1.5] {0};
      \luaexec{for i=1,4 do
        tex.print("\\draw[fill=purple,purple]",
          "(axis cs:{",tostring(r[i]),"},0) circle (1.5pt)",
          "(axis cs:{",tostring(r[i]),"},{",tostring(v[i]),"}) circle (1.5pt)",
          "(axis cs:{",tostring(r[i]),"},{",tostring(v[i]),"}) edge[dashed] (axis
          ↪  cs:{",tostring(r[i]),"},0);")
        end}
    \end{axis}
  \end{tikzpicture}
\end{multicols}
```

And here is Bob's completed project:

Tutorial 2: *A local max/min diagram for Bob.*

Consider the function $f(x)$ defined by: $f(x) = 10 + 3x - 4x^2 + x^4 + \dfrac{x^5}{5}$.



Note that:

$$f'(x) = 3 - 8x + 4x^3 + x^4.$$

The roots to $f'(x) = 0$ equation are:

$$\left\{1, -3, -1 + \sqrt{2}, -1 - \sqrt{2}\right\}$$

Recall: $f'(x_0)$ measures the slope of the tangent line to $y = f(x)$ at $x = x_0$. The values r where $f'(r) = 0$ correspond to places where the slope of the tangent line to $y = f(x)$ is horizontal (see the illustration). This gives us a method for identifying locations where the graph $y = f(x)$ attains a peak (local maximum) or a valley (local minimum).

3.3 Tutorial 3: Adding Functionality

Charlie, like Alice and Bob, is also teaching calculus. Charlie likes Alice's examples and wants to try something similar. But Charlie would like to do more involved examples using rational functions. Accordingly, Charlie copy-and-pastes Alice's code:

```
\begin{CAS}
    vars('x','h')
    f = 1/(x^2+1)
    subs = {[x]=x+h}
    q = (substitute(subs,f)-f)/h
    q = expand(q)
\end{CAS}
```

Unfortunately, `\[ q=\print{q} \]` produces:

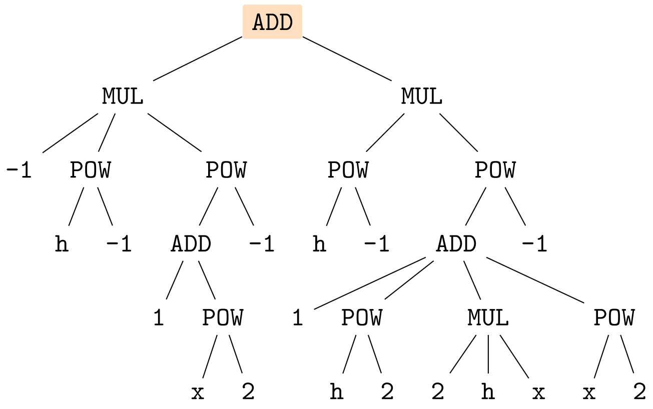
$$q = -\frac{1}{h\left(1+x^2\right)} + \frac{\frac{1}{h}}{1+h^2+2hx+x^2}$$

The `simplify()` command doesn't seem to help either! What Charlie truly needs is to combine terms, i.e., Charlie needs to find a *common denominator*. They're horrified to learn that no such functionality exists in this burgeoning package.

So what's Charlie to do? They could put a feature request in, but they're concerned that the schlubs in charge of managing the package won't get around to it until who-knows-when. So Charlie decides to take matters into their own hands. Besides, looking for that silver lining, they'll learn a little bit about how `luacas` is structured.

At the heart of any CAS is the idea of an `Expression`. Mathematically speaking, an `Expression` is a rooted tree. Luckily, this tree can be drawn using the (wonderful) `forest` package. In particular, the command `\parseforest{q}` will scan the contents of the expression `q` and parse the results into a form compatible with the `forest` package; those results are saved in a macro named `\forestresult`.

```
\parseforest{q}
\bracketset{action character =
↪  @}
\begin{forest}
    for tree = {
        font=\ttfamily,
        rectangle,
        rounded corners=1pt
    },
    where level=0{%
        fill=orange!25
    }{},
    @\forestresult
\end{forest}
```



The root of the tree above is `ADD` since q is, at its heart, the addition of two other expressions. Charlie wonders how they might check to see if a mystery `Expression` is an `ADD`? But this is putting the cart before the horse; Charlie should truly wonder how to check for the *type* of `Expression` – then they can worry about other attributes.

Charlie can print the `Expression` type directly into their document using the `\whatis` command:

```
\begin{CAS}
    r = diff(q,x,h)
\end{CAS}
\whatis{q} vs \whatis{r}
```

`BinaryOperation` vs `DiffExpression`

So `q` is a `BinaryOperation`? This strikes Charlie as a little strange. On the other hand, `q` is the result of a binary operation applied to two other expressions; so perhaps this makes a modicum of sense.

At any rate, Charlie now knows, according to `luacas`, that `q` is of the `Expression`-type `BinaryOperation`. The actual operator that's used to form `q` is stored in the attribute `q.operation`:

```
\luaexec{if q.operation == BinaryOperation.ADD then
    tex.print("I'm an \\texttt{ADD}")
end}
```

I'm an `ADD`

Of course, different `Expression` types have different attributes. For example, being a `DiffExpression`, `r` has the attribute `r.degree`:

```
\luaexec{tex.print("I'm an order", r.degree, "derivative.")}
```

I'm an order 2 derivative.

`BinaryOperation`s have several attributes, but the most important attribute for Charlie's purposes is `q.expressions`. In this case, `q.expressions` is a table with two entries; those two entries are precisely the `Expression`s whose sum forms `q`. In particular,

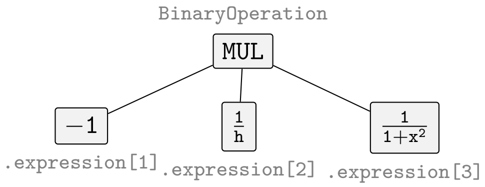
```
\[ \print{q.expressions[1]} \qquad \text{and} \qquad \print{q.expressions[2]} \]
```

produces:

$$-\frac{1}{h\left(1+x^{2}\right)} \qquad \text{and} \qquad \frac{\frac{1}{h}}{1+h^{2}+2hx+x^{2}}$$

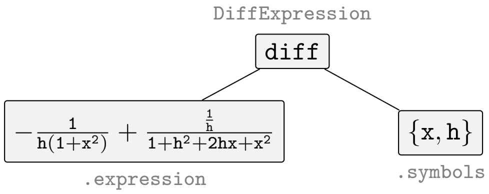
The expression `q.expressions[1]` is another `BinaryOperation`. Instead of printing the entire expression tree (as we've done above), Charlie might be interested in the commands `\parseshrub` and `\shrubresult`:

```
\parseshrub{q}
\begin{forest}
  for tree = {draw,rectangle,rounded
  ↪   corners=1pt,fill=lightgray!20
  ↪   font=\ttfamily}
  @\shrubresult
\end{forest}
```



The "shrub" is essentially the first level of the "forest", but with some extra information concerning attributes. For contrast, here's the result of `\parseshrub` and `\shrubresult` applied to `r`, the `DiffExpression` defined above.

```
\parseshrub{r}
\begin{forest}
    for tree = {draw,rectangle,
    rounded corners=1pt,fill=lightgray!20,
    font=\ttfamily, s sep=1cm}
    @\shrubresult
\end{forest}
```



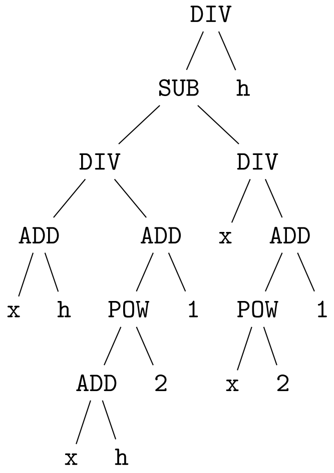
The attribute `r.degree` returns the size of the table stored in `r.symbols` which, in turn, records the variables (and order from left-to-right) with which to differentiate the expression stored in `r.expression`.

Now that Charlie knows the basics of how `luacas` is structured, they're ready to try their hand at adding some functionality.

First, Charlie decides to up the complexity of their expression `f` so that they have something more general to work with:

```
\begin{CAS}
    vars('x','h')
    f = x/(x^2+1)
    subs = {[x]=x+h}
    q = (substitute(subs,f)-f)/h
\end{CAS}
```

Next, Charlie decides to print the un`expand`ed expression tree for q to help give them a clear view (see right).



Charlie now wants to write their own function for combining expressions like this into a single denominator. It's probably best that Charlie writes this function in a separate file, say `myfile.lua`. Like most functions in `luacas`, Charlie defines this function as a *method* applied to an `Expression`:

```
function Expression:mycombine()
```

Next, Charlie declares some local variables to identify appropriate numerators and denominators:

```
    local a = self.expressions[1].expressions[1].expressions[1]
    local b = self.expressions[1].expressions[1].expressions[2]
    local c = self.expressions[1].expressions[2].expressions[1]
    local d = self.expressions[1].expressions[2].expressions[2]
```

So, for example, $a = x + h$, $b = (x + h)^2 + 1$, and so on. Charlie now forms the numerator and denominator, and returns the function:

```
    local numerator = a*d-b*c
    local denominator = self.expressions[2]*b*d
    return numerator/denominator
end
```

Now Charlie only needs to ensure that `myfile.lua` is in a location visible to their TeX installation (e.g. in the current working folder). Charlie can then produce the following:

```
\directlua{dofile('myfile.lua')}
\begin{CAS}
    q = q:mycombine()
\end{CAS}
\[ \print{q} \]
```

$$\frac{(x+h)\left(x^2+1\right)-\left((x+h)^2+1\right)x}{h\left((x+h)^2+1\right)\left(x^2+1\right)}$$

Charlie wants to simplify the numerator (but not the denominator). So they decide to write another function in `myfile.lua` that does precisely this:

```
function Expression:mysimplify()
    local a = self.expressions[1]
    local b = self.expressions[2]
    a = simplify(a)
    return a/b
end
```

Now Charlie has:

```
\begin{CAS}
    q = q:mysimplify()
\end{CAS}
\[ \print{q} \]
```

$$\frac{h-h^2x-hx^2}{h\left((x+h)^2+1\right)(x^2+1)}$$

Finally, Charlie wants to factor the numerator. So Charlie writes the following final function to `myfile.lua`:

```
function Expression:myfactor()
    local a = self.expressions[1]
    local b = self.expressions[2]
    a = factor(a)
    return a/b
end
```

```
After factoring the numerator:
\begin{CAS}
    q = q:myfactor()
\end{CAS}
\[ \print{q} \]
And then simplifying:
\begin{CAS}
    q = simplify(q)
\end{CAS}
\[ \print{q} \]
```

After factoring the numerator:

$$\frac{h\left(1-hx-x^2\right)}{h\left((x+h)^2+1\right)(x^2+1)}$$

And then simplifying:

$$\frac{\left(1-hx-x^2\right)}{(1+x^2)\left(1+(h+x)^2\right)}$$

Armed with their custom functions `mycombine`, `mysimplify`, and `myfactor`, Charlie can write examples just like Alice's examples, but using rational functions instead.

Of course, the schlubs that manage this package feel for Charlie, and recognize that there are other situations in which folks may want to combine a sum of rational expressions into a single rational expression. Accordingly, there is indeed a `combine` command included in `luacas` that performs this task:

```
\begin{CAS}
    vars('x','y','z')
    a = y/z
    b = z/x
    c = x/y
    d = combine(a+b+c)
\end{CAS}
\[ \print{a+b+c} = \print{d} \]
```

$$\frac{y}{z}+\frac{z}{x}+\frac{x}{y}=\frac{xy^2+x^2z+yz^2}{xyz}$$

Here's Charlie's complete code (but using `\directlua`) instead:

```
\begin{CAS}
    vars('x','h')
    f = x/(x^2+1)
\end{CAS}
```

```
Let $f(x) = \print{f}$. We wish to compute the derivative of $f(x)$ at $x$ using the limit
↪  definition of the derivative. Toward that end, we start with the appropriate difference
↪  quotient:
\begin{CAS}
    subs = {[x] = x+h}
    q = (f:substitute(subs) - f)/h
\end{CAS}
\directlua{
```

And now the Lua code:

```
    function Expression:mycombine()
        local a = self.expressions[1].expressions[1].expressions[1]
        local b = self.expressions[1].expressions[1].expressions[2]
        local c = self.expressions[1].expressions[2].expressions[1]
        local d = self.expressions[1].expressions[2].expressions[2]
        local numerator = a*d-b*c
        local denominator = self.expressions[2]*b*d
        return numerator/denominator
    end
    function Expression:mysimplify()
        local a = self.expressions[1]
        local b = self.expressions[2]
        a = simplify(a)
        return a/b
    end
    function Expression:myfactor()
        local a = self.expressions[1]
        local b = self.expressions[2]
        a = factor(a)
        return a/b
    end
```

And now back to the LaTeX code:

```
}
\[ \begin{aligned}
    \print{q} &=
    \begin{CAS}
        q = q:mycombine()
    \end{CAS}
    \print{q}& &\text{get a common denominator} \\
    &=
    \begin{CAS}
        q = q:mysimplify()
    \end{CAS}
    \print{q}& &\text{simplify the numerator} \\
    &=
    \begin{CAS}
        q = q:myfactor()
    \end{CAS}
    \print{q} & &\text{factor numerator} \\
    &=
    \begin{CAS}
        q = simplify(q)
    \end{CAS}
    \print{q}& &\text{cancel the $h$s} \\
    &\xrightarrow{h\to 0}
    \begin{CAS}
        subs = {[h] = 0}
        q = substitute(subs,q):autosimplify()
    \end{CAS}
    \print{q}& &\text{take limit.}
\end{aligned} \]
```

And here is Charlie's completed project:

Tutorial 3: *A limit definition of the derivative for Charlie.*

Let $f(x) = \frac{x}{x^2+1}$. We wish to compute the derivative of $f(x)$ at x using the limit definition of the derivative. Toward that end, we start with the appropriate difference quotient:

$$\begin{aligned}
\frac{\frac{x+h}{(x+h)^2+1} - \frac{x}{x^2+1}}{h} &= \frac{(x+h)\left(x^2+1\right) - \left((x+h)^2+1\right)x}{h\left((x+h)^2+1\right)\left(x^2+1\right)} && \text{get a common denominator} \\
&= \frac{h - h^2x - hx^2}{h\left((x+h)^2+1\right)\left(x^2+1\right)} && \text{simplify the numerator} \\
&= \frac{h\left(1 - hx - x^2\right)}{h\left((x+h)^2+1\right)\left(x^2+1\right)} && \text{factor numerator} \\
&= \frac{\left(1 - hx - x^2\right)}{\left(1+x^2\right)\left(1+(h+x)^2\right)} && \text{cancel the hs} \\
&\xrightarrow{h\to 0} \frac{\left(1-x^2\right)}{\left(1+x^2\right)^2} && \text{take limit.}
\end{aligned}$$

Part II

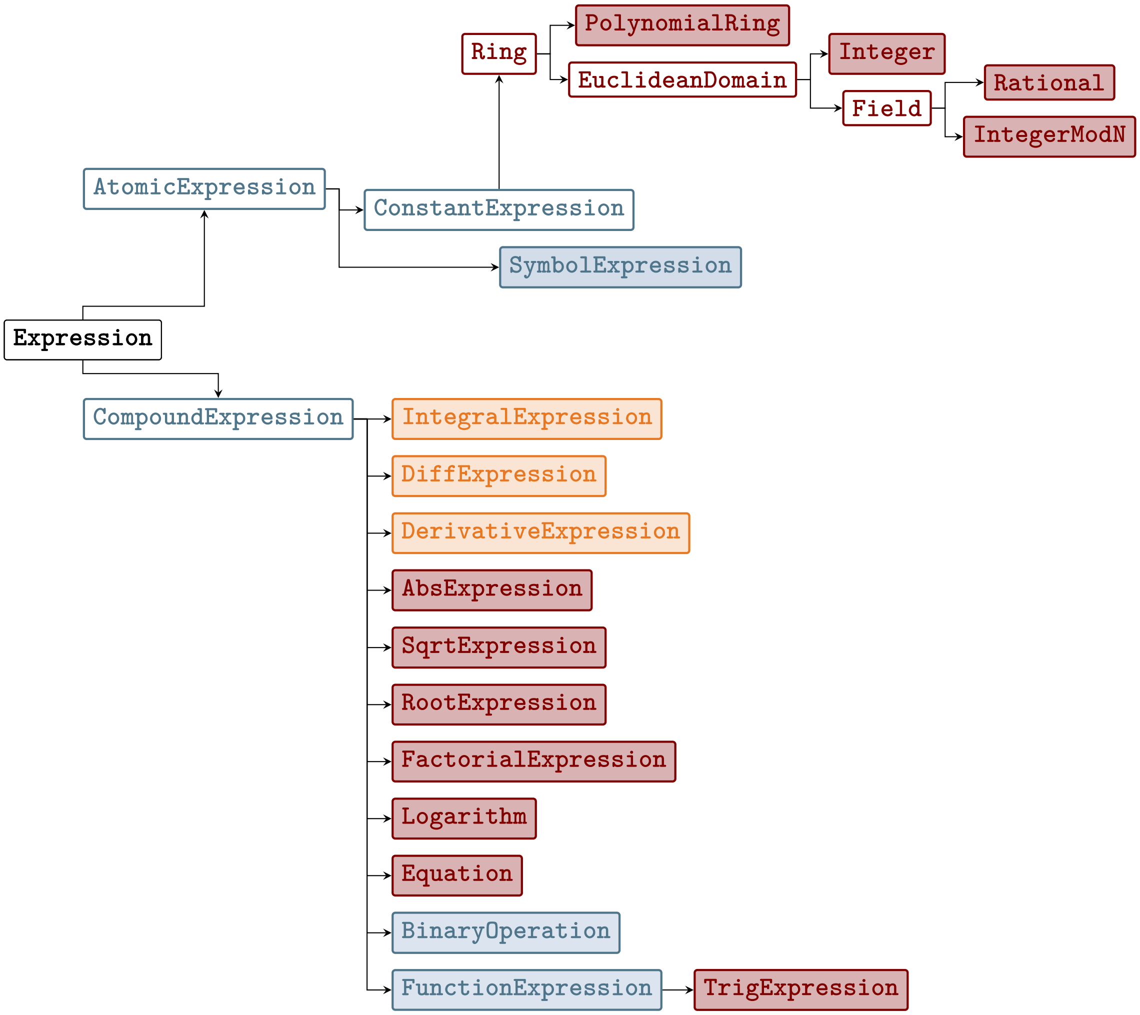
Reference

This part contains reference material for the classes and methods that incorporate the `luacas` package. Some classes are *concrete* while others are *abstract*. The concrete classes are essentially the objects that a user might reasonably interact with while using `luacas`. Thankfully, most of this interaction will be filtered through a rudimentary (but functional!) parser. Abstract classes exist for the purposes of inheritance.

The classes in the diagram below are color-coded according to:

- (`Class`) `Class`: a (concrete) class belonging to the core module;
- (`Class`) `Class`: a (concrete) class belonging to the algebra module;
- (`Class`) `Class`: a (concrete) class belonging to the calculus module.

Inheritance is indicated with an arrow:



Every object in `luacas` is an expression, meaning it inherits from the `Expression` type (class). Since the `Expression` type itself has no constructor and cannot be instantiated, it it closer to an interface in Java OOP terms.[1] `Expressions` can store any number of other expressions as sub-expressions, depending on type.

---

[1] In reality, interfaces are unnecessary in Lua due to its weak typing - Lua doesn't check whether an object has a method at

This means that `Expression` objects are really trees. Types that inherit from `Expression` that can not store other expressions are called *atomic expressions*, and correspond to the leaf nodes of the tree. Other expression types are *compound expressions*. Thus, every `Expression` type inherits from one of `AtomicExpression` or `CompoundExpression`. The `ConstantExpresssion` interface is a subinterface to `AtomicExpression`. Types that inherit from `ConstantExpression` roughly correspond to numbers (interpreted broadly).

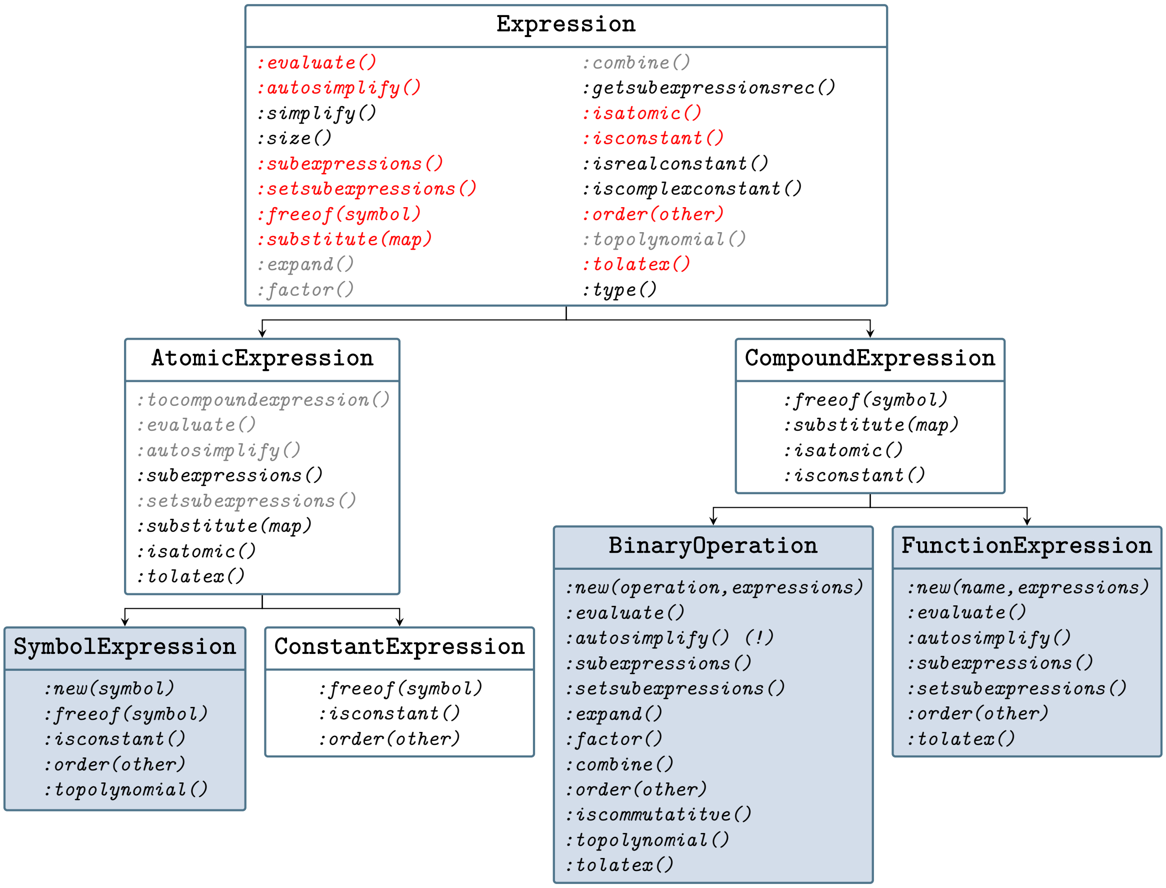
---

compile time. The `Expression` type is really an abstract class in Java terms.

4 Core

This section contains reference materials for the core functionality of `luacas`. The classes in this module are diagramed below according to inheritance along with the methods/functions one can call upon them.

- *`method`*: an abstract method (a method that must be implemented by a subclass to be called);
- *`method`*: a method that returns the expression unchanged;
- *`method`*: a method that is either unique, implements an abstract method, or overrides an abstract method;
- `Class` : a concrete class.



The number of core methods should generally be kept small, since every new type of expression must implement all of these methods. The exception to this, of course, is core methods that call other core methods that provide a unified interface to expressions. For instance, `size()` calls `subexpressions()`, so it only needs to be implemented in the expression interface.

All expressions should also implement the `__tostring` and `__eq` metamethods. Metamethods cannot be inherited using Lua, thus every expression object created by a constructor must assign a metatable to that object.

- `__tostring` provides a human-readable version of an expression for printing within Lua and exporting to external programs.
- `__eq` determines whether an expression is structurally identical to another expression.

4.1 Core Classes

There are several classes in the core module; but only some classes are concrete:

Abstract classes:

- `Expression`
- `AtomicExpression`
- `CompoundExpression`
- `ConstantExpression`

Concrete classes:

- `SymbolExpression`
- `BinaryOperation`
- `FunctionExpression`

The abstract classes provide a unified interface for the concrete classes (expressions) using inheritance. *Every* expression in `luacas` inherits from either `AtomicExpression` or `CompoundExpression` which, in turn, inherit from `Expression`.

`function SymbolExpression:new(string)` `return SymbolExpression`

Creates a new `SymbolExpression`. For example:

```
foo = SymbolExpression("bar")
tex.sprint("The Lua variable ``foo'' is the SymbolExpression: ", foo:tolatex(),".")
```

The Lua variable 'foo' is the SymbolExpression: bar.

Fields

`SymbolExpressions` have only one field: `symbol`. In the example above, the string `"bar"` is stored in `foo.symbol`.

Parsing

The command `vars()` in `test.parser` creates a new `SymbolExpression` for every string in the argument; each such `SymbolExpression` is assigned to a variable of the same name. For example:

```
vars('x','y')
```

is equivalent to:

```
x = SymbolExpression("x")
y = SymbolExpression("y")
```

`operation function, expressions table<number,Expression>`

`function BinaryOperation:new(operation, expressions)` `return BinaryOperation`

Creates a new `BinaryOperation` expression. For example:

```
vars('x','y','z')
w = BinaryOperation(
    BinaryOperation.ADD,
    {BinaryOperation(
        BinaryOperation.MUL,
        {x,y}
    ),y,z}
)
tex.print("\\[w=",w:tolatex(),"\\]")
```

$$w = xy + y + z$$

The variable `operation` must be a function `function f(a,b)` assigned to one of the following types:

`BinaryOperation.ADD:` `return a + b`

`BinaryOperation.SUB:` `return a - b`

`BinaryOperation.MUL:` `return a * b`

`BinaryOperation.DIV:` `return a / b`

`BinaryOperation.IDIV:` `return a // b`

`BinaryOperation.MOD:` `return a % b`

`BinaryOperation.POW:` `return a ^ b`

The variable `expressions` must be a table of `Expression`s index by Lua numbers.

Fields

`BinaryOperation`s have the following fields: `name`, `operation`, and `expressions`. In the example above, we have:

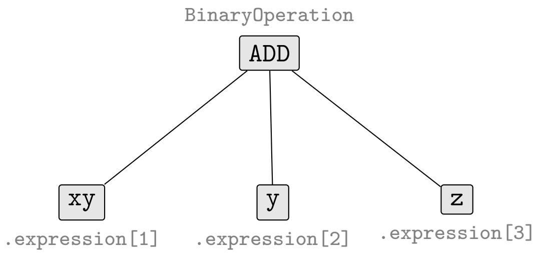
- the variable `expressions` is stored in `w.expressions`;
- `w.name` stores the string `"+"`; and
- `w.operation` stores the function:

  ```
  BinaryOperation.ADD = function(a, b)
      return a + b
  end
  ```

The entries of `w.expressions` can be used/fetched in a reasonable way:

```
$\print{w.expressions[1]} \quad
 \print{w.expressions[2]} \quad
 \print{w.expressions[3]}$
```

$xy \quad y \quad z$



Parsing

Thank goodness for this. Creating new `BinaryOperation`s isn't nearly as cumbersome as the above would indicate. Using Lua's powerful metamethods, we can parse expressions easily. For example, the construction of `w` given above can be done much more naturally using:

```
vars('x','y','z')
w = x*y+y+z
tex.print("\\[w=", w:tolatex(), "\\]")
```

$$w = xy + y + z$$

Warning: There are escape issues to be aware of with the operator %. If you're writing custom `luacas` functions in a separate `.lua` file, then there are no issues; use % with reckless abandon. But when using the operator % within, say `\begin{CAS}..\end{CAS}`, then one should write `\%` in place of %:

```
\begin{CAS}
    a = 17
    b = 5
    c = a \% b
\end{CAS}
\[ \print{c} \equiv \print{a}
↪   \bmod{\print{b}} \]
```

$$2 \equiv 17 \bmod 5$$

The above escape will **not** work with `\directlua`, but it will work for `\luaexec` from the `luacode` package. Indeed, the `luacode` package was designed (in part) to make escapes like this more manageable. Here is the equivalent code using `\luaexec`:

```
a = Integer(17)
b = Integer(5)
c = a \% b
tex.print("\\[",c:tolatex(),"\\equiv",a:tolatex(), "\\bmod{",b:tolatex(),"} \\]")
```

$$2 \equiv 17 \bmod 5$$

```
                                   name string|SymbolExpression, expressions table<number,Expression>
function FunctionExpression:new(name,expressions)                          return FunctionExpression
```

Creates a generic function. For example:

```
vars('x','y')
f = FunctionExpression('f',{x,y})
tex.print("\\[",f:tolatex(),"\\]")
```

$$f(x, y)$$

The variable `name` can be a string (like above), or another `SymbolExpression`. But in this case, the variable `name` just takes the value of the string `SymbolExpression.symbol`. The variable `expressions` must be a table of `Expression`s indexed by Lua numbers.

Fields

`FunctionExpression`s have the following fields: `name`, `expressions`, `variables`, `derivatives`. In the example above, we have:

- the variable `name`, i.e. the string `'f'`, is stored in `f.name`; and
- the variable `expressions`, i.e. the table `{x,y}` is stored in `f.expressions`.

Wait a minute, what about `variables` and `derivatives`!? The field `variables` essentially stores a copy of the variable `expressions` *as long as* the entries in that table are atomic. If they aren't, then `variables` will default to x, y, z, or $x_1, x_2, \ldots$ if the number of variables exceeds 3. For example:

```
vars('s','t')
f = FunctionExpression('f',{s*s,s+t+t})
tex.print("The variables of f are:")
for _,symbol in ipairs(f.variables) do
    tex.print(symbol:tolatex())
end
```

The variables of f are: x y

The field `derivatives` is a table of `Integer`s indexed by Lua numbers whose length equals `#o.variables`. The default value for this table is a table of (`Integer`) zeros. So for the example above, we have:

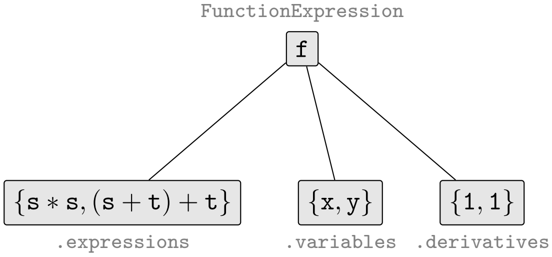
```
for _,integer in ipairs(f.derivatives) do
  if integer == Integer.zero() then
    tex.print("I'm a zero.\\newline")
  end
end
```

```
I'm a zero.
I'm a zero.
```

We can change the values of `variables` and `derivatives` manually (or more naturally by other gizmos found in `luacas`). For example, keeping the variables from above, we have:

```
f.derivatives = {Integer.one(),
    Integer.one()}
tex.print("\\[",
    f:simplify():tolatex(),
    "\\]")
```

$$f_{xy}\left(s^2, s+2t\right)$$



Parsing

Thank goodness for this too. The parser nested within the LaTeX environment `\begin{CAS}..\end{CAS}` allows for fairly natural function assignment; the name of the function must be declared in `vars(...)` (or rather, as a `SymbolExpression`) beforehand:

```
\begin{CAS}
    vars('s','t','f')
    f = f(s^2,s+2*t)
    f.derivatives = {1,1}
\end{CAS}
\[ \print{f} \]
```

$$f_{xy}\left(s^2, s+2t\right)$$

4.2 Core Methods

Any of the methods below can be used within `\begin{CAS}..\end{CAS}`. There are times when the parser or LaTeX front-end allows for simpler syntax or usability.

`function Expression:autosimplify()` `return Expression|table<number, Expression>`

Performs fast simplification techniques on an expression. The return depends on the type of input `Expression`. Generally, one should call `autosimplify()` on expressions before applying other core methods to them.

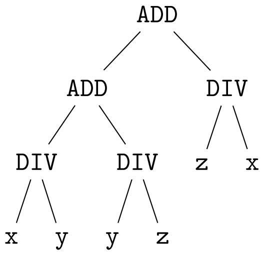
Consider the code:

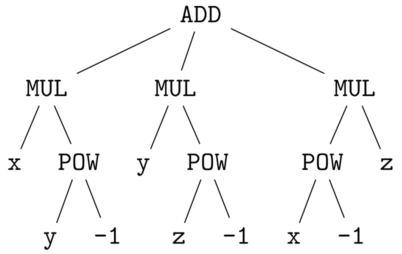
```
\begin{CAS}
    vars('x','y','z')
    w = x/y + y/z + z/x
\end{CAS}
\[ \print{w} = \print{w:autosimplify()} \]
```

The output is as follows:

$$\frac{x}{y}+\frac{y}{z}+\frac{z}{x}=\frac{x}{y}+\frac{y}{z}+\frac{z}{x}$$

It seems that `autosimplify()` did nothing; but there are significant structural differences between `w` and `w:autosimplify()`:


Expression tree for `w`

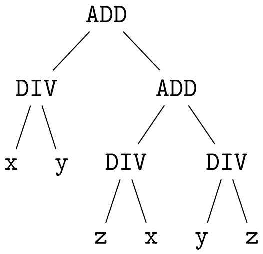

Expression tree for `w:autosimplify()`

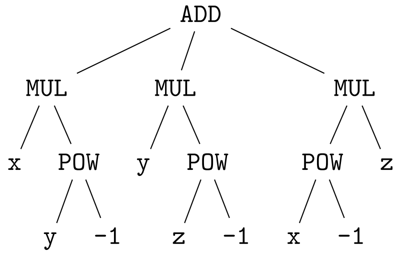
Ironically, the *autosimplified* expression tree on the right looks more complicated than the one on the left! But one of the primary functions of `autosimplify()` is to take an expression (that truly could be input in a myriad of ways) and convert that expression into something *anticipatable*.

For example, suppose the user inputs:

```
\begin{CAS}
    w = x/y + (z/x+y/z)
\end{CAS}
```

In this case, the expression trees for `w` and `w:autosimplify()`, respectively, look as follows:





Note: `w:autosimplify()` is exactly the same as it was before despite the different starting point. This is an essential function of `autosimplify()`.

Parsing

The starred variant of the LaTeX command `\print` will automatically apply the method `autosimplify()` to its argument:

```
\begin{CAS}
    vars('x')
    a = x+x/2
\end{CAS}
\[ \print{a} = \print*{a} \]
```

$$x + \frac{x}{2} = \frac{3x}{2}$$

Alternatively, you can call `autosimplify()` directly within `\begin{CAS}..\end{CAS}`:

```
\begin{CAS}
    vars('x')
    a = (x+x/2):autosimplify()
\end{CAS}
\[ \print{a} \]
```

$$\frac{3x}{2}$$

`function Expression:evaluate()` `return Expression`

Attempts to apply operations found in the expression tree of `Expression`. For instance, evaluating a `DerivativeExpression` applies the derivative operator with respect to the `symbol` field to its `expression` field. Evaluating a `BinaryOperation` with its `operation` field set to `ADD` returns the sum of the numbers in the `expressions` field, if possible. If the expression does not represent an operation or is unable to be evaluated, calling `evaluate()` on an expression returns itself.

For example, the code:

```
\directlua{
    x = Integer(1)/Integer(2)
    y = Integer(2)/Integer(3)
    z = BinaryOperation(BinaryOperation.ADD,{x,y})
}
\[ \print{z} = \print{z:evaluate()}.\]
```

produces:

$$\frac{1}{2} + \frac{2}{3} = \frac{7}{6}.$$

Parsing

Arithmetic like above is actually done automatically (via the `Ring` interface):

```
\begin{CAS}
    x = 1/2
    y = 2/3
    z = x+y
\end{CAS}
\[ z = \print{z} \]
```

$$z = \frac{7}{6}$$

Otherwise, the `evaluate()` method will attempt to evaluate all subexpressions, and then stop there:

```
\begin{CAS}
    vars('x')
    y = diff(x^2+x,x)+diff(2*x,x)
    y = y:evaluate()
\end{CAS}
\[ \print{y} \]
```

$$1 + 2x + 2$$

Whereas `autosimplify()` will return $3 + 2x$; indeed, the `autosimplify()` method (usually) begins by applying `evaluate()` first.

`function Expression:expand()` `return Expression`

Expands an expression, turning products of sums into sums of products.

```
\begin{CAS}
    vars('x','y','z','w')
    a = x+y
    b = z+w
    c = a*b
\end{CAS}
\[ \print{c} = \print{c:expand()} \]
```

$$(x + y)\,(z + w) = wx + wy + xz + yz$$

Parsing

There is an `expand()` function in the parser; though it calls the `autosimplify()` method first. So, for example, `expand(c)` is equivalent to `c:autosimplify():expand()`.

`function Expression:factor()` `return Expression`

Factors an expression, turning sums of products into products of sums. For general `Expressions` this functionality is somewhat limited. For example:

```
\begin{CAS}
    vars('x')
    a = x-1
    b = a*x+a
\end{CAS}
\[ \print{b} = \print{b:factor()} \]
```

$$(x - y)\,x + (x - y)\,y = (x + y)\,(x - y)$$

On the other hand:

```
\begin{CAS}
    vars('x','y')
    a = x^2-y^2
\end{CAS}
\[ \print{a} = \print{a:factor()} \]
```

$$x^2 - y^2 = x^2 - y^2$$

Parsing

There is a `factor()` function in the parser that is more class-aware than the basic `:factor()` method mentioned here. For example:

```
\begin{CAS}
    x = 12512
\end{CAS}
\[ \print{x:factor()} = \print{factor(x)} \]
```

$$12512 = 17^1 2^5 23^1$$

`function Expression:freeof(symbol)` `return bool`

Determines whether or not `Expression` contains a particular `symbol` somewhere in its expression tree.

The method `freeof()` is quite literal. For example:

```
vars('foo','bar')
baz = foo+bar
if baz:freeof(foo) then
  tex.sprint(baz:tolatex(), " is free of ",
            foo:tolatex(),"!")
else
  tex.sprint(baz:tolatex(), " is bound by ",
            foo:tolatex(),".")
end
```

foo+bar is bound by foo.

On the other hand, the expression tree for `SymbolExpression("foo")` contains a single node with no edges. With nary a `SymbolExpression("fo")` to find in such an expression tree, we have:

```
vars('foo','fo')
if foo:freeof(fo) then
  tex.sprint(foo:tolatex()," is free of ",
            fo:tolatex(),"!")
else
  tex.sprint(foo:tolatex()," is bound by ",
            fo:tolatex(),'.')
end
```

foo is free of fo!

`function Expression:isatomic()` `return bool`

Determines whether an expression is *atomic*. Typically, atomicity is measured by whether the `Expression` has any subexpression fields. So, for example, `Integer(5)` and `Integer(15)` are atomic, and so is `Integer(20)`. But:

```
BinaryOperation(BinaryOperation.ADD,
    {Integer(5),Integer(15)})
```

is non-atomic.

```
x = SymbolExpression("x")
y = x*x+x
if x:isatomic() then
  tex.print(tostring(x),"is atomic;")
end
if not y:isatomic() then
  tex.print(tostring(y),"is compound.")
end
```

x is atomic; (x * x) + x is compound.

Since `SymbolExpression` inherits from `AtomicExpression`, we have that `isatomic()` is taken literally as well. For example:

```
y = SymbolExpression("x*x+x")
if not y:isatomic() then
    tex.print(tostring(y),"is compound.")
else
    tex.print("But",tostring(y),"is atomic,
        from a certain point of view.")
end
```

But x*x+x is atomic, from a certain point of view.

`function Expression:iscomplexconstant()` `return bool`

Determines whether an expression is a complex number in the mathematical sense, such as $3 + \sqrt{2}i$. It's helpful to keep in mind that, oftentimes, content needs to be simplified/evaluated in order to obtain the intended results:

```
a = (Integer.one() + I) ^ Integer(2)
if a:iscomplexconstant() then
    tex.print("$",a:tolatex(),"$ is a complex constant.")
else
    tex.print("$",a:tolatex(),"$ is not a complex constant.")
end
```

$(1+i)^2$ is not a complex constant.

While:

```
a = (Integer.one()+I) ^ Integer(2)
a = a:expand():simplify()
if a:iscomplexconstant() then
    tex.print("$",a:tolatex(),"$ is a complex constant.")
else
    tex.print("$",a:tolatex(),"$ is not a complex constant.")
end
```

$2i$ is a complex constant.

`function Expression:isconstant()` `return bool`

Determines whether an expression is atomic and contains no variables. This method is counterintuitive in some cases. For instance:

```
if not pi:isconstant() then
  tex.print("$\\pi$ is not constant.")
end
```

π is not constant.

This is because `isconstant()` is meant to check for certain autosimplification transformations that can be performed on arbitrary `Ring` elements but not on those constants. Use `isrealconstant()` for what mathematicians think of as constants.

`function Expression:isrealconstant()` `return bool`

Determines whether an expression is a real number in the mathematical sense, such as 2, $\sqrt{5}$, or $\sin(3)$. For example:

```
if pi:isrealconstant() then
  tex.print("$\\pi$ is a real constant.")
end
```

π is a real constant.

`function Expression:order(Expression)` `return boolean`

For the goals of autosimplification, `Expressions` must be ordered. `Expression:order(other)` method returns `true` if `Expression` is "less than" `other` according to this ordering.

On certain classes, the ordering is intuitive:

```
a = 4
b = 3
if a:order(2) then
    tex.print(a:tolatex(),
    "is less than",
    b:tolatex())
else
    tex.print(b:tolatex(),
        "is less than",
         a:tolatex())
end
```

3 is less than 4

On `SymbolExpressions`, the ordering is lexigraphic:

```
vars('a')
vars('b')
if b:order(a) then
    tex.print(b:tolatex(),
    "is less than",
    a:tolatex())
else
    tex.print(a:tolatex(),
        "is less than",
         b:tolatex())
end
```

a is less than b

Of course, inter-class comparisons can be made as well – but these are predominantly dictated by typesetting conventions.

`function Expression:setsubexpressions(subexpressions)` `return Expression`

Creates a copy of an expression with the list of subexpressions as its new subexpressions. This can reduce code duplication in other methods.

`function Expression:simplify()` `return Expression`

Performs more extensive simplification of an expression. This may be slow, so this function is separate from autosimplification and is not called unless the user specifically directs the CAS to do so. The method aims to find an expression tree equivalent to the one given that is "smaller" in size as measured by the number of nodes in the expression tree.

The `simplify()` method does call the `autosimplify()` method first. Here's an example of where the results of `autosimplify()` and `simplify()` differ:

```
\begin{CAS}
    vars('x')
    a = 1-x+0*x
    b = 1+1*x
    c = a*b
\end{CAS}
\[ \print{c} = \print{c:autosimplify()} = \print{c:simplify()}. \]
```

The code above produces:

$$(1-x+0\cdot x)\,(1+1x) = (1+x)\,(1-x) = 1-x^2.$$

Parsing

There is a `simplify()` function for those unfamiliar with Lua methods. So, for example, `c:simplify()` is equivalent to `simplify(c)`.

`function Expression:size()` **`return Integer`**

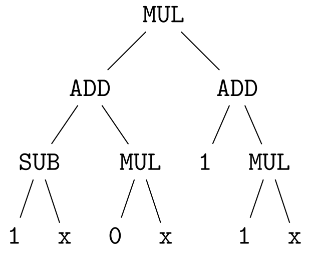
Returns the number of nodes of the tree that constitutes an expression, or roughly the total number of expression objects that make up the expression.

For example, consider:

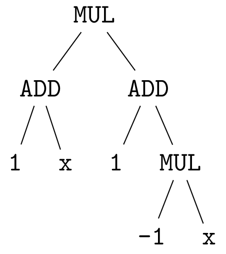
```
\begin{CAS}
    vars('x')
    a = (1-x+0*x)
    b = (1+1*x)
    c = a*b
\end{CAS}
```

Then the expression trees for `c`, `c:autosimplify()`, and `c:simplify()` are as follows:

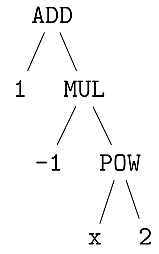
Tree for `c`:



Tree for `c:autosimplify()`:



Tree for `c:simplify()`:



Accordingly, we have:

```
tex.print("The size of \\texttt{c} is",
  tostring(c:size()),"\\newline")
tex.print("The size of
  \\texttt{c:autosimplify()} is",
  tostring(c:autosimplify():size()),
  ↪  "\\newline")
tex.print("The size of
  \\texttt{c:simplify()} is",
  tostring(c:simplify():size()))
```

The size of `c` is 13
The size of `c:autosimplify()` is 9
The size of `c:simplfy()` is 7

`function Expression:subexpressions()` **`return table<number, Expression>`**

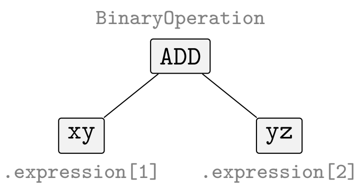
Returns a list of all subexpressions of an expression. This gives a unified interface to the instance variables for subexpressions, which have different names across classes. For example, consider:

```
\begin{CAS}
    vars('x','y','z')
    a = x*y+y*z
    b = int(sin(x),x,0,pi/2)
\end{CAS}
\[ a = \print{a} \quad \text{and} \quad
↪  b=\print{b}.\]
```

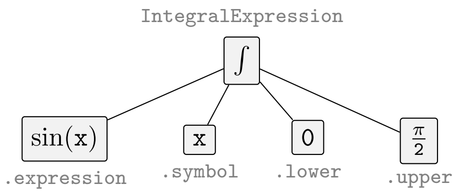
$$a = xy + yz \quad \text{and} \quad b = \int_0^{\frac{\pi}{2}} \sin(x)\,dx.$$

Here are the expression shrubs for `a` and `b`:

Expression shrub for `a`



Expression shrub for `b`



On the other hand:

```
for _,expr in ipairs(a:subexpressions()) do
    tex.print("$", expr:tolatex(),
    ↪  "$\\quad")
end
```

$xy \quad yz$

while:

```
for _,expr in ipairs(b:subexpressions()) do
    tex.print("$", expr:tolatex(),
    ↪  "$\\quad")
end
```

$\sin(x) \quad x \quad 0 \quad \frac{\pi}{2}$

`function Expression:substitute(map)` `return Expression`

The input `map` is a table that maps expressions to expressions; the method then recursively maps each instance of an expression with its corresponding table expression. One should take care when replacing multiple compound expressions in a single command, since there is no guarantee as to the order in which subexpressions in the table are replaced.

```
\begin{CAS}
  vars('foo','bar','baz')
  qux = (foo/bar)
  qux = qux:substitute({[foo]=bar,[bar]=baz})
\end{CAS}
\[ \print{qux} \]
```

$$\frac{bar}{baz}$$

Parsing

There is a `substitute()` function with a slightly more user-friendly syntax. In particular,

`(foo/bar):substitute({[foo]=bar,[bar]=baz})`

is equivalent to

`substitute({[foo]=bar,[bar]=baz}, foo/bar)`

`function Expression:tolatex()` `return string`

Converts an expression to LaTeX code. Some folks have strong feelings about how certain things are typeset. Case and point, which of these is your favorite:

$$\int \sin(\frac{y}{2}) dy \qquad \int \sin\left(\frac{y}{2}\right) dy \qquad \int \sin\left(\frac{y}{2}\right)\, dy \qquad \int\sin\left(\frac{y}{2}\right)\, dy \qquad \int \sin\left(\frac{y}{2}\right) \mathrm{d}y \qquad \int \sin\left(\frac{y}{2}\right) dy \quad ?$$

We've tried to remain neutral:

```
\begin{CAS}
    vars('y')
    f = diff(int(sin(y/2),y),y)
\end{CAS}
\[ \print{f} \]
```

$$\frac{d}{dy}\left(\int \sin\left(\frac{y}{2}\right) dy\right)$$

With any luck, we've pleased at least as many people as we've offended. In desperate times, one could rewrite the `tolatex()` method for any given class. Here, for example, is the `tolatex()` method as written for the `DerivativeExpression` class:

```
function DerivativeExpression:tolatex()
    return '\\frac{d}{d' .. self.symbol:tolatex() .. '}\\left(' ..
    ↪  self.expression:tolatex() .. '\\right)'
end
```

But there are heathens that live among us who might prefer:

```
function DerivativeExpression:tolatex()
    return '\\frac{\\mathrm{d}}{\\mathrm{d}' .. self.symbol:tolatex() .. '}\\left(' ..
    ↪  self.expression:tolatex() .. '\\right)'
end
```

If we include the above function in a separate file, say `mytex.lua`, and use:

```
\directlua{dofile('mytex.lua')}
```

or include the above function directly into the document via `\directlua` or `\luaexec`, then we would get:

```
\begin{CAS}
  f = DerivativeExpression(y+sin(y),y)
\end{CAS}
\[ \print{f} \]
```

$$\frac{\mathrm{d}}{\mathrm{d}y}\left(y+\sin(y)\right).$$

Parsing

The LaTeX command `\print` calls the method `tolatex()` unto its argument and uses `tex.print()` to display the results. The starred variant `\print*` applies the `autosimplify()` method before applying `tolatex()`.

Additionally, one can use the `disp()` function within `\begin{CAS}..\end{CAS}`.

```
\begin{CAS}
  f = DerivativeExpression(y+sin(y),y)
  disp(f)
\end{CAS}
```

$$\frac{d}{dy}\left(y+\sin(y)\right)$$

The function `disp` takes two optional boolean arguments both are set to `false` by default. The first optional boolean controls *inline* vs *display* mode; the second optional boolean controls whether the method `autosimplify()` is called before printing:

```
\begin{CAS}
    disp(f,true)
\end{CAS}
```

$\frac{d}{dy}\left(y+\sin(y)\right)$

```
\begin{CAS}
 disp(f,true,true)
\end{CAS}
```

$1+\cos(y)$

```
\begin{CAS}
 disp(f,false,true)
\end{CAS}
```

$$1+\cos(y)$$

`function Expression:topolynomial()` `return Expression | bool`

Attempts to cast `Expression` into a polynomial type (`PolynomialRing`); there are multiple outputs. The first output is `self` or `PolynomialRing`; the second output is `false` or `true`, respectively. `PolynomialRing` is the name of the class that codifies univariate polynomials proper.

Polynomial computations tend to be significantly faster when those polynomials are stored as arrays of coefficients (as opposed to, say, when they are stored as generic `BinaryOperations`). Hence the need for a method like `topolynomial()`.

Warning: the `topolynomial()` method expects the input to be autosimplified. For example:

```
\begin{CAS}
  vars('x')
  f = 3+2*x+x^2
  f,b = f:topolynomial()
  if b then
    tex.print("\\[",f:tolatex(),"\\]")
  else
    tex.print("womp womp")
  end
\end{CAS}
```

womp womp

```
\begin{CAS}
  vars('x')
  f = 3+2*x+x^2
  f,b = f:autosimplify():topolynomial()
  if b then
    tex.print("\\[",f:tolatex(),"\\]")
  else
    tex.print("womp womp")
  end
\end{CAS}
```

$$x^2 + 2x + 3$$

Parsing

There is a `topoly()` function that applies `:autosimplify()` automatically to the input. For example:

```
\begin{CAS}
    vars('x')
    f = 3+2*x+x^2
    f = topoly(f)
\end{CAS}
The Lua variable \texttt{f} is the \whatis{f}: $\print{f}$.
```

The Lua variable `f` is the `PolynomialRing`: $x^2 + 2x + 3$.

`function Expression:type()` `return Expression | bool`

Returns the `__index` field in the metatable for `Expression`. In other words, this function returns the type of `Expression`. Here's typical usage:

```
\begin{CAS}
  vars('x')
  if x:type() == SymbolExpression then
    tex.print(x:tolatex(), "is a
    ↪  SymbolExpression.")
  end
\end{CAS}
```

x is a SymbolExpression.

Parsing

The LaTeX command `\whatis` can be used to print the type of `Expression`:

```
x is a \whatis{x}
```

x is a `SymbolExpression`

Alternatively, there's a `whatis()` function and a `longwhatis()` function that can be called within a Lua environment (like `\directlua` or `\luaexec`):

```
tex.print(whatis(x), '\\newline')
tex.print(longwhatis(x))
```

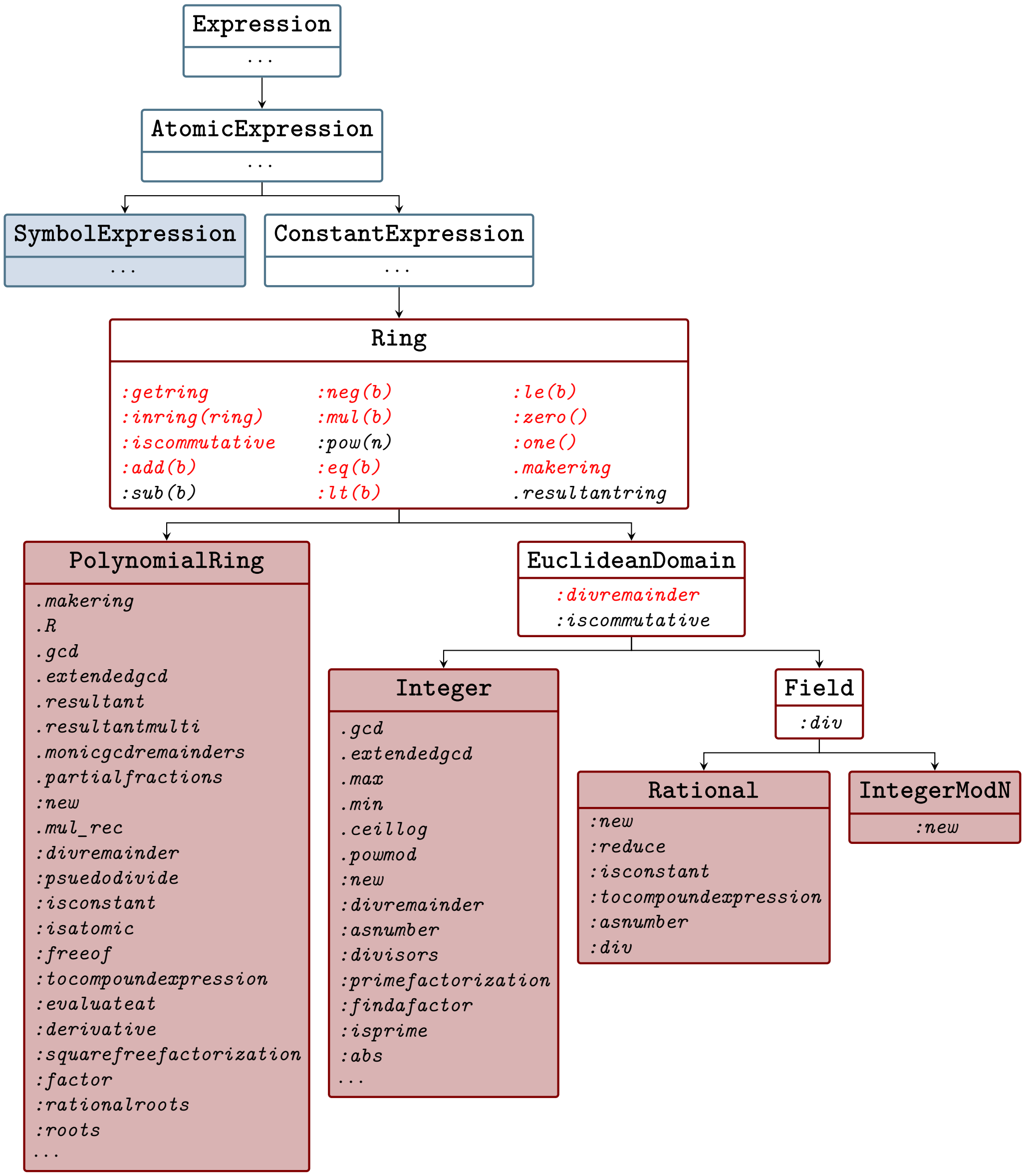
Sym
SymbolExpression

5 Algebra

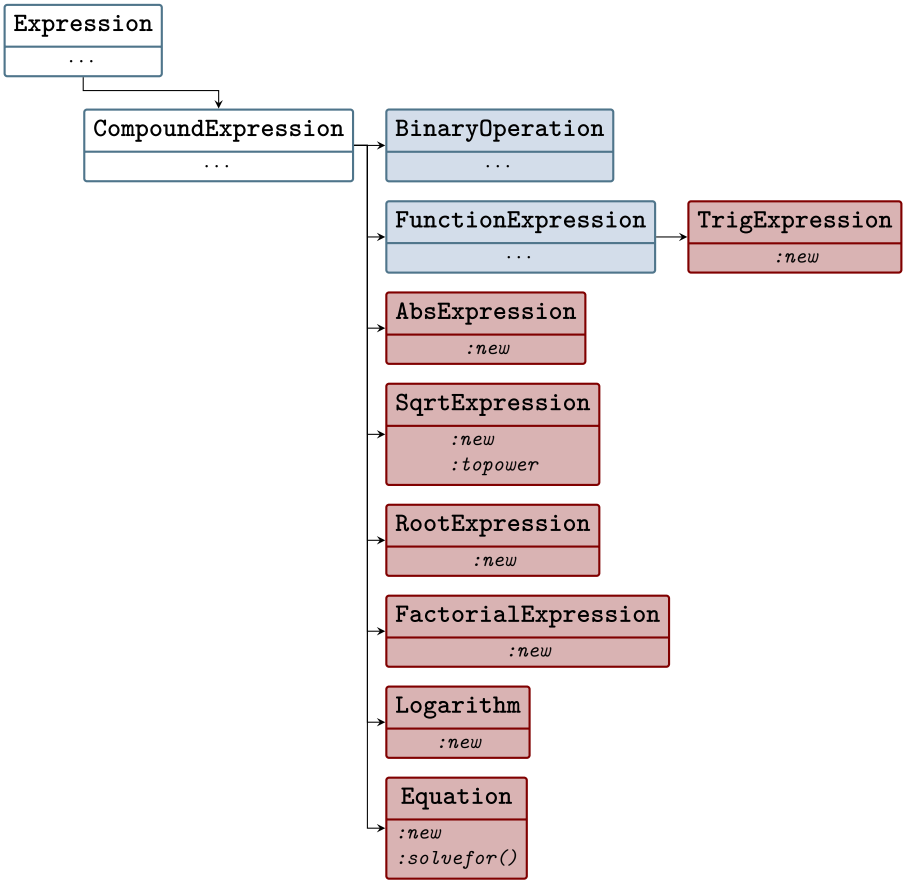
This section contains reference materials for the algebra functionality of `luacas`. The classes in this module are diagramed below according to inheritance along with the methods/functions one can call upon them.

- *method*: an abstract method;
- *method*: a method that returns the expression unchanged;
- *method*: method that is either unique, implements an abstract method, or overrides an abstract method;
- `Class` : a concrete class.

Here is an inhertiance diagram of the classes in the algebra module that are derived from the `AtomicExpression` branch of classes. However, not all of them are proper `ConstantExpressions`, so some of them override the `isconstant()` method. Most methods are stated, but some were omitted (because they inherit in the obvious way, they are auxiliary and not likely to be interesting to the end-user, etc).



Here is an inhertiance diagram of the classes in the algebra module that are derived from the `CompoundExpression` branch of classes. Again, most methods are stated, but some were omitted (because they inherit in the obvious way, they are auxiliary and not likely to be interesting to the end-user, etc).



5.1 Algebra Classes

The algebra package contains functionality for arbitrary-precision arithmetic, polynomial arithmetic and factoring, symbolic root finding, and logarithm and trigonometric expression classes. It requires the core package to be loaded.

The abstract classes in the algebra module all inherit from the `ConstantExpression` branch in the inheritance tree:

- `Ring`
- `EuclideanDomain`
- `Field`

The `EuclideanDomain` class is a sub-class to the `Ring` class, and the `Field` class is a sub-class to the `EuclideanDomain` class.

The following concrete classes inherit from the `Ring` class (or one of the sub-classes mentioned above). However, not all of them are proper `ConstantExpressions`, so some of them override the `isconstant()` method.

- `Integer`
- `IntegerModN`
- `Rational`
- `PolynomialRing`

The other concrete classes in the Algebra package do not inherit from the `Ring` interface, instead they inherit from the `CompoundExpression` interface:

- `AbsExpression`
- `Logarithm`
- `FactorialExpression`
- `SqrtExpression`
- `TrigExpression`
- `RootExpression`
- `Equation`

`n number|string|Integer`

```
function Integer:new(n)                                    return Integer
```

Takes a `string`, `number`, or `Integer` input and constructs an `Integer` expression. The `Integer` class allows us to perform exact arithmetic on integers. Indeed, since Lua can only store integers exactly up to a certain point, it is recommended to use strings to build large integers.

```
a = Integer(-12435)
b = Integer('-12435')
tex.print('\\[',a:tolatex(),
    '=',
    b:tolatex(),
    '\\]')
```

$$-12435 = -12435$$

An `Integer` is a table 1-indexed by Lua numbers consisting of Lua numbers. For example:

```
tex.print(tostring(b[1]))
```

12435

Whereas:

```
c = Integer('72405313609493819475281315O8')
tex.print('The first 14 digits of c:', tostring(c[1]),'. ')
tex.print('The last 14 digits of c:', tostring([2]),'.')
```

The first 14 digits of c: 81947528131508 . The last 14 digits of c: 72405313609493 .

The global field `DIGITSIZE` is set to `14` so that exact arithmetic on `Integer`s can be done as efficiently as possible while respecting Lua's limitations.

Fields

`Integer`s have a `.sign` field which contains the Lua number `1` or `-1` depending on whether `Integer` is positive or negative.

```
tex.print('The sign of',tostring(b),'is:',tostring(b.sign))
```

The sign of -12435 is: -1

Parsing

The contents of the environment `\begin{CAS}..\end{CAS}` are wrapped in the argument of a function `CASparse()` which, among other things, seeks out digit strings intended to represent integers, and wraps those in `Integer('...')`.

```
\begin{CAS}
    c = 7240531360949381947528131508
\end{CAS}
\directlua{
    tex.print(tostring(c[1]))
}
```

81947528131508

```
                                                i Integer, n Integer
function IntegerModN:new(i,n)                    return IntegerModN
```

Takes an `Integer i` and `Integer n` and constructs an element in the ring $\mathbf{Z}/n\mathbf{Z}$, the integers modulo n.

```
i = Integer(143)
n = Integer(57)
a = IntegerModN(i,n)
tex.print('\\[',i:tolatex(),'\\equiv',a:tolatex(true),'\\]')
```

$$143 \equiv 29 \mod 57$$

Fields

`IntegerModN`s have two fields: `.element` and `.modulus`. The reduced input `i` is stored in `.element` while the input `n` is stored in `.modulus`:

```
tex.print(a.element:tolatex(),'\\newline')
tex.print(a.modulus:tolatex())
```

29
57

Parsing

The function `Mod(,)` is a shortcut for `IntegerModN(,)`:

```
\begin{CAS}
    i = 143
    n = 57
    a = Mod(i,n)
\end{CAS}
\[\print{i}\equiv\print{a}\bmod{\print{n}}\]
```

$$143 \equiv 29 \bmod 57$$

```
                            coefficients table<number,Ring>, symbol string|SymbolExpression, degree Integer
function PolynomialRing:new(coefficients, symbol, degree)                           return PolynomialRing
```

Takes a table of `coefficients`, not all necessarily in the same ring, and a `symbol` to create a polynomial in `R`[`x`] where `x` is `symbol` and `R` is the smallest `Ring` possible given the coefficients. If `degree` is omitted, it will calculate the degree of the polynomial automatically. The list can either be one-indexed or zero-indexed, but if it is one-indexed, the internal list of coefficients will still be zero-indexed.

```
\begin{CAS}
  f = PolynomialRing({0,1/3,-1/2,1/6},'t')
\end{CAS}
\[ \print{f} \]
```

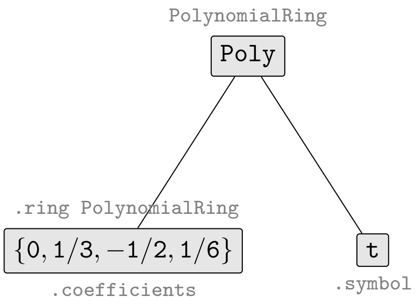
$$\frac{1}{6}t^3 - \frac{1}{2}t^2 + \frac{1}{3}t$$

The `PolynomialRing` class overwrites the `isatomic()` and `isconstant()` inheritances from the abstract class `ConstantExpression`.

Fields

`PolynomialRing`s have several fields:

- `f.coefficients` stores the 0-indexed table of coefficients of `f`;
- `f.degree` stores the `Integer` that represents the degree of `f`;
- `f.symbol` stores the `string` representing the variable or `symbol` of `f`.
- `f.ring` stores the `RingIdentifier` for the ring of coefficients.



For example:

```
for i=0,f.degree:asnumber() do
  tex.print('\\[',
    f.coefficients[i]:tolatex(),
    f.symbol,
    '^{',
    tostring(i),
    '}\\]')
end
if f.ring == Rational.getring() then
  tex.print('Rational coefficients')
end
```

$$0t^0$$

$$\frac{1}{3}t^1$$

$$-\frac{1}{2}t^2$$

$$\frac{1}{6}t^3$$

Rational coefficients

Parsing

The function `Poly()` is a shortcut for `PolynomialRing:new()`. If the second argument `symbol` is omitted, then the default is `'x'`:

```
\begin{CAS}
    f = Poly({0,1/3,-1/2,1/6})
\end{CAS}
\[ \print{f} \]
```

$$\frac{1}{6}x^3 - \frac{1}{2}x^2 + \frac{1}{3}x$$

Alternatively, one could typeset the polynomial naturally and use the `topoly()` function. This is the same as the `topolynomial()` method except that the `autosimplify()` method is automatically called first:

```
\begin{CAS}
    vars('x')
    f = 1/3*x - 1/2*x^2 + 1/6*x^3
    f = topoly(f)
\end{CAS}
\[ \print{f} \]
```

$$\frac{1}{6}x^3 - \frac{1}{2}x^2 + \frac{1}{3}x$$

`n Ring, d Ring, keep bool`

`function Rational:new(n,d,keep)` `return Rational`

Takes a numerator `n` and denominator `d` in the same `Ring` and constructs a rational expression in the field of fractions over that ring. For the integers, this is the ring of rational numbers. If the `keep` flag is omitted, the constructed object will be simplified to have smallest possible denominator, possibly returning an object in the original `Ring`. Typically, the `Ring` will be either `Integer` or `PolynomialRing`, so `Rational` can be viewed as a constructor for either a rational number or a rational function.

For example:

```
a = Integer(6)
b = Integer(10)
c = Rational(a,b)
tex.print('\\[',c:tolatex(),'\\]')
```

$$\frac{3}{5}$$

But also:

```
a = Poly({Integer(2),Integer(3)})
b = Poly({Integer(4),Integer(1)})
c = Rational(a,b)
tex.print('\\[',c:tolatex(),'\\]')
```

$$\frac{3x+2}{x+4}$$

Fields

`Rational`s naturally have the two fields: `numerator`, `denominator`. These fields store precisely what you think. `Rational`s also have a `ring` field which stores the `RingIdentifier` to which the numerator and denominator belong. (This is $\mathbb{Z}$ for the rational numbers.)

If `numerator` or `denominator` are `PolynomialRing`s, then the constructed `Rational` will have an additional field: `symbol`. This stores the symbol the polynomial rings are constructed over.

```
if c.ring == PolynomialRing.getring() then
  tex.print('$',c:tolatex(),'$ is a Rational Function in the variable',c.symbol)
end
```

$\frac{3x+2}{x+4}$ is a Rational Function in the variable x

Parsing

`Raionals` are constructed naturally using the / operator:

```
\begin{CAS}
    a = Poly({2,3})
    b = Poly({4,1})
    c = a/b
\end{CAS}
\[ \print{c} \]
```

$$\frac{3x+2}{x+4}$$

```
function AbsExpression:new(expression)                          return AbsExpression
```

Creates a new absolute value expression with the given expression.

```
\begin{CAS}
    f = Poly({1,1})
    g = Poly({-1,1})
    h = AbsExpression(f/g)
\end{CAS}
\[ h = \print{h} \]
```

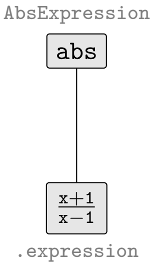
$$h = \left|\frac{x+1}{x-1}\right|$$

Fields

`AbsExpressions` have only one field: `.expression`. This field simply holds the `Expression` inside the absolute value:

```
tex.print('\\[',
    h.expression:tolatex(),
    '\\]')
```

$$\frac{x+1}{x-1}$$



Parsing

The function `abs()` is a shortcut to `AbsExpression:new()`. For example:

```
\begin{CAS}
    f = Poly({1,1})
    g = Poly({-1,1})
    h = abs(f/g)
\end{CAS}
\[ h = \print{h} \]
```

$$h = \left|\frac{x+1}{x-1}\right|$$

`base Expression, arg Expression`

`function Logarithm:new(base,arg)` `return Logarithm`

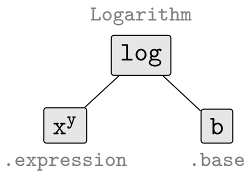
Creates a new `Logarithm` expression with the given `base` and `argument`. Some basic simplification rules are known to `autosimplify()`:

```
\begin{CAS}
    vars('b','x','y')
    f = Logarithm(b,x^y)
\end{CAS}
\[ \print{f} = \print*{f} \]
```

$$\log_b(x^y) = y\log_b(x)$$

Fields

`Logarithm`s have two fields: `base` and `expression`; `base` naturally stores the base of the logarithm (i.e., the first argument of `Logarithm`) while `expression` stores the argument of the logarithm (i.e., the second argument of `Logarithm`).



Parsing

The function `log()` is a shortcut to `Logarithm`:

```
\begin{CAS}
    vars('b')
    f = log(b,b)
\end{CAS}
\[ \print{f} = \print*{f} \]
```

$$\log_b(b) = 1$$

There is also a `ln()` function to shortcut `Logarithm` where the base is `e`, the natural exponent.

```
\begin{CAS}
    f = ln(e)
\end{CAS}
\[ \print{f} = \print*{f} \]
```

$$\ln(e) = 1$$

`expression Expression`

`function FactorialExpression:new(expression)` `return FactorialExpression`

Creates a new `FactorialExpression` with the given `expression`. For example:

```
\begin{CAS}
    a = FactorialExpression(5)
\end{CAS}
\[ \print{a} \]
```

$$5!$$

The `evaluate()` method will compute factorials of nonnegative `Integer`s:

```
\begin{CAS}
    a = FactorialExpression(5)
\end{CAS}
\[ \print{a} = \print{a:evaluate()} \]
```

$$5! = 120$$

Fields

`FactorialExpression`s have only one field: `expression`. This field stores the argument of `FactorialExpression()`.

Parsing

The function `factorial()` is a shortcut to `FactorialExpression()`:

```
\begin{CAS}
    a = factorial(5)
\end{CAS}
\[ \print{a} = \print{a:evaluate()} \]
```

$$5! = 120$$

```
                                        expression Expression, root Integer
function SqrtExpression:new(expression, root)          return SqrtExpression
```

Creates a new `SqrtExpression` with the given `expression` and `root`. Typically, `expression` is an `Integer` or `Rational`, and `SqrtExpression` is intended to represent a positive real number. If `root` is omitted, then `root` defaults to `Integer(2)`. For example:

```
a = SqrtExpression(Integer(8))
b = SqrtExpression(Integer(8),Integer(3))
c = a+b
tex.print('\\[',c:tolatex(),'\\]')
```

$$\sqrt{8} + \sqrt[3]{8}$$

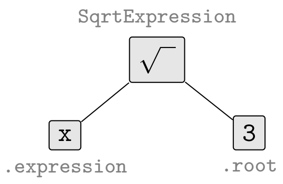
When `expression` and `root` are of the `Integer` or `Rational` types, then `autosimplify()` does a couple things. For example, with `a,b` as above, we get:

```
c = c:autosimplify()
tex.print('\\[',c:tolatex(),'\\]')
```

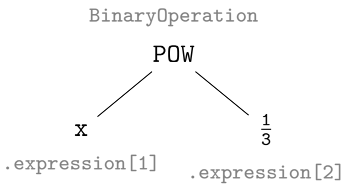
$$2 + 2\sqrt{2}$$

On the other hand, if `root` or `expression` are not constants, then typically `autosimplify()` will convert `SqrtExpression` to the appropriate `BinaryOperation`. For example:

Tree for `a`



Tree for `a:autosimplify()`



Parsing

The function `sqrt()` shortcuts `SqrtExpression()`:

```
\begin{CAS}
    a = sqrt(1/9)
    b = sqrt(27/16,3)
    c = a+b
\end{CAS}
\[ \print{c} = \print*{c} \]
```

$$\sqrt{\frac{1}{9}} + \sqrt[3]{\frac{27}{16}} = \frac{1}{3} + \frac{3\sqrt[3]{4}}{4}$$

```
                                          name string|SymbolExpression, expression Expression
function TrigExpression:new(name,expression)                            return TrigExpression
```

Creates a new trig expression with the given `name` and `expression`. For example:

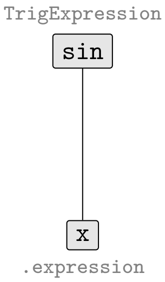
```
vars('x')
f = TrigExpression('sin',x)
tex.print('\\[',f:tolatex(),'\\]')
```

$$\sin(x)$$

Fields

`TrigExpression`s have many fields:

- `TrigExpression.name` stores the string `name`, i.e. the first argument of `TrigExpression()`;
- `TrigExpression.expression` stores the `Expression expression`, i.e. the second argument of `TrigExpression()`;
- and all fields inherited from `FunctionExpression` (e.g. `TrigExpression.derivatives` which defaults to `Integer.zero()`).



Parsing

The usual trigonometric functions have the anticipated shortcut names. For example:

```
\begin{CAS}
    f = arctan(x^2)
\end{CAS}
\[ \print{f} \]
```

$$\arctan(x^2)$$

```
                                                                    expression Expression
function RootExpression:new(expression)                                 return RootExpression
```

Creates a new `RootExpression` with the given `expression`. The method `RootExpression:autosimplify()` attempts to return a list of zeros of `expression`. If no such set can be found, then

`RootExpression(expression:autosimplify())`

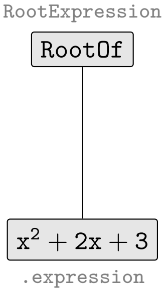
is returned instead. At the moment, `expression` must be a univariate polynomial of degree $0, 1, 2$ or 3 in order for the `autosimplify()` method to return anything interesting. Of course, `luacas` can find roots of higher degree polynomials, but this involves more machinery/methods within the `PolynomialRing` class.

Fields

`RootExpression`s have only one field: `.expression`. For example:

```
\begin{CAS}
    f = Poly({3,2,1})
    r = RootExpression(f)
\end{CAS}
\[ \print{r} \]
```

$$\mathrm{RootOf}\left(x^2+2x+3\right)$$



Parsing

The function `roots()` essentially shortcuts `RootExpression()`, but when `expression` is of the `PolynomialRing`-type, then `PolynomialRing:roots()` is returned.

```
\begin{CAS}
    r = roots(f)
\end{CAS}
\[ \print{r[1]} \qquad \print{r[2]} \]
```

$$-1+\sqrt{2}i \qquad -1-\sqrt{2}i$$

```
                                            lhs Expression, rhs Expression
function Equation:new(lhs, rhs)                            return Equation
```

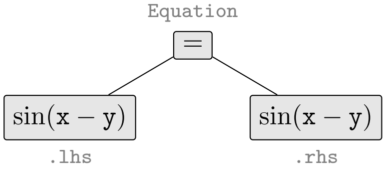
Creates a new `Equation` expression with the given `lhs` (left hand side) and `rhs` (right hand side). If both sides of the equation are constants, or structurally identical, `autosimplify()` will return a boolean:

```
\begin{CAS}
    vars('x','y')
    f = Equation(sin(x-y),sin(x-y))
    g = f:autosimplify()
\end{CAS}
\[ \print{f} \to \print{g} \]
```

$$\sin(x-y)=\sin(x-y)\to \mathit{true}$$

Fields

`Equations` have two fields: `lhs` and `rhs`; which store the expressions on the left and right sides of the equation.



5.2 Algebra Methods

Many classes in the algebra package inherit from the `Ring` interface. The `Ring` interface requires the following arithmetic operations, which have corresponding abstract metamethods listed below. Of course, these abstract methods get passed to the appropriate concrete methods in the concrete classes that inherit from `Ring`.

For `Ring` objects `a` and `b`:

`function a:add(b) return a + b`	Adds two ring elements.
`function a:sub(b) return a - b`	Subtracts one ring element from another. Subtraction has a default implementation in `Ring.lua` as adding the additive inverse, but this can be overwritten if a faster performance method is available.
`function a:neg() return -a`	Returns the additive inverse of a ring element.
`function a:mul(b) return a * b`	Multiplies two ring elements.
`function a:pow(n) return a ^ n`	Raises one ring element to the power of an integer. Exponentiation has a default implementation as repeated multiplication, but this can (and probably should) be overwritten for faster performance.
`function a:eq(b) return a == b`	Tests if two ring elements are the same.
`function a:lt(b) return a < b`	Tests if one ring element is less than another under some total order. If the ring does not have a natural total order, this method does not need to be implemented.
`function a:le(b) return a <= b`	Tests if one ring element is less than or equal to another under some total order. If the ring does not have a natural total order, this method does not need to be implemented.
`function a:zero() return Ring`	Returns the additive identity of the ring to which `a` belongs.
`function a:one() return Ring`	Returns the multiplicative identity of the ring to which `a` belongs.

☛ Arithmetic of `Ring` elements will (generally) not form a `BinaryOperation`. Instead, the appropriate `__RingOperation` is called which then passes the arithmetic to a specific ring, if possible. For example:

```
\begin{CAS}
    f = Poly({2,1})
    g = Poly({2,5})
    h = f+g
\end{CAS}
\[ (\print{f}) + (\print{g}) = \print{h} \]
```

$$(x+2)+(5x+2)=6x+4$$

So why have the `Ring` class to begin with? Many of the rings in the algebra package are subsets of one another. For instance, integers are subsets of rationals, which are subsets of polynomial rings over the rationals, etc. To smoothly convert objects from one ring to another, it's good to have a class like `Ring` to handle all the "traffic."

For example, the `RingIdentifier` object acts as a pseudo-class that stores information about the exact ring of an object, including the symbol the ring has if it's a polynomial ring. To perform operations on two elements of different rings, the CAS does the following:

To get the generic `RingIdentifier` from a class, it uses the static method:

```
function Ring.makering()                                        return RingIdentifier
```

To get the `RingIdentifier` from a specific instance (element) of a ring, it uses the method:

```
function Ring:getring()                                    return RingIdentifier
```

So, for example:

```
a = Integer(2)/Integer(3)
ring = a:getring()
if ring == Integer.makering() then
    tex.print('same rings')
else
    tex.print('different rings')
end
```

different rings

From there, the CAS computes the smallest `RingIdentifier` that contains the two `RingIdentifiers` as subsets using the static method:

```
                                   ring1 RingIdentifier, ring2 RingIdentifier
function Ring.resultantring(ring1,ring2)                   return RingIdentifier
```

So, for example:

```
a = Poly({Integer(2),Integer(1)})
b = Integer(3)
ring1 = a:getring()
ring2 = b:getring()
ring = Ring.resultantring(ring1,ring2)
if ring == a:getring() then
    tex.print('polynomial ring')
end
```

polynomial ring

Finally, the CAS converts both objects into the resultant `RingIdentifier`, if possible, using the method:

```
function Ring:inring(ring)                                 return Ring
```

So, for example:

```
b = b:inring(ring)
if b:type() == PolynomialRing then
    tex.print('b is a polynomial now')
end
```

b is a polynomial now

Finally, the CAS is able to perform the operation with the correct `__RingOperation`. This all happens within the hierarchy of `Ring` classes automatically:

```
\begin{CAS}
    a = Poly({1/2,3,1})
    b = 1/2
    c = a+b
\end{CAS}
\[ \print{a} + \print{b} = \print{c} \]
```

$$x^2 + 3x + \frac{1}{2} + \frac{2}{3} = x^2 + 3x + \frac{7}{6}$$

To add another class that implements `Ring` and has proper conversion abilities, the `resultantring` method needs to be updated to include all possible resultant rings constructed from the new ring and existing rings. The other three methods need to be implemented as well.

We now discuss the more arithmetic methods included in the algebra package beginning with the `PolynomialRing` class.

`function PolynomialRing:decompose()` `return table<number, PolynomialRing`

Returns a list of polynomials that form a complete decomposition of the given polynomial. For example:

```
\begin{CAS}
    f = Poly({5,-4,5,-2,1})
    d = f:decompose()
\end{CAS}
\[ \left\{ \lprint{d} \right\} \]
```

$$\left\{x^2 - x, x^2 + 4x + 5\right\}$$

In particular, the code:

```
g = d[2]:evaluateat(d[1])
tex.print('\\[', g:tolatex(), '\\]')
```

recovers f:

$$x^4 - 2x^3 + 5x^2 - 4x + 5$$

`function PolynomialRing:derivative()` `return PolynomialRing`

Returns the formal derivative of the given polynomial. For example:

```
\begin{CAS}
    f = Poly({1,1,1/2,1/6})
    g = f:derivative()
\end{CAS}
\[ \print{f} \xrightarrow{d/dx}
    \print{g} \]
```

$$\frac{1}{6}x^3 + \frac{1}{2}x^2 + x + 1 \xrightarrow{d/dx} \frac{1}{2}x^2 + x + 1$$

`function PolynomialRing:divisors()` `return table<number, PolynomialRing>`

Returns a list of all monic divisors of positive degree of the polynomial, assuming the polynomial ring is a Euclidean Domain. For example:

```
\begin{CAS}
    vars('x')
    f = topoly(x^4 - 2*x^3 - x + 2)
    d = f:divisors()
\end{CAS}
\[ \left\{ \lprint{d} \right\} \]
```

$$\left\{x - 1, x - 2, x^2 - 3x + 2, x^2 + x + 1, x^3 - 1, x^3 - x^2 - x - 2, x^4 - 2x^3 - x + 2\right\}$$

`poly1 PolynomialRing,..., poly3 PolynomialRing`

`function PolynomialRing:divremainder(poly1)` `return poly2,poly3`

Uses synthetic division to return the quotient (`poly2`) and remainder (`poly3`) of `self/poly1`. For example:

```
\begin{CAS}
    f = Poly({2,2,1})
    g = Poly({1,1})
    q,r = f:divremainder(g)
\end{CAS}
\[ \print{f} = (\print{g})(\print{q})
    + \print{r} \]
```

$$x^2 + 2x + 2 = (x + 1)(x + 1) + 1$$

`poly1 PolynomialRing, poly2 PolynomialRing, ..., poly5 PolynomialRing`

`function PolynomialRing.extendedgcd(poly1,poly2)` **`return poly3, poly4, poly5`**

Given two `PolynomialRing` elements `poly1,poly2` returns:

- `poly3`: the gcd of `poly1,poly2`;
- `poly4,poly5`: the coefficients from Bezout's lemma via the extended gcd.

For example:

```
\begin{CAS}
    vars('x')
    f = topoly((x-1)*(x-2)*(x-3))
    g = topoly((x-1)*(x+2)*(x+3))
    h,a,b = PolynomialRing.extendedgcd(f,g)
\end{CAS}
\[ \print{f*a+g*b} = (\print{f})\left( \print{a} \right) +
    (\print{g})\left(\print{b} \right)\]
```

$$x-1=(x^3-6x^2+11x-6)\left(\frac{1}{60}x+\frac{1}{12}\right)+(x^3+4x^2+x-6)\left(-\frac{1}{60}x+\frac{1}{12}\right)$$

Parsing

The function `gcdext()` is a shortcut to `Polynomial.extendedgcd()`:

```
\begin{CAS}
    f = topoly((x+2)*(x-3))
    g = topoly((x+4)*(x-3))
    h,a,b = gcdext(f,g)
\end{CAS}
\[ \print{h} = (\print{f}) \left( \print{a} \right) +
    (\print{g})\left( \print{b} \right). \]
```

$$x-3=(x^2-x-6)\left(-\frac{1}{2}\right)+(x^2+x-12)\left(\frac{1}{2}\right).$$

`function PolynomialRing:evaluateat(Expression)` **`return Expression`**

Uses Horner's rule to evaluate a polynomial at `Expression`. Typically, the input `Expression` is an `Integer` or `Rational`. For example:

```
\begin{CAS}
    f = Poly({2,2,1})
    p = f:evaluateat(1/2)
\end{CAS}
\[ \left. \print{f} \right|_{x=1/2}
    = \print{p} \]
```

$$\left. x^2+2x+2\right|_{x=1/2}=\frac{13}{4}$$

`function PolynomialRing:factor()` **`return BinaryOperation`**

Factors the given polynomial into irreducible terms over the polynomial ring to which the coefficients belong. For example:

```
\begin{CAS}
    f = Poly({8,24,32,24,10,2})
    a = f:factor()
\end{CAS}
\[ \print{a} \]
```

$$2\left(x+1\right)^{1}\left(x^{2}+2x+2\right)^{2}$$

On the other hand:

```
\begin{CAS}
    f = Poly({Mod(1,5),Mod(0,5),Mod(1,5)})
    a = f:factor()
\end{CAS}
\[ \print{a} \]
```

$$1\left(x+3\right)^{1}\left(x+2\right)^{1}$$

The syntax `f = Poly({Mod(1,5),Mod(0,5),Mod(1,5)})` is awkward. Alternatively, one can use the following instead:

```
\begin{CAS}
    f = Mod(Poly({1,0,1}),5)
    a = f:factor()
\end{CAS}
\[ \print{f} = \print{a} \]
```

$$x^{2}+1=1\left(x+3\right)^{1}\left(x+2\right)^{1}$$

Parsing

The function `factor()` shortcuts `PolynomialRing:factor()`. For example:

```
\begin{CAS}
    f = Poly({8,24,32,24,10,2})
    a = factor(f)
\end{CAS}
\[ \print{a} \]
```

$$2\left(x+1\right)^{1}\left(x^{2}+2x+2\right)^{2}$$

`symbol SymbolExpression`

`function PolynomialRing:freeof(symbol)` `return bool`

Checks the value of the field `PolynomialRing.symbol` against `symbol`; returns `true` if these symbols are not equal, and returns `false` otherwise.

Recall: the default symbol for `Poly` is `'x'`. So, for example:

```
\begin{CAS}
    f = Poly({2,2,1})
    vars('t')
    if f:freeof(t) then
        tex.print('$',f:tolatex(),'$ is free of $',t:tolatex(),'$')
    else
        tex.print('$',f:tolatex(),'$ is bound by $',t:tolatex(),'$')
    end
\end{CAS}
```

$x^{2}+2x+2$ is free of t

`poly1 PolynomialRing,..., poly3 PolynomialRing`

`function PolynomialRing.gcd(poly1,poly2)` `return poly3`

Returns the greatest common divisor of two polynomials in a ring (assuming `poly1,poly2` belong to a Euclidean domain). For example:

```
\begin{CAS}
    vars('x')
    f = topoly((x^2+1)*(x-1))
    g = topoly((x^2+1)*(x+2))
    h = PolynomialRing.gcd(f,g)
\end{CAS}
\[ \gcd(\print{f},\print{g}) = \print{h} \]
```

$$\gcd(x^3 - x^2 + x - 1, x^3 + 2x^2 + x + 2) = x^2 + 1$$

Parsing

The function `gcd()` shortcuts `PolynomialRing.gcd()`. For example:

```
\begin{CAS}
    vars('x')
    f = topoly(x^3 - x^2 + x - 1)
    g = topoly(x^3 + 2*x^2 + x + 2)
    h = gcd(f,g)
\end{CAS}
\[ \gcd(\print{f},\print{g}) = \print{h}.\]
```

$$\gcd(x^3 - x^2 + x - 1, x^3 + 2x^2 + x + 2) = x^2 + 1.$$

`function PolynomialRing:isatomic()` `return false`

`function PolynomialRing:isconstant()` `return false`

The inheritances from `ConstantExpression` are overridden for the `PolynomialRing` class.

`poly1 PolynomialRing, poly2 PolynomialRing`

`function PolynomialRing.monicgcdremainders(poly1,poly2)` `return table<number, Ring>`

Given two polynomials `poly1` and `poly2`, returns a list of the remainders generated by the monic Euclidean algorithm.

```
\begin{CAS}
  vars('x')
  f = topoly(x^13-1)
  g = topoly(x^8-1)
  r = PolynomialRing.monicgcdremainders(f,g)
\end{CAS}
\luaexec{
  for i=1,\#r do
    tex.print('\\[', r[i]:tolatex(), '\\]')
  end
}
```

$$x^{13} - 1$$

$$x^8 - 1$$

$$x^5 - 1$$

$$x^3 - 1$$

$$x^2 - 1$$

$$x - 1$$

`function PolynomialRing.mul_rec(poly1,poly2)` `return PolynomialRing`

Performs Karatsuba multiplication without constructing new polynomials recursively. But grade-school multiplication of polynomials is actually faster here up to a very large polynomial size due to Lua's overhead.

`g PolynomialRing, f PolynomialRing, ffactors BinaryOperation`

`function PolynomialRing.partialfractions(g,f,ffactors)` **`return BinaryOperation`**

Returns the partial fraction decomposition of the rational function `g/f` given `PolynomialRing`s `g, f`, and some (not necessarily irreducible) factorization `ffactors` of `f`. If the factorization is omitted, the irreducible factorization is used. The degree of `g` must be less than the degree of `f`.

```
\begin{CAS}
    g = topoly(4*x^2+2*x+2)
    f = topoly((x^2+1)^2*(x+1))
    a = PolynomialRing.partialfractions(g,f)
\end{CAS}
\[ \print{g/f} = \print*{a} \]
```

$$\frac{4x^2+2x+2}{x^5+x^4+2x^3+2x^2+x+1} = \frac{1}{1+x} + \frac{2x}{\left(1+x^2\right)^2} + \frac{1-x}{1+x^2}$$

Parsing

The function `parfrac()` shortcuts the more long winded `PolynomialRing.partialfractions()`. Additionally, the `parfrac` function will automatically try to convert the first two arguments to the `PolynomialRing` type via `topoly()`.

```
\begin{CAS}
    g = 4*x^2+2*x+2
    f = (x^2+1)^2*(x+1)
    a = parfrac(g,f)
\end{CAS}
\[ \print{g/f} = \print*{a} \]
```

$$\frac{4x^2+2x+2}{\left(x^2+1\right)^2(x+1)} = \frac{1}{1+x} + \frac{2x}{\left(1+x^2\right)^2} + \frac{1-x}{1+x^2}$$

`remaining PolynomialRing, roots table<number,PolynomialRing>`

`function PolynomialRing:rationalroots()` **`return remaining, roots`**

This method finds the factors of `PolynomialRing` (up to multiplicity) that correspond to rational roots; these factors are stored in a table `roots` and returned in the second output of the method. Those factors are then divided out of `Polynomialring`; the `PolynomialRing` that remains is returned in the first output of the method. For example:

```
\begin{CAS}
  f = topoly((x-1)^2*(x+1)*(x^2+1))
  g,r = f:rationalroots()
\end{CAS}
The factors of $f$ corresponding to
↪  rational roots are:
\luaexec{
  for i =1, \#r do
    tex.print('\\[', r[i]:tolatex(), '\\]')
  end
}
The part of $f$ that remains after dividing
↪  out these linear terms is:
\[ \print{g} \]
```

The factors of f corresponding to rational roots are:

$$x-1$$

$$x-1$$

$$x+1$$

The part of f that remains after dividing out these linear terms is:

$$x^2+1$$

`function PolynomialRing:roots()` `return table<number, Expression`

Returns a list of roots of `PolynomialRing`, simplified up to cubics. For example:

```
\begin{CAS}
    f = topoly(x^6 + 3*x^5 + 6*x^4 + 7*x^3 + 6*x^2 + 3*x + 2)
    r = f:roots()
\end{CAS}
$ \left\{ \lprint{r} \right\}$
```

$\left\{-\frac{1}{2}+\frac{\sqrt{-1+2\sqrt{3}i}}{2},-\frac{1}{2}-\frac{\sqrt{-1+2\sqrt{3}i}}{2},-\frac{1}{2}+\frac{\sqrt{-1-2\sqrt{3}i}}{2},-\frac{1}{2}-\frac{\sqrt{-1-2\sqrt{3}i}}{2},-\frac{1}{2}+\frac{\sqrt{7}i}{2},-\frac{1}{2}-\frac{\sqrt{7}i}{2}\right\}$

If the decomposition of `PolynomialRing` (or a factor thereof) is not a chain of cubics or lesser degree polynomials, then `RootExpression` is returned instead. For example:

```
\begin{CAS}
    f = topoly(x^6 + x^5 - x^4 + 2*x^3 + 4*x^2 - 2)
    r = f:roots()
\end{CAS}
\[ \left\{ \lprint{r} \right\} \]
```

$$\left\{-\frac{1}{2}+\frac{\sqrt{5}}{2},-\frac{1}{2}-\frac{\sqrt{5}}{2},\mathrm{RootOf}\left(x^4+2x+2\right)\right\}$$

Parsing

The function `roots()` shortcuts `PolynomialRing:roots()`. Also, the function `roots` attempts to cast the argument as a polynomial automatically using `topoly()`. For example:

```
\begin{CAS}
  f = x^6+x^5-x^4+2*x^3+4*x^2-2
  r = roots(f)
\end{CAS}
$ \left\{ \lprint{r} \right\}$
```

$\left\{-\frac{1}{2}+\frac{\sqrt{5}}{2},-\frac{1}{2}-\frac{\sqrt{5}}{2},\mathrm{RootOf}\left(x^4+2x+2\right)\right\}$

`a PolynomialRing, b PolynomialRing`

`function PolynomialRing.resultant(a,b)` `return Field`

Returns the resultant of two polynomials `a,b` in the same ring, whose coefficients are all part of a field. For example:

```
\begin{CAS}
    f = topoly(x^2-2*x+1)
    g = topoly(x^2+2*x-3)
    r = PolynomialRing.resultant(f,g)
\end{CAS}
\[ \operatorname{res}(f,g) = \print{r} \]
```

$$\operatorname{res}(f,g) = 0$$

`function PolynomialRing:squarefreefactorization()` `return BinaryOperation`

Returns the square-free factorization of a polynomial defined over the rationals.

```
\begin{CAS}
    vars('x')
    f = topoly(x^7 - 13*x^6 + 66*x^5 - 158*x^4 + 149*x^3 + 63*x^2 - 216*x + 108)
    s = f:squarefreefactorization()
\end{CAS}
\[ \print{s} \]
```

$$1\left(x^2-1\right)^1\left(x-2\right)^2\left(x-3\right)^3$$

If the polynomial is defined over $\mathbf{Z}/p\mathbf{Z}$ (where p is prime), then the method `modularsquarefreefactorization()` should be used.

Parsing

The function `factor()` has an optional boolean argument that if set to `true` returns `squarefreefactorization()` or `modularsquarefreefactorization()` (as appropriate). For example:

```
\begin{CAS}
    f = topoly(x^6 + 2*x^5 + 4*x^4 + 4*x^3 + 5*x^2 + 2*x + 2)
    s = factor(f,true)
\end{CAS}
\[ \print{s} \]
```

$$1\left(x^2+2x+2\right)^1\left(x^2+1\right)^2$$

And also:

```
\begin{CAS}
    f = topoly(x^6 + 2*x^5 + 4*x^4 + 4*x^3 + 5*x^2 + 2*x + 2)
    f = Mod(f,5)
    s = factor(f,true)
\end{CAS}
\[ \print{s} \]
```

$$1\left(x-1\right)^1\left(x+2\right)^2\left(x+3\right)^3$$

a Integer, b Integer

`function Integer.gcd(a,b)` `return Integer`

Returns the greatest common divisor of `a,b`. For example:

```
\begin{CAS}
    a = 408
    b = 252
    c = Integer.gcd(a,b)
\end{CAS}
\[ \gcd(a,b) = \print{c} \]
```

$$\gcd(a,b) = 12$$

Parsing

The function `gcd()` shortcuts `Integer.gcd()`. For example:

```
\begin{CAS}
    a = 408
    b = 252
    c = gcd(a,b)
\[ \gcd(a,b) = \print{c} \]
```

$$\gcd(a,b) = 12$$

a Integer, b Integer

`function Integer.extendedgcd(a,b)` `return Integer, Integer, Integer`

Returns the greatest common divisor of `a,b` as well as Bezout's coefficients via extended gcd. For example:

```
\begin{CAS}
    a = 408
    b = 252
    c,x,y = Integer.extendedgcd(a,b)
\end{CAS}
\[ \gcd(a,b) = \print{c} = \print{a}(\print{x}) + \print{b}(\print{y}) \]
```

$$\gcd(a,b) = 12 = 408(-8) + 252(13)$$

Parsing

The function `gcdext()` shortcuts `Integer.extendedgcd()`. For example:

```
\begin{CAS}
    a = 408
    b = 252
    c,x,y = gcdext(a,b)
\end{CAS}
\[ \gcd(a,b) = \print{c} = \print{a}(\print{x}) + \print{b}(\print{y}) \]
```

$$\gcd(a,b) = 12 = 408(-8) + 252(13)$$

a Integer, b Integer

`function Integer.max(a,b)` `return Integer, Integer` `a Integer, b Integer`

`function Integer.min(a,b)` `return Integer, Integer`

Returns the max/min of `a,b`; the second output is the min/max (respectively).

```
\begin{CAS}
    a = 8
    b = 7
    c = Integer.max(a,b)
\end{CAS}
\[ \max(\print{a},\print{b}) = \print{c} \]
```

$$\max(8,7) = 8$$

`a Integer, b Integer`

`function Integer.absmax(a,b)` `return Integer, Integer, number`

Methods for computing the larger magnitude of two integers. Also returns the other integer for sorting purposes, and the number -1 if the two values were swapped, 1 if not.

`a Integer, base Integer`

`function Integer.ceillog(a,base)` `return Integer`

Returns the ceiling of the log base (defaults to 10) of a. In other words, returns the least n such that $(\mathtt{base})^{\mathtt{n}} > \mathtt{a}$.

```
\begin{CAS}
    a = 101
    b = 10
    c = Integer.ceillog(a,b)
\end{CAS}
\[ \print{c} \]
```

$$3$$

`a Integer,..., n Integer`

`function Integer.powmod(a,b,n)` `return Integer`

Returns the `Integer` c such that $c \equiv a^b \bmod n$. This should be used when a^b is potentially large.

```
\begin{CAS}
    a = 12341
    b = 2^16+1
    p = 62501
    c = Integer.powmod(a,b,p)
\end{CAS}
\[ \print{c} \equiv \print{a}^{\print{b}} \bmod{\print{p}} \]
```

$$47275 \equiv 12341^{65537} \bmod 62501$$

`b Integer`

`function Integer:divremainder(b)` `return Integer, Integer`

Returns the quotient and remainder over the integers. Uses the standard base 10 long division algorithm.

```
\begin{CAS}
    a = 408
    b = 252
    q,r = Integer.divremainder(a,b)
\end{CAS}
\[ \print{a} = \print{b} \cdot \print{q} + \print{r} \]
```

$$408 = 252 \cdot 1 + 156$$

`function Integer:asnumber()` `return number`

Returns the integer as a floating point number. Can only approximate the value of large integers.

`function Integer:divisors()` `return table<number, Integer>`

Returns all positive divisors of the integer. Not guaranteed to be in any order.

```
\begin{CAS}
    a = 408
    d = a:divisors()
\end{CAS}
\[ \left\{ \lprint{d} \right\} \]
```

$$\left\{1, 2, 4, 8, 3, 6, 12, 24, 17, 34, 68, 136, 51, 102, 204, 408\right\}$$

`function Integer:primefactorization()` `return BinaryOperation`

Returns the prime factorization of the integer as a `BinaryOperation`.

```
\begin{CAS}
    a = 408
    pf = a:primefactorization()
\end{CAS}
\[ \print{pf} \]
```

$$17^{1}2^{3}3^{1}$$

`function Integer:findafactor()` `return Integer`

Return a non-trivial factor of `Integer` via Pollard Rho, or returns `Integer` if `Integer` is prime.

```
\begin{CAS}
    a = 4199
    f = a:findafactor()
\end{CAS}
\[ \print{f} \mid \print{a} \]
```

$$13 \mid 4199$$

`function Integer:isprime()` `return bool`

Uses Miller-Rabin to determine whether `Integer` is prime up to a very large number.

```
\begin{CAS}
    p = 7038304939
    if p:isprime() then
        tex.print(p:tolatex(), "is prime!")
    end
\end{CAS}
```

7038304939 is prime!

`function Rational:reduce()` `return Rational`

Reduces a rational expression of integers to standard form. This method is called automatically when a new `Rational` expression is constructed:

```
\begin{CAS}
    a = Rational(8,6)
\end{CAS}
\[ \print{a} \]
```

$$\frac{4}{3}$$

`function Rational:tocompoundexpression()` `return BinaryOperation`

Converts a `Rational` expression into the corresponding `BinaryOperation` expression.

`function Rational:asnumber()` `return number`

Returns the given rational as an approximate floating point number. Going the other way, the parser in `\begin{CAS}..\end{CAS}` will convert decimals (as written) to fractions. For example:

```
\begin{CAS}
    a = 0.375
\end{CAS}
\[ \print{a} \]
```

$$\frac{3}{8}$$

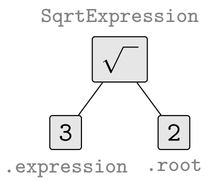
`function SqrtExpression:topower()` `return BinaryOperation`

Converts a `SqrtExpression` to the appropriate `BinaryOperation`. For example, consider:

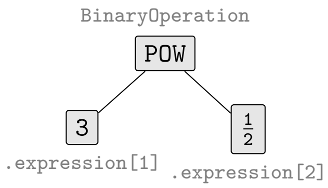
```
\begin{CAS}
    a = sqrt(3)
    b = a:topower()
\end{CAS}
```

Then:

Expression shrub for `a`:



Expression shrub for `b`:



`var SymbolExpression`

`function Equation:solvefor(var)` `return Equation`

Attempts to solve the equation for a particular variable.

```
\begin{CAS}
    vars("x", "y", "z")
    lhs = e ^ (x^2 * y)
    rhs = z + 1
    eq = Equation(lhs, rhs):autosimplify()
    eqx = eq:solvefor(x)
\end{CAS}
\[ \print{eq} \to \print{eqx} \]
```

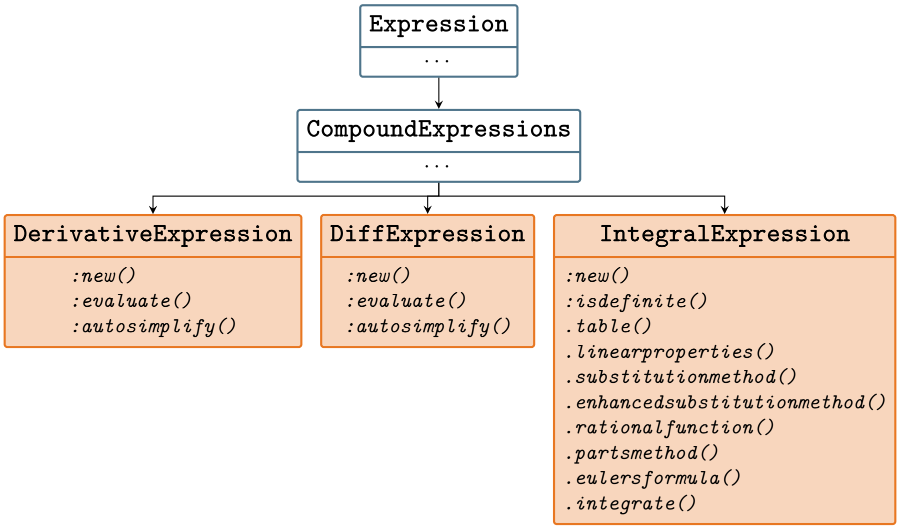
$$e^{x^2 y} = 1 + z \to x = \sqrt{\frac{\ln(1+z)}{y}}$$

6 Calculus

This section contains reference materials for the calculus functionality of `luacas`. The classes in this module are diagramed below according to inheritance along with the methods/functions one can call upon them.

- *`method`*: an abstract method;
- *`method`*: a method that returns the expression unchanged;
- *`method`*: method that is either unique, implements an abstract method, or overrides an abstract method;
- `Class` : a concrete class.

Here is an inheritance diagram of the classes in the calculus module; all these classes inherit from the `CompoundExpression` branch of the inheritance tree. Most methods are stated, but some were omitted (because they inherit in the obvious way, they are auxiliary and not likely to be interesting to the end-user, etc).



6.1 Calculus Classes

There are only a few classes (currently) in the calculus module all of which are concrete:

- `DerivativeExpression`
- `DiffExpression`
- `IntegralExpression`

```
                                 expression Expression, symbol SymbolExpression
function DerivativeExpression:new(expression, symbol)    return DerivativeExpression
```

Creates a new single-variable derivative operation of the given `expression` with respect to the given `symbol`. If `symbol` is omitted, then `symbol` takes the default value of `SymbolExpression("x")`. For example:

```
vars('x')
f = DerivativeExpression(sin(x)/x)
tex.print('\\[', f:tolatex(), '\\]')
```

$$\frac{d}{dx}\left(\frac{\sin(x)}{x}\right)$$

Parsing

The function `DD()` shortcuts `DerivativeExpression()`.

```
\begin{CAS}
    vars('x')
    f = DD(sin(x)/x)
\end{CAS}
\[ \print{f} \]
\end{CAS}
```

$$\frac{d}{dx}\left(\frac{\sin(x)}{x}\right)$$

Alternatively, one could also use `diff()` (see below).

```
                      expression Expression, symbols table<number, Symbol>
function DiffExpression:new(expression, symbols)         return DiffExpression
```

Creates a new multi-variable higher-order derivative operation of the given `expression` with respect to the given `symbols`. As opposed to `DerivativeExpression`, the argument `symbols` cannot be omitted. For example:

```
vars('x','y')
f = DiffExpression(sin(x*y)/y,{x,y})
tex.print('\\[', f:tolatex(), '\\]')
```

$$\frac{\partial^2}{\partial y \partial x}\left(\frac{\sin(xy)}{y}\right)$$

We can also use `DiffExpression` to create higher-order single variable derivatives:

```
vars('x')
f = DiffExpression(sin(x)/x,{x,x})
tex.print('\\[', f:tolatex(), '\\]')
```

$$\frac{d^2}{dx^2}\left(\frac{\sin(x)}{x}\right)$$

Parsing

The function `diff()` shortcuts `DiffExpression()`. The arguments of `diff()` can also be given in a more user-friendly, compact form. For example:

```
\begin{CAS}
    vars('x','y')
    f = diff(sin(x)/x, {x,2})
    g = diff(sin(x*y)/y,x,{y,2})
\end{CAS}
\[ \print{f} = \print*{f} \qquad \print{g} = \print*{g} \]
```

$$\frac{d^2}{dx^2}\left(\frac{\sin(x)}{x}\right) = -\frac{2\cos(x)}{x^2} + \frac{2\sin(x)}{x^3} - \frac{\sin(x)}{x} \qquad \frac{\partial^3}{\partial y^2 \partial x}\left(\frac{\sin(xy)}{y}\right) = -x^2\cos(xy)$$

```
                                expression Expression, symbol SymbolExpression, lower Expression, upper Expression
function IntegralExpression:new(expression,symbol,lower,upper)   return IntegralExpression
```

Creates a new integral operation of the given `expression` with respect to the given `symbol` over the given `lower` and `upper` bounds. If `lower` and `upper` are omitted, then an *indefinite* `IntegralExpression` is constructed. For example:

```
vars('x')
f = IntegralExpression(sin(sqrt(x)), x)
g = IntegralExpression(sin(sqrt(x)), x,
↪  Integer.zero(), pi)
tex.print('\\[', f:tolatex(), '\\]')
tex.print('\\[', g:tolatex(), '\\]')
```

$$\int \sin\left(\sqrt{x}\right)\, dx$$

$$\int_0^{\pi} \sin\left(\sqrt{x}\right)\, dx$$

Parsing

The function `int()` shortcuts `IntegralExpression()`. For example:

```
\begin{CAS}
    g = int(sin(sqrt(x)),x,0,pi)
\end{CAS}
\[ \print{g} = \print*{g} \]
```

$$\int_0^{\pi} \sin\left(\sqrt{x}\right)\, dx = -2\sqrt{\pi}\cos\left(\sqrt{\pi}\right) + 2\sin\left(\sqrt{\pi}\right)$$

6.2 Calculus Methods

`integral IntegralExpression`

`function IntegralExpression.table(integral)` `return Expression|nil`

Attempts to integrate `integral.expression` with respect to `integral.symbol` by checking a table of basic integrals; returns nil if the integrand isn't in the table. For example:

```
\begin{CAS}
  vars('x')
  f = int(cos(x),x)
  f = f:table()
  g = int(x*cos(x),x)
  g = g:table()
\end{CAS}
\[ f = \print{f} \qquad g = \print{g} \]
```

$$f = \sin(x) \qquad g = nil$$

The table of integrals consists of power functions, exponentials, logarithms, trigonometric, and inverse trigonometric functions.

`integral IntegralExpression`

`function IntegralExpression.linearproperties(integral)` `return Expression|nil`

Attempts to integrate `integral.expression` with respect to `integral.symbol` by using linearity properties (e.g. the integral of a sum/difference is the sum/difference of integrals); returns nil if any individual component cannot be integrated using `IntegralExpression:integrate()`. For example:

```
\begin{CAS}
  vars('x')
  f = int(sin(x) + e^x,x)
  g = f:table()
  f = f:linearproperties()
\end{CAS}
\[ f = \print*{f} \qquad g = \print*{g} \]
```

$$f = e^x - \cos(x) \qquad g = nil$$

`integral IntegralExpression`

`function IntegralExpression.substitutionmethod(integral)` `return Expression|nil`

Attempts to integrate `integral.expression` with respect to `integral.symbol` via u-substitution; returns nil if no suitable substitution is found to be successful.

```
\begin{CAS}
  vars('x')
  f = int(x*e^(x^2),x)
  g = int(x*e^x,x)
  f = f:substitutionmethod()
  g = g:substitutionmethod()
\end{CAS}
\[ f = \print*{f} \qquad g = \print*{g}.\]
```

$$f = \frac{e^{x^2}}{2} \qquad g = nil.$$

`integral IntegralExpression`

`function IntegralExpression.enhancedsubstitutionmethod(integral)` `return Expression|nil`

Attempts integrate `integral.expression` with respect to `integral.symbol` via u-substitutions. This method distinguishes itself from the `.substitutionmethod` by attempted to solve $u = g(x)$ for the original variable and then substituting the result into the expression. This behavior is not included in `.substitutionmethod` due to speed concerns. For example:

```
\begin{CAS}
  vars('x')
  f = int(x^5*sqrt(x^3+1),x)
  g = f:substitutionmethod()
  h = f:enhancedsubstitutionmethod()
\end{CAS}
\[ g= \print*{g} \]
\[ h= \print*{h} \]
```

$$g = nil$$

$$h = -\frac{2\left(1+x^3\right)^{\frac{3}{2}}}{9} + \frac{2\left(1+x^3\right)^{\frac{5}{2}}}{15}$$

`function IntegralExpression.trialsubstitutions(Expression) return table<number, Expression`

Generates a list of possible u-substitutions to attempt in `substitutionmethod()` and `enhancedsubstitutionmethod()`. For example:

```
\begin{CAS}
  vars('x')
  f = cos(x)/(1+sin(x))
  f = f:autosimplify()
  l = IntegralExpression.trialsubstitutions(f)
\end{CAS}
$\left\{ \lprint{l} \right\}$.
```

$\left\{ \frac{\cos(x)}{1+\sin(x)}, \cos(x), \frac{1}{1+\sin(x)}, 1+\sin(x), \sin(x) \right\}$.

`function IntegralExpression.rationalfunction(IntegralExpression)     return Expression|nil`

Integrates `integrand` with respect to `symbol` via Lazard, Rioboo, Rothstein, and Trager's method in the case when `expression` is a rational function in the variable `symbol`. If `integrand` is not a rational function, then nil is returned.

```
\begin{CAS}
    vars('x')
    f = (x^2+2*x+2)/(x^2+3*x+2)
    f = f:autosimplify()
    g = int(f,x):rationalfunction()
\end{CAS}
\[ \int \print{f}\ dx = \print*{g} \]
```

$$\int \frac{2+2x+x^2}{2+3x+x^2}\ dx = x + \ln(1+x) - 2\ln(2+x)$$

In some cases, the `.rationalfunction` method returns non-standard results. For example:

```
\begin{CAS}
  vars('x')
  num = x^2
  den = ((x+1)*(x^2+2*x+2)):expand()
  f = (num/den):autosimplify()
  f = int(f,x):rationalfunction()
\end{CAS}
\[ \print{simplify(f)} \]
```

$$\ln(1+x) - i\ln(1+i+x) + i\ln(1-i+x)$$

On the other hand:

```
\begin{CAS}
  pfrac = parfrac(num,den)
\end{CAS}
\[ \print*{int(pfrac,x)} \]
```

$$-2\arctan(1+x) + \ln(1+x)$$

`function IntegralExpression.partsmethod(IntegralExpression)` `return Expression|nil`

Attempts to integrate `integral.expression` with respect to `integral.symbol` via *integration by parts*; returns nil if no suitable application of IBP is found. For example:

```
\begin{CAS}
    vars('x')
    a = int(x*e^x,x)
    b = a:partsmethod()
    c = int(e^(x^2),x)
    d = c:partsmethod()
\end{CAS}
\[ b=\print*{b} \]
\[ d=\print*{d} \]
```

$$b = -e^x + e^x x$$

$$d = nil$$

`integral IntegralExpression`

`function IntegralExpression.eulersformula(integral)` `return Expression|nil`

Attempts to integrate `integral.expression` with respect to `integral.symbol` by using the Euler formulas:

$$\cos x = \frac{e^{ix} + e^{-ix}}{2} \qquad \sin x = \frac{e^{ix} - e^{-ix}}{2i}.$$

Per usual, this method returns nil if such a method is unsuccessful (or if the integrand is unchanged after applying the above substitutions). This can often be used as an alternative for integration by parts. For example:

```
\begin{CAS}
  vars('x')
  a = int(e^x*sin(x),x)
  b = int(x^2,x)
  c = a:eulersformula()
  d = b:eulersformula()
\end{CAS}
\[ c= \print*{c} \]
\[ d= \print*{d} \]
```

$$c = -\frac{e^x \cos(x)}{2} + \frac{e^x \sin(x)}{2}$$

$$d = nil$$

`integral IntegralExpression`

`function IntegralExpression.integrate(integral)` `return Expression|nil`

Recursive part of the indefinite integral operator; returns nil if the expression could not be integrated. The methods above get called (roughly) in the following order:

(i) `.table`

(ii) `.linearproperties`

(iii) `.substitutionmethod`

(iv) `.rationalfunction`

(v) `.partsmethod`

(vi) `.eulersformula`

(vii) `.enhancedsubstitutionmethod`

Between (vi) and (vii), the `.integrate` method will attempt to expand the integrand and retry. The method is recursive in the sense that (most) of the methods listed above will call `.integrate` at some point. For example, after a list of trial substitutions is created, the method `.substitutionmethod` will call `.integrate` to determine whether the new integrand can be integrated via the methods in the above list.

Parsing

Recall the function `int()` which acts as a shortcut for `IntegralExpression:new()`. When `:autosimplify()` is called upon an `IntegralExpression`, then `IntegralExpression.integrate` is applied. If `nil` is returned, then `:autosimplify()` returns `self`; otherwise the result of `.integrate` is returned and evaluated over the bounds, if any are given. For example:

```
\begin{CAS}
    vars('x')
    f = cos(x)*e^(sin(x))
    f = int(f,x,0,pi/2)
\end{CAS}
\[ \print{f} = \print*{f}\]
```

$$\int_0^{\frac{\pi}{2}} \cos(x)\, e^{\sin(x)}\, dx = -1 + e$$

On the other hand:

```
\begin{CAS}
    vars('x')
    f = e^(e^x)
    f = int(f,x,0,1)
\end{CAS}
\[ \print{f} = \print*{f} \]
```

$$\int_0^1 e^{e^x}\, dx = \int_0^1 e^{e^x}\, dx$$

`function IntegralExpression:isdefinite()` `return bool`

Returns `true` of `IntegralExpression` is definite (i.e. if `.upper` and `.lower` are defined fields), otherwise returns `false`.

A The LaTeX code

As noted above, this package is really a Lua program; the package `luacas.sty` is merely a shell to make accessing that Lua program easy and manageable from within LaTeX.

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{luacas}
    [2023/05/26 v1.0.2 CAS written in Lua for LaTeX]
```

We check to make sure the user is compiling with LuaLaTeX; if not, an error message is printed and compilation is aborted.

```
\RequirePackage{iftex}
\ifluatex
  \RequirePackage{luacode}
\else
  {\PackageError{luacas}
  {Not running under LuaLaTeX}
  {This package requires LuaLaTeX. Try compiling this document with\MessageBreak
  ↪  'lualatex' instead of 'latex'. This is a fatal error; I'm aborting now.}%
  }\stop
\fi
```

The following packages are required for various macros:

```
\RequirePackage{xparse}
\RequirePackage{pgfkeys}
\RequirePackage{verbatim}
\RequirePackage{tikz}
\RequirePackage{xcolor}
\RequirePackage{mathtools}
```

The files `helper.lua` and `parser.lua` help bridge the gap between the Lua program and LaTeX.

```
\directlua{require('test.parser')
           require('test.helper')
}
```

We now define the `\begin{CAS}..\end{CAS}` environment:

```
\NewDocumentEnvironment{CAS}%
  {+b}%
  {\luaexec{CASparse([[#1]])}}%
  {}
```

Note: The contents are wrapped in the function `CASparse()`. We now define the retrieving macros `\get`, `\fetch`, and `\store`:

```
\newcommand{\get}%
  [2][true]%
  {\directlua{disp(#2, #1)}}

\newcommand{\fetch}[1]{
  \directlua{tex.print(tostring(#1))}
}

\NewDocumentCommand{\store}{m O{#1}}{
  \expandafter\def\csname #2\endcsname{%
    \directlua{
      input = #1
      if not input[1] then
        tex.sprint{tostring(input)}
      else
        tex.sprint("{")
        for _,entry in ipairs(input) do
            tex.sprint(tostring(entry),",")
        end
        tex.sprint("}")
      end
    }%
  }%
}%
```

And now we define the printing macros `\print`, `\vprint`, and `\lprint`:

```
\NewDocumentCommand{\print}{s m}{%
  \IfBooleanTF{#1}{%
    \directlua{
      local sym = #2
      if sym then
        tex.print(sym:autosimplify():tolatex())
      else
        tex.print('nil')
      end
    }%
  }{%
    \directlua{
      local sym = #2
      if sym then
        tex.print(sym:tolatex())
      else
        tex.print('nil')
      end
    }%
  }%
}

\NewDocumentCommand{\vprint}{s m}{%
  \IfBooleanTF{#1}{%
    \directlua{
      local sym = #2
      tex.sprint([[\unexpanded{\begin{verbatim}]] .. tostring(sym) ..
       ↪  [[\end{verbatim}}]])
    }%
  }{%
    \directlua{
      local sym = #2
      tex.sprint([[\unexpanded{\begin{verbatim}]] .. tostring(sym:autosimplify()) ..
       ↪  [[\end{verbatim}}]])
    }%
  }%
}

\NewDocumentCommand{\lprint}{m O{nil,nil}}{%
  \luaexec{
    local tbl = #1
    local low,upp = #2
    local tmp =0
    if tbl[0] == nil then
      tmp = 1
    end
    upp = upp or \#tbl
    low = low or tmp
    for i=low,upp do
      tex.print(tbl[i]:tolatex())
      if tbl[i+1] then
          tex.print(",")
      end
    end
  }
}
```

And finally, we define the macros useful for printing expression trees:

```
%prints the first level of an expression tree; for use within a tikzpicture environment

\NewDocumentCommand{\printshrub}{s m}{%
  \IfBooleanTF{#1}{%
    \directlua{
      local sym = #2
      sym = sym:autosimplify()
      tex.print("\\node [label=90:", whatis(sym), "] {", nameof(sym), "}")
      tex.print(sym:gettheshrub())
      tex.print(";")
    }%
  }{%
    \directlua{
      local sym = #2
      tex.print("\\node [label=90:", whatis(sym), "] {", nameof(sym), "}")
      tex.print(sym:gettheshrub())
      tex.print(";")
    }%
    }
}

%prints the full expression tree; for use within a tikzpicture environment

\NewDocumentCommand{\printtree}{s m}{%
  \IfBooleanTF{#1}{%
    \luaexec{
      local sym = #2
      sym = sym:autosimplify()
      tex.print("\\node {",nameof(sym),"}")
      tex.print(sym:getthetree())
      tex.print(";")
    }%
  }{%
    \luaexec{
      local sym = #2
      tex.print("\\node {",nameof(sym),"}")
      tex.print(sym:getthetree())
      tex.print(";")
    }%
  }
}

%parses an expression tree for use within the forest environment; result is stored in \forestresult

\NewDocumentCommand{\parseforest}{s m}{%
  \IfBooleanTF{#1}{%
    \luaexec{
      local sym = #2
      sym = sym:autosimplify()
      tex.print("\\def\\forestresult{")
      tex.print("[")
      tex.print(nameof(sym))
      tex.print(sym:gettheforest())
      tex.print("]")
      tex.print("}")
    }%
  }{%
    \luaexec  {
      local sym = #2
      tex.print("\\def\\forestresult{")
      tex.print("[")
      tex.print(nameof(sym))
      tex.print(sym:gettheforest())
      tex.print("]")
      tex.print("}")
    }%
  }
}

\NewDocumentCommand{\parseshrub}{s m}{%
  \IfBooleanTF{#1}{%
    \luaexec{
      local sym = #2
      sym = sym:autosimplify()
      tex.print("\\def\\shrubresult{")
      tex.print("[")
      tex.print(nameof(sym))
      tex.print(", tikz+={\\node[anchor=south] at (.north) {test};}")
      tex.print(sym:getthefancyshrub())
      tex.print("]")
      tex.print("}")
    }%
  }{%
    \luaexec{
      local sym = #2
      tex.print("\\def\\shrubresult{")
      tex.print("[")
      tex.print(nameof(sym))
      tex.print(", tikz+={\\node[anchor=south  font=\\ttfamily\\footnotesize,gray] at
       ↪  (.north) {",longwhati  (sym),"};}")
      tex.print(sym:getthefancyshrub())
      tex.print("]")
      tex.print("}")
    }%
  }
}

\NewDocumentCommand{\whatis}{m}{%
  \luaexec{
    tex.sprint("{\\ttfamily",longwhatis(#1),"}")
  }%
}
```

B Version History

v1.0.2

- Fix Polynomial Rings displaying redundant ones in LaTeX
- Fix rational forced non-simplification not working
- Added ability to load LuaCAS modules as local variables
- Fix arithmetic with decimal expressions

v1.0.1

- Update CAS file names for TeXLive

v1.0.0

- Intial release