original: July 2006
this version: November 2nd, 2012
AEI-2006-038

Cadabra

A field-theory motivated approach to symbolic computer algebra

Tutorial and reference guide

WARNING: This document refers to versions 1.x of the Cadabra software. Most of the logic is unchanged in 2.x, but you will need to adapt the syntax. An updated document will be released soon(ish).

Cadabra is a computer algebra system for the manipulation of tensorial mathematical expressions such as they occur in "field theory problems". It is aimed at, but not necessarily restricted to, high-energy physicists. It is constructed as a simple tree-manipulating core, a large collection of standalone algorithmic modules which act on the expression tree, and a set of modules responsible for output of nodes in the tree. All of these parts are written in C++. The input and output formats closely follow TEX, which in many cases means that cadabra is much simpler to use than other similar programs. It intentionally does not contain its own programming language; instead, new functionality is added by writing new modules in C++.

This document contains a description of the core program as well as a detailed listing of the functionality of the currently available algorithm modules. Given the origin of cadabra, the bias is currently towards algorithms for problems in (quantum) field theory, general relativity, group theory and related areas. The last part of this text consists of a user guide on how to implement new algorithmic and display modules and thereby extend the functionality of cadabra.

The software is available for download under the terms of the GNU General Public License from http://cadabra.phi-sci.com/ .

A paper introducing cadabra to high-energy physicists is available as

> "*Introducing Cadabra: a symbolic computer algebra system for field theory problems*",
> Kasper Peeters, also available as preprint hep-th/0701238.

Furthermore, a short paper describing the motivation and key technical aspects of cadabra is available as

> "*A field-theory motivated approach to symbolic computer algebra*",
> Kasper Peeters,
> *Comp. Phys. Comm.* **176** (2007) 550-558 (title changed in print),
> cs.CS/0608005.

If you use cadabra in your own work, please cite these two papers.

http://maths.dur.ac.uk/users/kasper.peeters/

mailto:kasper.peeters@gmail.com

Table of Contents

Overview and motivation

> "*I think that several of you are missing the obvious. The majority of people use these programs as symbolic calculators – occasionally. As such they want their input and output to match what they would have written with pencil and paper.*" `soft-sys.math.maple`, 2002

Cadabra is a computer algebra system for the manipulation of what could loosely be called *tensorial expressions*. It is aimed at, but not necessarily restricted to, theoretical high-energy physicists. The program's interface, storage system and underlying philosophy differ substantially from other computer algebra systems. Its main characteristics are:

- Usage of TEX notation for both input and output, which eliminates many errors in transcribing problems from paper to computer and back.
- Built-in understanding of dummy indices and dummy symbols, including their automatic relabelling when necessary. Powerful algorithms for canonicalisation of objects with index symmetries, both mono-term and multi-term.
- A new way to deal with products of non-commuting objects, enabling a notation which is identical to standard physicist's notation (i.e. no need for special non-commuting product operators).
- A flexible way to associate meaning ("type information") to tensors by attaching them to "properties".
- An optional unlimited undo system. Interactive calculations can be undone to arbitrary level without requiring a full re-evaluation of the entire calculation.
- A simple and documented way to add new algorithms in the form of C++ modules, which directly operate on the internal expression tree.
- A command line interface as well as a graphical one, and a TEXmacs frontend.

This document contains a small tutorial, an extensive reference guide and a technical summary of the program's internals together with a guide on how to write new algorithm modules in C++.

Acknowledgements

This program uses code written by several other people. The tensor monomial canonicalisation routines rely on the `xPerm` code written by José Martin-Garcia [1]). All representation-theory related problems are handled by the `LiE` software by Marc van Leeuwen, Arjeh Cohen and Bert Lisser [2].

The name Cadabra is an implicit acknowledgement to Mees de Roo, who introduced me to his (so far unpublished) Pascal program `Abra` in the fall of 2000. This program has an extremely physicist-friendly way of dealing with fermions and tensor symmetries, and a formula history mechanism still not found in any other comparable computer algebra system. Cadabra was originally planned to be "my private C++ version of `Abra`", and even though it does not show much similarity anymore, the development was to a large extent inspired by `Abra`.

Tutorial

The best way to explain the advantages of a new computer algebra system is to demonstrate its ease-of-use for a real-world problem. But if you lack the patience to even read the tutorial on the next few pages, at least read the remainder of this page for an absolute minimum of information. All input in the examples is prefixed with a "▷" symbol.

- Start the program by typing "`prompt cadabra`". Quit with the "`@quit`" command.
- Tensor expressions are entered as in TEX, with subscripts and superscripts as you know them, e.g.

```
▷ A_{m n} B^{m q} R^{n}_{q};
```

  Input lines are terminated with ";", ":" or "." punctuation; see section 4.1 on page 16 for information on what these characters mean.

- Tensors carry properties, such as "being a Riemann tensor". These properties are attached by using a double-double dot notation,

```
▷ R_{m n p q}::RiemannTensor.
▷ g_{m n}::Metric.
▷ \psi::Spinor.
```

  A list of all properties and the place where they are described in this manual can be found on page 37.

- For many operations cadabra needs to know about the names which it can use for 'dummy' indices. You declare dummy indices as

```
▷ {m,n,p,q}::Indices(vector).
```

  where "`vector`" is a chosen name for this index set. See section 4.5 on page 21.

- For many other operators, cadabra needs to know the range over which indices run. Set these index ranges by attaching the `Integer` property to the indices, e.g.

```
▷ {m,n,p,q}::Integer(0..10).
```

- Expressions can be given a label so you can refer to them again later; you do this by writing the label before the expression and a ":=" in between,

```
▷ MaxwellEom:= \diff{F^{m n}}_{n} = 0;
```

- All things starting with '@' are commands (also called "active nodes"). Some of the frequently used commands are `@substitute` (page 90), `@canonicalise` (page 56) and `@collect_terms` (page 55).

2.1 Web tutorials and sample notebooks

Apart from the tutorials listed below, there is a growing collection of sample notebooks available on the cadabra web site. In addition, help is available through the mailing list.

2.2 Tutorial 1: Tensor monomials and multi-term symmetries

Cadabra contains powerful algorithms to bring any tensorial expression into a canonical form. For multi-term symmetries, cadabra relies on Young tableau methods to generate a canonical form for tensor monomials. [1]

As an example, consider the identity

$$W_{pqrs}W_{ptru}W_{tvqw}W_{uvsw} - W_{pqrs}W_{pqtu}W_{rvtw}W_{svuw} = W_{mnab}W_{npbc}W_{mscd}W_{spda} - \frac{1}{4}W_{mnab}W_{psba}W_{mpcd}W_{nsdc}\,. \quad (2.1)$$

in which W_{mnpq} is a Weyl tensor (all contracted indices have been written as subscripts for easier readability). Proving this identity requires multiple uses of the Ricci cyclic identity,

$$W_{m[npq]} = 0\,. \quad (2.2)$$

With cadabra's Young tableau methods the proof is simple. We first declare our objects and input the identity which we want to prove,

```
▷ {m,n,p,q,r,s,t,u,v,w,a,b,c,d,e,f}::Indices(vector).
▷ W_{m n p q}::WeylTensor.

▷ W_{p q r s} W_{p t r u} W_{t v q w} W_{u v s w}
    - W_{p q r s} W_{p q t u} W_{r v t w} W_{s v u w}
    - W_{m n a b} W_{n p b c} W_{m s c d} W_{s p d a}
    + (1/4) W_{m n a b} W_{p s b a} W_{m p c d} W_{n s d c};
```

Using a Young projector to project all Weyl tensors onto a form which shows the Ricci symmetry in manifest form is done with

```
▷ @young_project_tensor!(%){ModuloMonoterm};
```

This algorithm knows that the Weyl tensor sits in the $\tilde{\square\!\square}$ representation of the rotation group SO(d), and effectively leads to a replacement

$$W_{mnpq} \to \frac{2}{3}W_{mnpq} - \frac{1}{3}W_{mqnp} + \frac{1}{3}W_{mpnq}\,. \quad (2.3)$$

We then expand the products of sums and canonicalise using mono-term symmetries,

[1] The user interface for multi-term symmetries is under active development and will simplify substantially in upcoming releases.

```
▷ @distribute!(%):
▷ @canonicalise!(%):
▷ @rename_dummies!(%):
▷ @collect_terms!(%);
```

The last line produces the expected "zero". A slightly more complicated multi-term example can be found in TEXmacs format in `texmacs/showcase1.tm`.

2.3 Tutorial 2: Tensor monomials, part two

It is easy to generate complete bases of tensor monomials, for any arbitrary tensors (not necessarily for tensors in irreps, as in the previous example). As an example, let us show how this works for a basis constructed from three powers of the Riemann tensor. All that is required is

```
▷ {m,n,p,q,r,s,t,u,v,w,a,b}::Indices(vector).
▷ {m,n,p,q,r,s,t,u,v,w,a,b}::Integer(0..9).

▷ R_{m n p q}::RiemannTensor.

▷ basisR3:= R_{m n p q} R_{r s t u} R_{v w a b};
▷ @all_contractions(%);
```

The result can be further simplified using

```
▷ @canonicalise!(%):
▷ @substitute!(%)( R_{m n m n} -> R ):
▷ @substitute!(%)( R_{m n m p} -> R_{n p} ):
▷ @substitute!(%)( R_{n m p m} -> R_{n p} ):
▷ @substitute!(%)( R_{n m m p} -> R_{n p} );
```

which leads something like[2]

```
basisR3:= \{ R_{m n p q} * R_{m p r s} * R_{n r q s},
             R * R_{q r} * R_{q r},
             R_{n p} * R_{n q p r} * R_{q r},
             R_{n p} * R_{n q r s} * R_{p r q s},
             R * R_{p q r s} * R_{p q r s},
             R_{n p} * R_{n r} * R_{p r},
             R_{m n p q} * R_{m r p s} * R_{n r q s},
             R * R * R \};
```

This result is equivalent to the basis given in the "$\mathcal{R}^0_{6,3}$" table on page 1184 of [3].

[2] The algorithm involves a random step which implies that the basis is not always the same, though it is always complete. Further improvements are in preparation which will eliminate this randomness (and also lead to a speedup).

2.4 Tutorial 3: World-sheet supersymmetry

Cadabra not only deals with bosonic tensors, but also with fermionic objects, i.e. anti-commuting variables. A simple example on which to illustrate their use is the Ramond-Neveu-Schwarz superstring action. We will here show how one uses cadabra to show invariance of this action under supersymmetry (a calculation which is easy to do by hand, but illustrates several aspects of cadabra nicely). We will use a conformal gauge and complex coordinates.

We first define the properties of all the symbols which we will use,

```
▷ {\del{#}, \delbar{#}}::Derivative.
▷ {\Psi_\mu, \Psibar_\mu, \eps, \epsbar}::AntiCommuting.
▷ {\Psi_\mu, \Psibar_\mu, \eps, \epsbar}::SelfAntiCommuting.
▷ {\Psi_\mu, \Psibar_\mu, X_\mu}::Depends(\del,\delbar).
▷ {\Psi_\mu, \Psibar_\mu, \eps, \epsbar, X_\mu, i}::SortOrder.
```

All objects are by default commuting, so the bosons do not have to be declared separately. You can at any time get a list of the declared properties by using the command `@proplist`.

If you try this example in the graphical front-end, `\\del`, `\\eps` and so on would not print correctly as these are not valid LaTeX symbols. However, you can tell cadabra to print such symbols by using the `LaTeXForm` property,

```
▷ \del{#}::LaTeXForm("\partial").
▷ \delbar{#}::LaTeXForm("\bar{\partial}").
▷ \eps::LaTeXForm("\epsilon").
▷ \epsbar::LaTeXForm("\bar{\epsilon}").
▷ \Psibar{#}::LaTeXForm("\bar{\Psi}").
```

If you use the command-line version, these `LaTeXForm` properties are not needed.

Now we have to input the action density (see [4] for the conventions used here)

```
▷ action:= \del{X_\mu} \delbar{X_\mu}
         + i \Psi_\mu \delbar{\Psi_\mu} + i \Psibar_\mu \del{\Psibar_\mu};
```

Observe how we wrapped the `\del` and `\delbar` operators around the objects on which they are supposed to act. We are now ready to perform the supersymmetry transformation. This is done by using the `@vary` algorithm,

```
▷ @vary(%)( X_\mu       ->   i \epsbar \Psi_\mu + i \eps \Psibar_\mu,
            \Psi_\mu    -> - \epsbar \del{X_\mu},
            \Psibar_\mu -> - \eps \delbar{X_\mu} );

▷ @distribute!(%);
▷ @prodrule!(%);
```

The `@prodrule` command has applied the Leibnitz rule on the derivatives, so that the derivative of a product becomes a sum of terms. We expand again the products of sums, and use `@unwrap` to take everything out of the derivatives which does not depend on it,

```
▷ @distribute!(%);
▷ @unwrap!(%);
▷ @prodsort!(%);
```

At this stage we are left with an expression which still contains double derivatives. In order to write this in a canonical form, we eliminate all double derivatives by doing one partial integration. This is done by first marking the derivatives which we want to partially integrate, and then using `@pintegrate`,

```
▷ @substitute!(%)( \del{\delbar{X_{\mu}}} -> \pdelbar{\del{X_{\mu}}} ):
▷ @substitute!(%)( \delbar{\del{X_{\mu}}} -> \pdel{\delbar{X_{\mu}}} ):
▷ @pintegrate!(%){ \pdelbar }:
▷ @pintegrate!(%){ \pdel }:
▷ @rename!(%){"\pdelbar"}{"\delbar"}:
▷ @rename!(%){"\pdel"}{"\del"};
▷ @prodrule!(%);
▷ @distribute!(%);
▷ @unwrap!(%);
▷ @prodsort!(%);
```

Notice how, after the partial integration, we renamed the partially integrated derivatives back to normal ones (and again apply Leibnitz' rule). If we now collect terms,

```
▷ @collect_terms!(%);
```

we indeed find that the total susy variation vanishes.

2.5 Tutorial 4: Super-Maxwell

The following example illustrates the use of a somewhat more complicated set of object properties. A TEXmacs version of this problem can be found in the distribution tarball in the file `texmacs/showcase3.tm`. We start with the super-Maxwell action, given by

$$S = \int \mathrm{d}^4x \Big[-\frac{1}{4}(F_{ab})^2 - \frac{1}{2}\bar{\lambda}\gamma^a\partial_a\lambda \Big] , \tag{2.4}$$

It is supposed to be invariant under the transformations

$$\delta A_a = \bar{\epsilon}\gamma_a\lambda , \quad \delta\lambda = -\frac{1}{2}\gamma^{ab}\epsilon\, F_{ab} . \tag{2.5}$$

The object properties for this problem are

```
▷ { a,b,c,d,e }::Indices(vector).
▷ \bar{#}::DiracBar.
▷ { \partial{#}, \ppartial{#} }::PartialDerivative.
▷ { A_{a}, f_{a b} }::Depends(\partial, \ppartial).
▷ { \epsilon, \gamma_{#} }::Depends(\bar).
▷ \lambda::Depends(\bar, \partial).
```

```
▷ { \lambda, \gamma_{#} }::NonCommuting.
▷ { \lambda, \epsilon }::Spinor(dimension=4, type=Majorana).
▷ { \epsilon, \lambda }::SortOrder.
▷ { \epsilon, \lambda }::AntiCommuting.
▷ \lambda::SelfAntiCommuting.
▷ \gamma_{#}::GammaMatrix(metric=\delta).
▷ \delta{#}::Accent.
▷ f_{a b}::AntiSymmetric.
▷ \delta_{a b}::KroneckerDelta.
```

Note the use of two types of properties: those which apply to a single object, like `Depends`, and those which are associated to a list of objects, like `AntiCommuting`. Clearly $\partial_a\lambda$ and $\bar{\epsilon}$ are anti-commuting too, but the program figures this out automatically from the fact that `\partial` has an associated `PartialDerivative` property associated to it.[3]

The actual calculation is an almost direct transcription of the formulas above. First we define the supersymmetry transformation rules,

```
▷ susy:= { \delta{A_{a}} = \bar{\epsilon} \gamma_{a} \lambda,
           \delta{\lambda} = -(1/2) \gamma_{a b} \epsilon f_{a b} };
```

The action is also written just as it is typed in TEX,

```
▷ S:= -(1/4) f_{a b} f_{a b}
                - (1/2) \bar{\lambda} \gamma_{a} \partial_{a}{\lambda};
```

Showing invariance starts by applying a variational derivative,

```
▷ @vary(%)( f_{a b} ->   \partial_{a}{\delta{A_{b}}}
                       - \partial_{b}{\delta{A_{a}}},
            \lambda -> \delta{\lambda} );

▷ @distribute!(%):
▷ @substitute!(%)( @(susy) ): @prodrule!(%): @distribute!(%): @unwrap!(%);
```

After these steps, the result is (shown exactly as it appears in the TEXmacs [5] frontend)

$$S = \bar{\epsilon}\gamma_a\partial_b\lambda\, f_{ab} + \frac{1}{4}\overline{\gamma_{cb}\epsilon}\gamma_a\partial_a\lambda\, f_{cb} + \frac{1}{4}\bar{\lambda}\gamma_a\gamma_{cd}\epsilon\partial_a f_{cb}\,. \tag{2.6}$$

Since the program knows about the properties of gamma matrices it can rewrite the Dirac bar, and then we do one further partial integration,

```
▷ @rewrite_diracbar!(%);
▷ @substitute!(%)( \partial_{c}{f_{a b}} -> \ppartial_{c}{f_{a b}} ):
▷ @pintegrate!(%){\ppartial}:
▷ @rename!(%){"\ppartial"}{"\partial"}:
▷ @prodrule!(%): @unwrap!(%);
```

[3] This is similar to Macsyma's types and features: the property which is attached to a symbol is like a 'type', while all properties which the symbol inherits from child nodes are like 'features'.

What remains is the gamma matrix algebra, sorting of spinors (which employs inheritance of the `Spinor` and `AntiCommuting` properties as already alluded to earlier) and a final canonicalisation of the index contractions,

```
▷ @join!(%){expand}: @distribute!(%): @eliminate_kr!(%): @prodsort!(%);
▷ @substitute!(%)( \partial_{a}{\bar{\lambda}}
                                        -> \bar{\partial_{a}{\lambda}} );
▷ @spinorsort!(%):
▷ @rename_dummies!(%): @canonicalise!(%): @collect_terms!(%);
```

The result is a Bianchi identity on the field strength, and thus invariance of the action.

Graphical user interface

3.1 General information

The graphical user interface is called `xcadabra`.

Context-sensitive help is available: if you put your cursor on a command or property name and press F1 or the help button, you will get a help screen (which is essentially the corresponding section in this manual).

3.2 Cell types

There are *two* input cell types and *two* output cell types. Normal input cells contain cadabra input. TEX input cells contain comments which you can add to the notebook but which will not be sent to cadabra. Output cells can be either verbatim output, such as status messages, or TEX output of expressions. Output cells are associated to input cells, so if you delete an input cell all the associated output cells will be removed too.

In the current version there is no visual separation between cells. When you start a new blank notebook, your cursor is in the first input cell. After entering an expression there and pressing `shift-enter`, this input cell will be evaluated and all output will be placed below it in the first output cell. In addition, a new input cell will be created below the output cell.

The "edit" menu can be used to add input cells or delete them, and also shows keyboard shortcuts.

3.3 Editing, loading, saving, printing, exporting

The file format used by the front-end is a LATEX compatible file. If necessary, it can be edited by hand using a text editor.

As the file format which the front-end uses to save notebooks is a LATEX compatible file, it is simple to print a notebook: simply save it and run it through LATEX as usual. You will need two special-purpose packages, `breqn` (which is separately packaged and also available from the AMS) and `tableaux` (which is included with the cadabra distribution).

Notebooks can be exported to a normal cadabra text input file. This results in a file which contains only the input cells of the notebook. Such a file can be fed into the command line version of cadabra by using

```
▷ cadabra --input [filename]
```

The default extension for cadabra notebooks is `.cnb`. For cadabra input files (for the command line version of cadabra) it is `.cdb`.

3.4 Evaluating cells

In order to evaluate cells, press `shift-enter` in an input cell. This will send the input to the kernel and display any results or informal messages below the input cell.

3.5 Algorithm and property auto-completion

The GUI has shell-like automatic completion of algorithm and property names. If you type only part of an algorithm or property, and subsequently press the `TAB` key, the name will be completed to in maximally unambiguous way. For instance, by typing `::Com` and hitting the `TAB` key, the input will be completed to `::Commuting`. By adding `AsP` and hitting TAB again, this gets completed to `::CommutingAsProduct`.

3.6 Cutting and pasting

You can select output cells and paste them back into input cells or into some other application. The program offers two formats for pasting, a LaTeX one which is useful to paste into a paper, and a cadabra one which represents the way in which the expression can be pasted back into the program.

In most cases, the format will be selected correctly automatically. Under Emacs, a middle-mouse paste will paste the LaTeX version. You can paste the internal cadabra format by adding the following to your `.emacs` file:

```
(defun pastecdb ()
   (interactive)
   (insert (x-get-selection-internal 'PRIMARY 'cadabra)))

(global-set-key (kbd "C-x p") 'pastecdb)
```

This will make the combination "Ctrl-x p" insert the current selection in cadabra internal format.

For more advanced users: the list of formats can be inspected using

```
▷ (x-get-selection-internal 'PRIMARY 'TARGETS)
  [TIMESTAMP TARGETS MULTIPLE cadabra UTF8_STRING TEXT]
```

If e.g. an output cell with the content `1/2 A_{m}^{n}` is selected, the two different paste formats can be obtained by evaluating the following two LISP expressions in a scratch buffer,

```
▷ (x-get-selection-internal 'PRIMARY 'cadabra)
  #("1/2 A_{m}^{n};" 0 14 (foreign-selection STRING))
```

```
▷ (x-get-selection-internal 'PRIMARY 'TEXT)
  #("1 := \frac{1}{2}\, A_{m}\,^{n};" 0 31 (foreign-selection COMPOUND_TEXT))
```

Note that the TEXT format contains markup such as spacing commands which make it more useful for papers. This is the default format when the middle mouse button is pressed.

3.7 The TEXmacs frontend

The program also comes with a TEXmacs frontend (which is now no longer actively maintained because the native graphical frontend is in many ways easier to use and more adapted to cadabra). If everything is installed correctly, the "Insert → Session" submenu of TEXmacs should contain an entry for Cadabra. A few sample TEXmacs files can be found in the `texmacs` subdirectory of the source tarball.

Cadabra sends its output more-or-less unchanged to the TEXmacs frontend. It is therefore useful to use tensor names which have a meaning in TEX. For example, if you declare a derivative operator, you might be tempted to use

```
▷ \diff{#}::Derivative.
```

(perhaps because you want to mimic Maple's notation). However, this will lead to ugly output since `\diff` is not a TEX symbol and TEXmacs does not know what to do with it. A better choice would be

```
▷ \nabla{#}::Derivative.
```

since `\nabla` actually prints as a nice symbol in TEXmacs.

If you want to cut-and-paste formulas from a TEX document into a TEXmacs notebook, you will encounter the problem that all backslashes get swallowed. To avoid this, choose the menu entry

Edit → Paste from → Verbatim

which pastes text unmodified. You can make this the default for middle-button paste by choosing

Tools → Selections → Import → Verbatim

which will remain active during the current session.

In order to make PDF output work, you have to have `pfbtops` installed (it is usually part of the `groff` package).

Reference guide

4.1 Basics about the input format

The input format of cadabra is closely related to the notation used by TEX to denote tensorial expressions. That is, one can use not only bracketed notation to denote child objects, like in

```
object[child,child]
```

but also the usual sub- and superscript notation like

```
object^{child child}_{child}
```

One can use backslashes in the names of objects as well, just as in TEX. All of the symbols that one enters this way are considered "passive", that is, they will go into the expression tree just like one has entered them.

"Active nodes", on the other hand, are symbols pre-fixed with a "@" symbol. These are nodes which lead to an algorithm being applied to their children; they will thus not end up in the actual tree stored in memory, but instead get replaced with the output that their algorithm produces (if any). Finally, every non-empty expression that one enters will labelled by a number. Additionally, one can give an expression a label by entering it as "`label-text:= expression;`". One can use either the expression number or the label to refer to an expression.

Names of objects can contain any printable character, except for brackets, sub- and super-script symbols and backslash characters (the latter can only occur at the beginning of a name). The sole exception to this rule is the names of active nodes: these names *can* contain underscores to separate words. The program can also deal with "accents" added to symbol names, like

```
\hat{A}
\bar{B}
\prime{Q}
```

The precise way in which this works is explained in section 4.11.

Input lines always have to be terminated with either a ";", a ":" or a ".". The first two of these delimiting symbols act in the same way as in Maple: the second form suppresses the output of the entered expression (see section 4.14 for additional information on how to redirect output to files). The last delimiter has the effect of evaluating the resulting expression, and immediately discarding it, also disabling printing. Long expressions can, because of these delimiters, be spread over many subsequent input lines. Any line starting with a "#" sign is considered to be a comment (even when it appears within a multi-line expression). Comments are always ignored completely (they do not end up in the expression tree).

All internal storage is in prefix notation, but a converter is applied to the input which transforms normal infix notation for products and sums to prefix notation.

4.2 Active nodes or "commands"

Active nodes are nodes which lead to immediate evaluation after the input has been entered; these could thus be called "commands". Commands always start with a "@" symbol, to distinguish them from normal, unevaluated input. Used in the standard "notebook" way, active nodes act on their first argument and get replaced with the result that they compute. This form uses square brackets to denote the argument on which to act:

```
@command[expression]{arg1}{arg2}...{argn}
```

If you want to re-use an existing expression as the argument, this can be cloned by using the "@" operator itself:

```
@[equation number or label]
```

or (for convenience, as a single exception to the square bracket rule)

```
@(equation number or label)
```

both evaluate to a copy of the expression to which they refer (one can use the "%" character to denote the number of the last-used expression). Note that all of these forms create a *new* expression. One is advised *not* to refer to equation numbers using this command, as this makes expressions tied together in a rather fragile way.

Algorithms can also be made to act on existing expressions such as to modify the expression history (i.e. in the language of the previous section, to stay within a "mini-notebook"). This is done by using round brackets instead of square ones:

```
@command(expression number or label){arg1}{arg2}...{argn}
```

Here it is no longer problematic to refer to equation numbers, as the result of this command will appear in the history list of the indicated expression (and is therefore not tied to the particular number of the expression).

All of these commands act on the top of the argument subtree. You can make them act subsequently on all nodes to which they apply by postfixing the name with an exclamation mark, as in

```
@command![expression]{arg1}{arg2}...{argn}
```

This will search the tree in pre-order style, applying the algorithm on every node to which the algorithm applies.

If you want an algorithm to act until the expression no longer changes, use a double exclamation mark. Compare the single-exclamation mark version,

```
▷ A_{m} B_{m} A_{n} B_{n};
▷ @substitute!(%)( A_{m} B_{n} -> Q);
  Q A_{n} B_{n};
```

with the double-exclamation mark one,

```
▷ A_{m} B_{m} A_{n} B_{n};
▷ @substitute!!(%)( A_{m} B_{n} -> Q );
  Q Q;
```

Be careful with this functionality, as there is currently no safeguard in case the expression keeps changing forever.

Instead of acting only at the top of the argument subtree, or acting on all nodes individually, it is also possible to act only a specific level of the expression. The notation for this is an exclamation mark with an extra number attached to it, which indicates the level (starting at 1 for the highest level),

```
@command!level(expression number of label){arg1}{arg2}...{argn}
```

Note that `\sum` and `\prod` nodes in the expression tree also count as a level.

4.3 Object properties and declaration

Symbols in cadabra have no a-priori "meaning". If you write `\Gamma`, the program will not know that it is supposed to be, for instance, a Clifford algebra generator. You will have to declare the properties of symbols, i.e. you have to tell cadabra explicitly that if you write `\Gamma`, you actually mean a Clifford algebra generator. This indirect way of attaching a meaning to a symbol has the advantage that you can use whatever notation you like; if you prefer to write `\gamma`, or perhaps even `\rho` if your paper uses that, then this is perfectly possible (object properties are a bit like "attributes" in `Mathematica` or "domains" in `Axiom` and `MuPAD`).

Properties are all written using capitals to separate words, as in `AntiSymmetric`. This makes it easier to distinguish them from commands. Properties of objects are declared by using the "::" characters. This can be done "at first use", i.e. by just adding the property to the object when it first appears in the input. As an example, one can write

```
▷ F_{m n p}::AntiSymmetric;
```

This declares the object to be anti-symmetric in its indices and at the same time creates a new expression with this object. The property information is stored separately, so that further appearances of the "`F_{m n p}`" object will automatically share this property. It is of course also possible to keep object property declarations separated (for instance at the top of an input file). This is done as follows:

```
▷ F_{m n p}::AntiSymmetric.
▷ F_{m n p};
```

A list of all properties is available in the graphical interface through the Help menu, and can also be obtained using the `@properties` command.

Note that properties are attached to patterns. Therefore, you can have

```
▷ R::RicciScalar.
▷ R_{m n p q}::RiemannTensor.
```

at the same time. The program will not warn you if you use incompatible properties, so if you make a declaration like above and then later on do

```
▷ R::Coordinate.
```

this may lead to weird results.

The fact that objects are attached to patterns also means that you can use something like wildcards. In the following declaration,

```
▷ { m#, n# }::Indices(vector).
```

the entire infinite set of objects $m1, m2, m3, \ldots$ and $n1, n2, n3, \ldots$ are declared to be in the dummy index set "vector".[4] Range wildcards can also be used to match one or more objects of a specific type. In this case, they always start with a "#" sign, followed by optional additional information to specify the type of the object and the number of them. The declaration

```
{m,n,p,q,r,s}::Indices(vector).
A_{ #{m, 1..3} }::AntiSymmetric.
B_{ #{m} }::Symmetric.
```

indicates that whenever the object `A` appears with one to three indices of the vector type, it is antisymmetric. If the index type does not match, or if there are zero or more than three indices, the property does not apply. Similarly, `B` is always symmetric when all of its indices are of the vector type.

Properties can be assigned to an entire list of symbols with one command, namely by attaching the property to the list. For example,

```
▷ {n, m, p, q}::Integer(1..d).
```

This associates the property "Integer" to each and every symbol in the list. However, there is also a concept of "list properties", which are properties which are associated to the list as a whole. Examples of list properties are "AntiCommuting" or "Indices". See for a discussion of list properties section 4.4.

Objects can have more than one property attached to them, and one should therefore not confuse properties with the "type" of the object. Consider for instance

```
▷ x::Coordinate.
▷ W_{m n p q}::WeylTensor.
▷ W_{m n p q}::Depends(x).
```

[4] This way of declaring ranges of objects is similar to the "autodeclare" declaration method of FORM [6].

This attaches two completely independent properties to the pattern W_{mnpq}.

In the examples above, several properties had arguments (e.g. "`vector`" or "`1..d`"). The general form of these arguments is a set of key-value pairs, as in

```
▷ T_{m n p q}::TableauSymmetry(shape={2,1}, indices={0,2,1}).
```

In the simple cases discussed so far, the key and the equal sign was suppressed. This is allowed because one of the keys plays the role of the default key. Therefore, the following two are equivalent,

```
▷ { m, n }::Integer(range=0..d).
▷ { m, n }::Integer(0..d).
```

See the detailed documentation of the individual properties for allowed keys and the one which is taken as the default.

Finally, there is a concept of "inherited properties". Consider e.g. a sum of spinors, declared as

```
▷ {\psi1, \psi2, \psi3}::Spinor.
▷ \psi1 + \psi2 + \psi3;
```

Here the sum has inherited the property "Spinor", even though it does not have normal or intrinsic property of this type. Properties can also inherit from each other, e.g.

```
▷ \Gamma_{#}::GammaMatrix.
▷ \Gamma_{p o i u y};
▷ @indexsort!(%);
```

The `GammaMatrix` property inherits from `AntiSymmetric` property, and therefore the `\Gamma` object is automatically anti-symmetric in its indices.

A list of all properties known to cadabra can be obtained by using the `@proplist` command.

4.4 List properties and symbol groups

Some properties are not naturally associated to a single symbol or object, but have to do with collections of them. A simple example of such a property is `AntiCommuting`. Although it sometimes makes sense to say that "ψ_m is anticommuting" (meaning that $\psi_m\psi_n = -\psi_n\psi_m$), it happens just as often that you want to say "ψ and χ anticommute" (meaning that $\psi\chi = -\chi\psi$). The latter property is clearly relating two different objects.

Another example is dummy indices. While it may make sense to say that "m is a dummy index", this does not allow the program to substitute m with another index when a clash of dummy index names occurs (e.g. upon substitution of one expression into another). More useful is to say that "m, n, and p are dummy indices of the same type", so that the program can relabel a pair of m's into a pair of p's when necessary.

In cadabra such properties are called "list properties". You can associate a list property to a list of symbols by simply writing, e.g. for the first example above,

```
▷ { \psi, \chi }::AntiCommuting.
```

Note that, as described in section 4.3, you can also attach normal properties to multiple symbols in one go using this notation. The program will figure out automatically whether you want to associate a normal property or a list property to the symbols in the list.

Lists are ordered, although the ordering does not necessarily mean anything for all list properties (it is relevant for e.g. `SortOrder` but irrelevant for e.g. `AntiCommuting`).

4.5 Indices, dummy indices and automatic index renaming

In cadabra, all objects which occur as subscripts or superscripts are considered to be "indices". The names of indices are understood to be irrelevant when they occur in a pair, and automatic relabelling will take place whenever necessary in order to avoid index clashes.

Cadabra knows about the differences between free and dummy indices. It checks the input for consistency and displays a warning when the index structure does not make sense. Thus, the input

```
▷ A_{m n} + B_{m} = 0;
```

results in an error message.

The location of indices is, by default, not considered to be relevant. That is, you can write

```
▷ A_{m} + A^{m};
▷ A_{m} B_{m};
```

as input and these are considered to be consistent expressions. If, however, the position of an index means something (like in general relativity, where index lowering and raising implies contraction with a metric), then you can declare index positions to be "fixed". This is done using

```
▷ {m, n, p}::Indices(position=fixed).
```

When substituting an expression into another one, dummy indices will automatically be relabelled when necessary. To see this in action, consider the following example:

```
▷ {p,q}::Indices(vector).
▷ F_{m n} F^{m n};
  F_{m n} F^{m n}

▷ G_{m n} @(1);
  G_{m n} F_{p q} F^{p q}
```

The m and n indices have automatically been converted to p and q in order to avoid a conflict with the free indices on the G_{mn} object.

Refer to section 5.7 for commands that deal with dummy indices.

argument	whitespace	dummy	example
{}	product		`\frac{a b}{c} = \frac{a * b}{c}`
()	product		`sin(a b) = sin(a * b)`
_{}	separator	yes	`M_{a b} = M_{a}_{b}`
^{}	separator	yes	`M^{a b} = M^{a}^{b}`
_()	separator		`M_(a b) = M_(a)_(b)`
^()	separator		`M^(a b) = M^(a)^(b)`
[]	product		`[a b, c] = [a * b, c]`

Table 1: The implicit meaning of objects and whitespace inside the various types of brackets.

4.6 Exponents, indices and labels

Exponents, indices and labels are traditionally all placed using super- or subscript notation, making these slightly ambiguous for a computer algebra system. Since this situation can become quite complex (how do you represent the third power of the first element of a vector?) cadabra requires all powers to be entered with a double-star "$**$" notation:[5]

```
▷ a**2;
▷ a**(2 + 3b);
```

In addition, all sub- or superscripts with *curly* braces indicate indices, to which the summation convention applies. In particular, there are not allowed to be more than two identical indices in a single product. The summation convention does not apply to arguments with any other bracket types. In particular, sub- or superscripts with *square* or *pointy* brackets have (as of yet) no fixed meaning.

4.7 Spacing and brackets

Cadabra is reasonably flexible as far as spacing and brackets are concerned, but the fact that objects do not have to be declared before they can be used means that spaces have to be introduced in some cases to avoid ambiguity. As a general rule, all terms in a product have to be separated by at least one whitespace character. Thus,

`A(B+C)` incorrect, (interpreted as `A` with argument `B+C`),

`AB` incorrect, (interpreted as one object, not two)

`A (B+C)` correct,

`A*(B+C)` correct.

If a whitespace character is absent, all brackets are interpreted as enclosing *argument* groups. Products of variables (e.g. AB) have to be separated by a space, otherwise the input will be read as the name of a single variable. However, spaces will automatically be

[5] Solutions involving the declaration of which indices are exponents and which are indices have been considered, but rejected in favour of the explicit "$**$" notation. Cases like "the square of the second component of a vector" quickly become ambiguous even with rather complicated property declarations, while the "$**$" notation remains transparent.

inserted after numbers, between a closing bracket and a name or number, and between two names if the second name starts with a backslash. The following expressions are therefore interpreted as one would expect:

`3A`	interpreted as `3*A`
`(A+B)C`	interpreted as `(A+B)*C`
`(A+B)3`	interpreted as `(A+B)*3`
`A\Gamma`	interpreted as `A*`Γ

This conversion is done by the preprocessor. Finally, note that brackets in the input *must* be balanced (a decision made to simplify the parser; it means that cadabra uses a different way to indicate groups of symmetric or anti-symmetric indices than one often encounters in the literature; see section 5.9).

4.8 Implicit versus explicit indices

When writing expressions which involves vectors, spinors and matrices, one often employs an implicit notation in which some or all of the indices are suppressed. Examples are

$$a = Mb\,, \qquad \bar{\psi}\gamma^m\chi\,, \tag{4.1}$$

where a and b are vectors, ψ and χ are spinors and M and γ^m are matrices. Clearly, the computer cannot know this without some further information. In cadabra objects can carry implicit indices, through the `ImplicitIndex` property. There are derived forms of this, e.g. `Matrix` and `Spinor`,

```
▷ {a,b}::ImplicitIndex;
▷ M::Matrix.
▷ a = M b;
▷ @prodsort!(%);
  @prodsort: not applicable.
```

If you had not made the property assignment in the first two lines, the `@prodsort` would have incorrectly swapped the matrix and vector, leading to a meaningless expression.

If you have more than one set of implicit indices, you can label them just like you can label explicit indices,

```
▷ {a,b}::ImplicitIndex(type1);
▷ {c,d}::ImplicitIndex(type2);
▷ M::Matrix(type1).
▷ a = M d c b;
▷ @prodsort!(%);
  a = M b d c;
```

4.9 Index brackets

Indices can be associated to tensors, as in $T_{\mu\nu}$, but it often also happens that we want to associate indices to a sum or product of tensors, without writing all indices out explicitly.

Examples are

$$(A + B + C)_{\alpha\beta}\,, \quad \text{or} \quad (\psi\Gamma_{mn}\Gamma_p)_\beta\,. \tag{4.2}$$

Here the objects A, B, C and Γ are matrices, while ψ is a vector. Their explicit components are labelled with α and β indices, but the notation above keeps most of these vector indices implicit.

Cadabra can deal with such expressions through a construction which is called the "indexbracket". It is possible to convert from one form to the other by using the `@combine` and `@expand` algorithms. Combining terms goes like this,

```
▷ (\Gamma_r)_{\alpha\beta} (\Gamma_{s t u})_{\beta\gamma}:
▷ @combine!(%);
  (\Gamma_r \Gamma_{s t u})_{\alpha\gamma};
```

or as in

```
▷ (\Gamma_r)_{\alpha\beta} Q_\beta:
▷ @combine!(%);
  (\Gamma_r Q)_{\alpha};
```

If the index bracket has only one index, either the first or the last argument should be a matrix, but not both:

```
▷ A::Matrix.
▷ {m,n,p}::Indices(vector).
  (A B)_{m};
▷ @expand(%);
  A_{m n} B_{n};
```

If the index bracket has two indices, all arguments should be matrices,

```
▷ {A,B}::Matrix.
▷ {m,n,p}::Indices(vector).
▷ (A B)_{m n};
  @expand(%);
  A_{m p} B_{p n};
```

If there are more arguments inside the bracket, these of course all need to be matrices (and are assumed to be so by default).

4.10 Derivatives and implicit dependence on coordinates

There is no fixed notation for derivatives; as with all other objects you have to declare derivatives by associating a property to them, in this case the `Derivative` property.

```
▷ \nabla{#}::Derivative.
```

Derivative objects can be used in various ways. You can just write the derivative symbol, as in

```
▷ \nabla{ A_{\mu} };
```

(a notation used e.g. in the tutorial example in section 2.4). Or you can write the coordinate with respect to which the derivative is taken,

```
▷ s::Coordinate.
▷ A_{\mu}::Depends(s).
▷ \nabla_{s}{ A_{\mu} };
```

Finally, you can use an index as the subscript argument, as in

```
▷ { \mu, \nu }::Indices(vector).
▷ \nabla_{\nu}{ A_{\mu} };
```

(in which case the first line is, for the purpose of using the derivative operator, actually unnecessary).

The main point of associating the `Derivative` property to an object is to make the object obey the Leibnitz or product rule, as illustrated by the following example,

```
▷ \nabla{#}::Derivative.
▷ \nabla{ A_{\mu} * B_{\nu} };
▷ @prodrule!(%);
  \nabla{A_{\mu}} B_{\nu} + A_{\mu} \nabla{B_{\nu}};
```

This behaviour is a consequence of the fact that `Derivative` derives from `Distributable`. Note that the `Derivative` property does not automatically give you commuting derivatives, so that you can e.g. use it to write covariant derivatives.

More specific derivative types exist too. An example are partial derivatives, declared using the `PartialDerivative` property. Partial derivatives are commuting and therefore automatically symmetric in their indices,

```
▷ \partial{#}::PartialDerivative.
▷ {a,b,m,n}::Indices(vector).
▷ C_{m n}::Symmetric.

▷ T^{b a} \partial_{a b}( C_{m n} D_{n m} );
▷ @canonicalise!(%);
  T^{a b} \partial_{a b}( C_{m n} D_{m n} );
```

4.11 Accents

It often occurs that you want to put a hat or tilde or some other accent on top of a symbol, as a means to indicate a slightly different object. In most of these cases, the properties of the normal symbol and the accented symbol are identical. Such accents are declared using the `Accent` property, as in

```
\hat{#}::Accent.
```

This automatically makes all symbols with hats inherit the properties of the unhatted symbols,

```
▷ \hat{#}::Accent.
▷ {\psi, \chi}::AntiCommuting.
▷ {\psi, \chi}::SortOrder.
▷ \hat{\chi} \psi:
▷ @prodsort!(%);
  (-1) \psi \hat{\chi};
```

If you want to put an accent on an object with indices, wrap the accent around the entire object, do not leave the indices outside.

Note that it is also possible to mark objects by attaching sub- or superscripted symbols to them, as in e.g. $A^\dagger$. This can be done by declaring these symbols explicitly using the `Symbol` property,

```
▷ \dagger::Symbol.
```

If you do not do this, `\dagger` will be seen as an index and an error will be reported if it appears more than twice.

4.12 Symbol ordering and commutation properties

A conventional way to sort factors in a product is to use lexographical ordering. However, this is almost never what one wants when transcribing a physics problem to the computer. Therefore, cadabra allows you to specify the sort order of symbols yourself. This is done by associating the `SortOrder` list property to a list of symbols, as in

```
▷ {X,G,Y,A,B}::SortOrder.
▷ A*B*G*X*A*X:
▷ @prodsort(%);
  X*X*G*A*A*B;
```

More complicated objects with indices are of course also allowed, such as in

```
▷ { W_{m n}, W_{m} }::SortOrder.
▷ W_{m n} W_{p} W_{q r} W_{s} W_{t}:
▷ @prodsort(%);
  W_{m n} * W_{q r} * W_{p} * W_{s} * W_{t};
```

For the time being, it is not allowed to have more than one such set contain the same symbol. Thus,

```
▷ {X,G}::SortOrder.
▷ {X,A,B}::SortOrder.
```

is not allowed (and will, in fact, take X out of the first list).

Apart from the preferred sort order, there are more properties which influence the way in which products can be sorted. In particular, sort order is influenced by whether symbols commute or anti-commute with each other. Physicists in general use a very implicit notation as far as commutativity of objects in a product is concerned. Consider for instance a situation in which we deal with two operators $\hat{M}$ and $\hat{N}$, as well as some constants q and p. These two expressions are equivalent:

$$2q\,\hat{M}p\hat{N} \quad \text{and} \quad 2pq\,\hat{M}\hat{N}\,. \tag{4.3}$$

But this is not obvious from the notation that has been used to indicate the product. In fact, the product symbol is usually left out completely.

In many other computer algebra systems, you have to introduce special types of "non-commuting" products (e.g. the `&*` operator in Maple or the `**` operator in Mathematica). This can be rather cumbersome, for a variety of reasons. The main reason, however, is that it does not match with what you do on paper. On paper, you never write special product symbols for objects which do not commute. You just string them together, and know from the properties of the symbols whether objects can be moved through each other or not.

In order to make these sort of things possible in cadabra, it is necessary to declare "sets" of objects which mutually do not commute (i.e. for which the order inside a product cannot be changed without consequences) but which commute with objects of other sets. Many computer algebra systems only use one such set: objects are either "commuting" or "non-commuting". This is often too restrictive. For instance, when Ψ is a fermion and Γ denotes a Clifford algebra element, a system with only one set of non-commuting objects is unable to see that

$$\bar{\Psi}_a(\Gamma_n\Gamma_m)_{ab}\Psi_b \quad \text{and} \quad \bar{\Psi}_a\Psi_b(\Gamma_n\Gamma_m)_{ab} \tag{4.4}$$

are equivalent. In cadabra, one would simply put Ψ_a and Γ_m in two different sets, mutually commuting, but non-commuting among themselves. To be precise, the example above is reproduced by

```
▷ \bar{#}::Accent.
▷ \Psi_{a}::SelfNonCommuting.
▷ \Gamma_{#}::GammaMatrix.
▷ \bar{\Psi_a} (\Gamma_n \Gamma_m)_{ab} \Psi_b:
▷ @prodsort(%);
  \bar{\Psi_a} \Psi_b (\Gamma_n \Gamma_m)_{ab};
```

(see section 4.11 for an explanation of the `Accent` property).

Commutation properties always refer to components. If you associate an `ImplicitIndex` property to an object, then it will never commute with itself or with any other such object. You can see this in the example above, as the `@prodsort` command kept the order of the two gamma matrices unchanged.

4.13 Anti-commuting objects and derivatives

We have discussed anti-commuting objects to some extend in the previous section, but things become somewhat more hairy in combination with derivatives. Hence the current section, in which we will look in more detail at how to write properties which take care of anti-commuting derivatives (as they occur in e.g. superspace field theories).

We will use the `@prodrule` command to show how the various property assignments work.

There are various ways in which you may want to write derivatives which are anti-commuting. One way is to use explicit indices, and declare them to be anti-commuting, as in

```
▷ {\alpha,\beta,\gamma}::Indices(spinor).
▷ {\alpha,\beta,\gamma}::AntiCommuting.
▷ D{#}::Derivative.
▷ D_{\alpha}{\theta^\beta \theta^\gamma};
  D_{\alpha}{\theta^{\beta}} \theta^{\gamma}
             - \theta^{\beta} D_{\alpha}{\theta^{\gamma}};
```

Note the appearance of the minus sign for the second term. Writing things in this way is the most straightforward way to handle anti-commuting derivatives, and only requires the assignment of the `AntiCommuting` property to the indices (second line above).

However, one often deals with implicit indices when tackling problems involving anti-commuting variables. Here is an example to illustrate what is meant by this:

```
▷ test 19d does not do the right thing: it does not pick up a minus sign
▷ at the first step.
```

4.14 Input and output redirection

By default, all generated output is displayed on the screen. As mentioned in section 4.1, the delimiter of a line determines whether output is suppressed or not. It is, however, also possible to write output to a file, by appending a file name to the line delimiter, as in

```
▷ a^2+b^2 = c^2; "output_file"
```

(note that there is no semi-colon following the file name). This shows the filename "output_file" on the display, and writes to this file the expression

```
1:=  a^{2}+b^{2}=c^{2}
```

This is exactly as it would have appeared on the display. If you need a different kind of output, use the algorithms provided in the `output` module (see section 5.10).

It is also possible to read input from files instead of from the terminal:

```
▷ < "input_file"
```

reads from the file "input_file" until the end or until a line `@end;` is found, whichever comes first. Input redirect can be nested. In case an error occurs, reading from the file is also aborted and input is taken from the terminal again.

4.15 Default rules

By default, cadabra does very few things "by itself" with your expressions. This goes as far as not collecting equal terms, so that you can write

```
▷ A + A;
```

and have this stored as a sum of two terms. However, you can add an arbitrary number of default commands, which are run just after you have entered an expression on the command line (`PreDefaultRules`) or just before the output is returned to you (`PostDefaultRules`). So if you want terms to be collected automatically, just enter

```
▷ ::PostDefaultRules( @@collect_terms!(%) ).
```

Note the double "@@" symbol, which prevents immediate execution of the command. This post default rule will collect equal terms before they ever make it to internal storage.

Many more complicated things can be achieved with this feature. You can, for instance, also use complicated substitution commands as arguments to `PreDefaultRules` or `PostDefaultRules`. An example is

```
▷ {m,n,p,q}::Indices(vector).
▷ ::PreDefaultRules( @@substitute!(%)( A_{m n} -> B_{m q} B_{q n} ).

▷ A_{m n} A_{n m};
```

which immediately returns

```
B_{m q} B_{q n} B_{n p} B_{p m};
```

As usual dummy indices have been relabelled appropriately.

4.16 Patterns, conditionals and regular expressions

Patterns in cadabra are quite a bit different from those in other computer algebra systems, because they are more tuned towards the pattern matching of objects common in tensorial expressions, rather than generic tree structures.

Name patterns are constructed by writing a single question mark behind the name, as in

```
▷ @substitute!(%)( A? + B? -> 0 );
```

which matches all sums with two terms, each of which is a single symbol without indices or arguments. If you want to match instead any object, with or without indices or arguments, use the double question mark instead. To see the difference more explicitly, compare the two substitute commands in the following example:

```
▷ A_{m n} + B_{m n}:
▷ @substitute!(%)( A? + B? -> 0 );
  A_{m n} + B_{m n};
▷ @substitute!(%)( A?? + B?? -> 0 );
  0;
```

Note that it does not make sense to add arguments or indices to object patterns; a construction of the type "`A??_{\mu}(x)`" is meaningless and will be flagged as an error.

There is a special handling of objects which are dummy objects in the classification of section 4.5 (see table 1). Objects of this type do not need the question mark, as their explicit name is never relevant. You can therefore write

```
▷ @substitute!(%)( A_{m n} -> 0 );
```

to set all occurrances of the tensor A with two subscript indices to zero, regardless of the names of the indices (i.e. this command sets A_{pq} to zero).

When replacing object wildcards with something else that involves these objects, use the question mark notation also on the right-hand side of the rule. For instance,

```
▷ @substitute!(%)( A? + B? -> A? A? );
```

replaces all sums with two elements by the square of the first element. The following example shows the difference between the presence or absence of question marks on the right-hand side:

```
▷ C + D:
▷ @substitute!(%)( A? + B? -> A? A? );
  C * C;
▷ @pop(%);
▷ @substitute!(%)( A? + B? -> A A );
  A * A;
```

Note that you can also use patterns to remove or add indices or arguments, as in

```
▷ {\mu,\nu,\rho}::Indices(vector).
▷ A_{\mu} B_{\nu} C_{\nu} D_{\mu};
▷ @substitute!!(%)( A?_{\rho} B?_{\rho} -> \dot(A?)(B?) );
  A.D B.C;
▷ @substitute!!(%)( \dot(A?)(B?) -> A?_{\mu} B?_{\mu} );
```

In many algorithms, patterns can be supplemented by so-called conditionals. These are constraints on the objects that appear in the pattern. For instance, the substitution command

```
▷ @substitute!(%)( A_{m n} B_{p q} | _{n}!=_{p} -> 0 );
```

applies the substitution rule only if the second index of A and the first index of B do not match. Note that the conditional follows directly after the pattern, not after the full substitution rule. A way to think about this is that the conditional is part of the pattern, not of the rule. The reason why the conditional follows the full pattern, and not directly the symbol to which it relates, is clear from the example above: the conditional is a "global" constraint on the pattern, not a local one on a single index.

These conditions can be used to match names of objects using regular expressions. Consider the following example:

```
▷ A + B3 + C7:
▷ @substitute!(%)( A + M? + N?  | { \regex{M?}{"B[0-9]*"},
                                  \regex{N?}{"[A-Z]7"} } -> sin(M? N?)/N? );
  sin(B3 C7)/C7;
```

It is also possible to select objects based on whether or not they are associated with a particular property, e.g.[6]

```
▷ A::SelfAntiCommuting.
▷ A A + B B;
▷ @substitute!(%)( Q?? Q?? | \hasprop{Q??}{SelfAntiCommuting} -> 0 );
  B B;
```

Combinations of different types of conditionals are of course also possible. This type of constraint can be used to construct replacement rules which act on objects of a certain type without knowing what their name is. The following example illustrates this.

```
▷ \sigma_{a?}::SigmaMatrix.
▷ \sigma_{1}\sigma_{2} + \rho_{1} \rho_{2};
▷ @substitute!(%)( S?_{1} S?_{2} | \hasprop{S?_{1}}{SigmaMatrix} -> I*S?_{3}  );
  I \sigma_{3} + \rho_{1} \rho_{2};
```

This could also have been written without specifying the numerical values of the indices explicitly,

```
▷ \sigma_{a?}::SigmaMatrix.
▷ \sigma_{1}\sigma_{2} + \rho_{1} \rho_{2};
▷ @substitute!(%)( S?_{a?} S?_{b?} | \hasprop{S?_{a?}}{SigmaMatrix} -> I*\epsilon_{a? b? c} \sig
  I \epsilon_{1 2 c} \sigma_{c} + \rho_{1} \rho_{2};
```

[6] More compact notations may look tempting, but were abandoned in favour of the constraint-based notation, the reason being that more compact notations tend to become hard to read when the constraint is more complicated.

As a general rule, patterns on the right-hand side of a substitution rule get replaced even if their parent relation does not match the one on the left-hand side,

```
▷ q_{c d};
▷ @substitute!(%)( q_{a b} -> a Q_{a b} );
▷ c Q_{c d};
```

This follows the general logic of explicit wildcards,

```
▷ q_{c d};
▷ @substitute!(%)( q_{a? b?} -> a? Q_{a? b?} );
▷ c Q_{c d};
```

If this is not what you want, just rename the dummy index labels,

```
▷ q_{c d};
▷ @substitute!(%)( q_{e f} -> a Q_{e f} );
▷ a Q_{c d};
```

4.17 Procedures

Although the present version of cadabra does not contain a full-fledged programming language, it does allow you to construct groups of commands which can be applied term-by-term on a large sum. This is sometimes useful if e.g. the application of a command on a single term already leads to a large intermediate result.

Procedures are defined by making use of the `@procedure` construction. Here is an example, which distributes all products and performs a substitution:

```
▷ @procedure{ExpandAndCollect};
▷ @distribute!(%);
▷ @substitute!(%)( A C -> Q );
▷ @collect_terms!(%);
▷ @procedure_end;
```

A sample session which calls this procedure is

```
▷ (A+C)*(B+C+A) + C*(E+A);
▷ @call{ExpandAndCollect};
```

This produces some status output, followed by

```
A B + 2 Q + A A + C B + C C + C E + Q;
```

Note how the commands inside the procedure only act on each term in turn. The `@collect_terms` command has only collected two `Q` symbols which arose from the first term. Note also that all output of the commands inside the procedure has been suppressed, despite the appearance of the semi-colon line delimiter.

4.18 Reserved node names

There is a small number of node names which is reserved to always mean the same thing, i.e. for which the properties are not determined by the property system. These can be obtained using the '@reserved' command. Explicitly, they are

\anticommutator, \arrow, \comma, \commutator, \conditional, \frac, \cdot, \equals, \expression, \factorial, \indexbracket, \infty, \label, \pow, \prod, \regex, \sequence, \sum, \unequals.

Moreover, cadabra uses a number of standard names for mathematical operators as reserved keywords, to wit

\sin, \cos, \tan, \sinh, \cosh, \tanh, \arcsin, \arccos, \arctan, \sqrt

Some properties of these nodes are still obtained from the property system in order to make algorithms look more uniform, but the names of these reserved nodes can not be changed.

\anticommutator
Denotes the anti-commutator of two objects.

The default properties of \anticommutator are

```
\anticommutator{#}::IndexInherit.
\anticommutator{#}::Derivative.
```

\arrow
Denotes a substitution rule.

\comma
Denotes comma-separated objects. The parser rewrites any set of objects which are separated by an infix comma using the \comma object.

\commutator
Denotes the commutator of two objects.

The default properties of \commutator are

```
\commutator{#}::IndexInherit.
\commutator{#}::Derivative.
```

\conditional
Denotes a conditional pattern. The parser rewrites an infix vertical bar operator using a \conditional object, in which the first argument is the pattern and the second argument is the condition under which the pattern holds.

\frac
Denotes a generic ratio of objects.

\cdot
Denotes a dot product of vectors in which the contracted indices are suppressed. Displays as an infix dot.

`\equals`
Denotes equalities. The parser rewrites an infix = operator using an `\equals` object.

`\expression`
Denotes the top-level node of an expression.

`\factorial`
Denotes the factorial of an object. The parser rewrites a post-fix exclamation mark as a `\factorial` object.

`\indexbracket`
Denotes a group of objects with indices which are collectively written, as in $(A + B + C)_{mn}$.

The default properties of `\indexbracket` are

```
\indexbracket{#}::Distributable.
\indexbracket{#}::IndexInherit.
```

`\infty`
Denotes infinity.

`\label`
Denotes the label of an expression.

`\pow`
Denotes a generic exponent of one object raised to the power of another object. The parser translates the infix "`**`" to `\pow`.

The default properties of `\prod` are

```
\pow{#}::DependsInherit.
```

`\prod`
Denotes a generic product of objects. The parser translates objects separated by whitespace or by a "`*`" symbol to products.

The default properties of `\prod` are

```
\prod{#}::Distributable.
\prod{#}::IndexInherit.
\prod{#}::CommutingAsProduct.
\prod{#}::DependsInherit.
\prod{#}::WeightInherit(label=all, type=Multiplicative).
\prod{#}::NumericalFlat.
```

`\regex`
Denotes a regular expression name pattern. The first argument is a symbol and the second argument is a string containing the regular expression.

`\sequence`
Denotes a sequence or range of objects. The parser rewrites an infix `..` operator (two dots) using a `\sequence` object.

`\sum`
Denotes a generic sum of objects. The parser translates objects separated by a "+" symbol tol sums.

The default properties of `\sum` are

```
\sum{#}::CommutingAsSum.
\sum{#}::DependsInherit.
\sum{#}::IndexInherit.
\sum{#}::WeightInherit(label=all, type=Additive).
```

`\unequals`
Denotes inequalities. The parser rewrites an infix `!=` operator using an `\unequals` object.

4.19 Startup and program options

There are a few startup options for the command line version of cadabra. If present, a startup file `~/.cadabra` will be read when the program is started (only when running cadabra in command-line mode). This file can contain any cadabra input.

`--bare`
Disable reading of the startup file `~/.cadabra`.

`--silent`
Disable normal output (so that the only remaining output is through output redirection into files).

`--input [filename]`
Before switching to interactive mode, first read and process the indicated file.

`--prompt [string]`
Set the cadabra prompt to the given string.

`--silentfail`
Silently fail (i.e. do not complain) if an algorithm is activated which cannot be applied.

`--loginput`
Write the input into the debug file as well.

`--nowarnings`
Suppress displaying of any warnings which are not fatal errors.

`--texmacs`
Send output in TeXmacs format.

In order to have command-line editing functionality, the `prompt` program is provided. So the two ways of starting cadabra are

```
cadabra
prompt cadabra
```

the second of which gives cursor control and an editing history. There are various other settings of cadabra which can be influenced through environment variables; see section 4.20 for details.

4.20 Environment variables

There are a few environment variables which may be set to influence the behaviour of cadabra:

1. **CDB_LOG** When set, debug logging will be enabled.
2. **CDB_ERRORS_ARE_FATAL** If this variable is set, errors (inconsistency of the tree or attempted use of non-existing algorithm) will abort the program (the default action, when this variable is unset, is to ignore the offending input line).
3. **CDB_PARANOID** Perform extensive consistency checks on the internal data format after each algorithm has been applied (only useful for debugging).
4. **CDB_PRINTSTAR** Determines whether or not multiplication star symbols are displayed.
5. **CDB_TIGHTSTAR** Make output of multiplication symbols more compact.
6. **CDB_TIGHTBRACKETS** Make output of bracketed expressions more compact.
7. **CDB_USE_UTF8** Use UTF8 line breaking characters in the output.

Algorithm modules

All algorithms are stored in modules separate from the core of the program. These are at present all distributed together with the main program, but will in the future be available separately as binary or source plugin modules that can be loaded on demand.

Any expression that starts with a name beginning with an "@" symbol is interpreted as a command. Commands act on the currently selected objects in an expression or on the expression for which the label or number is given as the first argument. See section 4.2 for more details on how to apply algorithms to expressions.

5.1 Built-in core algorithms

There is a very small number of properties and algorithms built into the core program. These have to do with global memory management and global output settings.

properties:

`::KeepHistory`
This is a global property. When set to false, only the last version of each expression is kept in memory, thereby reducing memory usage for very large expressions. The default is "true". When set to "false", `@pop` is no longer available to return to previous forms of the expression:

```
▷ ::KeepHistory(false).
▷ (a+b)*(c+d);
▷ @distribute!(%);
▷ @pop(%);
  @pop: not applicable.
```

In general this is only useful to conserve memory when extremely long expressions are manipulated.

`::PreDefaultRules`
Set the default rules to be executed on all input *before* the active nodes in that input are expanded.

`::PostDefaultRules`
Set the default rules, to be applied *after* every new input has been processed and active nodes have been executed. Use the inert form of active nodes for those ones that only have to be come active upon actual evaluation of the rule, i.e. `@@collect_terms!(%)`. Here is an example containing more than one rule:

```
▷ ::PostDefaultRules( @@distribute!(%), @@prodsort!(%), @@collect_terms!(%) ).
▷ A*(B+C) + B*(A+C);
  2 A B + A C + B C;
```

Note that the rules have to be given in order; the list will only be traversed once.

algorithms:

`@pop`
Remove the last step of the indicated expression history.

```
▷ A (B + C);
▷ @distribute!(%);
  A B + A C;
▷ @pop(%);
  A (B + C);
```

This only works if history keeping was not turned off with the `KeepHistory` property.

`@amnesia`
Forget the history of the indicated expression, that is, remove all previous forms of the expression from memory.

```
▷ A (B + C);
▷ @distribute!(%);
  A B + A C;
▷ @amnesia(%);
▷ @pop(%);
  @pop: not applicable.
```

This is mainly useful to purge extremely long intermediate expressions from memory, while still retaining the option to use the expression history for other expressions (instead of switching expression histories off altogether with `KeepHistory`).

`@algorithms`
Display a list of all available algorithms.

```
▷ @algorithms;
  @
  @acanonicalorder
  @all_contractions
  ...
```

This list is also available in the graphical notebook interface from the Help menu.

`@properties`
Display a list of all available properties.

```
▷ @properties;
  ::Accent
  ::AntiCommuting
  ::AntiSelfDual
  ...
```

This list is also available in the graphical notebook interface from the Help menu.

`@timing`
Display a list of all algorithms together with the total time spent in them since program start or since a `@reset`.

```
▷ @timing;
                              @           0  0 sec and 0 microsec
                 @acanonicalorder         0  0 sec and 0 microsec
               @all_contractions          0  0 sec and 0 microsec
  ...
```

`@output_format`
Set the output format according to the argument given. It can take one of the three values This can take one of the values `cadabra`, `mathematica`, `reduce`, `maple`, `texmacs`, `xcadabra`, `mathml`.

```
▷ @output_format{mathematica};
  A_{m n} B_{p q};
  Tensor[A, {m, n}] * Tensor[B, {p, q}];
  @output_format{reduce};
  A(m,n) * B(p,q);
```

Not all of these output formats are fully functional and some are meant only as an internal format (e.g. `xcadabra` is used for communication with the graphical front-end).

`@quit`
Exit the program. Any open output file will be flushed and closed.

`@end`
This is a marker which can be used in an input file to make the program stop reading (working just like "`\end{document}`" does for LaTeX or "`\end` for TeX).

`@reset`
Erases all expressions and object properties, effectively resetting the program without restarting it.

`@xterm_title`
When the program is running in command-line mode, this command set the title of the xterm to the given argument:

```
▷ @xterm_title{"second part"};
```

This can be useful to monitor the stages of a long calculation: by putting this command at intermediate stages it becomes easier to spot the current status of a running job.

5.2 Rationals, complex numbers and other numerics

properties:

`::Integer(`range`)`
Indicates that the object takes values in the integers. An optional range can be specified,

```
▷ p::Integer;
▷ m::Integer(1..10);
▷ n::Integer(1..d-p);
```

This property is often used in combination with the `Indices` property to indicate the range of numbers over which indices run.

`::NumericalFlat`
Indicates that an operator is such that numerical factors can be taken out of its argument.

```
▷ Q{#}::NumericalFlat.
▷ Q( 3 A ) + R( 3 A );
  @numerical_flatten!(%);
  3 Q( A ) + R( 3 A );
```

algorithms:

`@numerical_flatten`
Move numerical factors out of operators whenever possible. This is mostly useful in combination with derivative operators, as in the example below.

```
▷ \partial{#}::PartialDerivative.
▷ 3 A \partial{4 B};
▷ @numerical_flatten!(%);
  12 A \partial{B};
```

In contrast, the `@unwrap` algorithm moves non-numerical factors out of derivative operators.

Objects to which this algorithm applies should carry the `NumericalFlat` property (which the `Derivative` objects inherit automatically).

5.3 Sums and products

This module handles expansion of sums and products as well as distributing products over sums and similar algorithms. It recognises

properties:

`::Matrix`
Declares an object to have two implicit indices, thus making it a matrix object. This is useful in combination with the index bracket notation. Example:

```
▷ {m,n,p,q}::Indices.
▷ A::Matrix.
  V::ImplicitIndex.
  (A V)_{m};
  @expand!(%);
  A_{m n} V_{n};
```

Objects with the `Matrix` property automatically become non-commuting among themselves, but retain their commutativity properties with other objects.

```
▷ {A,C}::Matrix.
▷ A C B A;
  @prodsort!(%);
  A B C A;
```

`::ImplicitIndex`
Indicates that the object carries implicit indices, e.g. for objects representing matrices or vectors. Such objects will not be moved through each other, i.e. they are mutually noncommuting.

```
▷ {M,A}::ImplicitIndex;
▷ M A;
▷ @prodsort!(%);
  M A;
```

Without the `ImplicitIndex` property, `@prodsort` would have freely moved the objects through each other.

For more information on how to use matrices and vectors with implicit indices, see `@expand`.

`::Diagonal`
Indicates that the object to which the property is attached only has non-zero components on the diagonal, i.e. when all indices take the same value.

```
▷ \delta_{m n}::Diagonal.
▷ \delta_{1 2} \delta_{1 2} - \delta_{1 1} \delta_{2 2};
  @canonicalise!(%);
  - \delta_{1 1}\delta_{2 2};
```

`::SelfAntiCommuting`
Used to make objects with indices anti-commuting when their index values are different. Example:

```
▷ \psi^{\mu}::SelfAntiCommuting.
▷ \psi^{\nu} \psi^{\mu}:
▷ @prodsort!(%);
  (-1) \psi^{\mu} \psi^{\nu};
▷ \psi^{\mu} \psi^{\mu}:
▷ @canonicalise!(%);
  0;
```

This could not be handled with `AntiCommuting` because that property handles the behaviour of *different* expression patterns.

`::SelfNonCommuting`
Used to make objects with indices non-commuting when their index values are different. See `SelfAntiCommuting` for more information.

`::SelfCommuting`
Used to make objects with indices commuting when their index values are different. This is the default, so usually not needed. See `SelfAntiCommuting` for more information.

`::Distributable`
Makes an object distributable. When the object has a sum as argument, it can be distributed over the terms with `@distribute`, as in

```
▷ \hat{#}::Distributable.
▷ \hat{A+B+C D}:
▷ @distribute!(%);
  \hat{A} + \hat{B} + \hat{C D};
```

`::AntiSymmetric`
Makes an object antisymmetric in all its indices. Example:

```
▷ A_{m n}::AntiSymmetric.
▷ B_{m n}::Symmetric.
▷ A_{m n} B_{m n};
```

```
▷ @canonicalise!(%);
  0;
```

For more complicated symmetries, use `TableauSymmetry`.

`::Symmetric`
Makes an object symmetric in all its indices. For an example and more information, see `AntiSymmetric`.

`::TableauSymmetry(`*shape*=tableau shape*, indices*=index positions*, selfdual, antiselfdual)*
Takes lists of two key-value pairs as arguments, indicating the shape of the Young tableau and the index slots associated to each box in the tableau. For instance

```
▷ R_{a b c d}::TableauSymmetry( shape={2,2}, indices={0,2,1,3} ).
```

yields the symmetries of the Riemann tensor. Note that indices are counted from zero. It is also possible to label tensors as self-dual or anti-selfdual using the optional arguments.

`::DAntiSymmetric`
Declares an object to have the symmetries of the derivative of a fully anti-symmetric tensor, i.e. to have a `TableauSymmetry` corresponding to a hook-shape Young tableau.

```
▷ DF_{m n p q}::DAntiSymmetric.
▷ A_{m n p q}::AntiSymmetric.
  DF_{m n p q} A^{m n p q};
  @impose_bianchi!(%);
  0;
```

The symmetries can also be declared directly using `TableauSymmetry` but it is usually easier to read the `DAntiSymmetric` form.

`::Commuting`
Makes components commuting. This is completely analogous to `AntiCommuting` and in fact the default for all objects, so usually does not need to be specified.

`::NonCommuting`
Makes components non-commuting. See `AntiCommuting` for more information on commutativity properties.

`::AntiCommuting`
Makes components anti-commuting. Example:

```
▷ {A,B}::AntiCommuting.
▷ B A;
  @prodsort!(%);
  (-1) A B;
```

It also works for objects with indices:

```
▷ {\psi_{m}, \chi}::AntiCommuting.
▷ \psi_{m} \chi \psi_{n};
  @prodsort!(%);
  (-1) \chi \psi_{m} \psi_{n};
```

If you want a pattern like `\psi_{m}` to anti-commute with itself, you should use the `SelfAntiCommuting` property instead.

You can think about the difference between `SelfAntiCommuting` and `AntiCommuting` in the following way. If `A_{m n}` is `SelfAntiCommuting`, it means that for each value of the indices the expression `A_{m n}` is an operator which anti-commutes with the operator for any other value of the indices. The matrix A is thus a matrix of operator-valued components which mutually anti-commute. On the other hand, if `A` and `B` are declared to be `AntiCommuting`, then these can be viewed as two matrices of commuting components, whose matrix product satisfies $AB = -BA$.

If you attach the `AntiCommuting` property to an object with an `ImplicitIndex` property, the commutation property does not refer to the object as a whole, but rather to its components. The logic behind that becomes clear when considering e.g. spinor bilinears

```
▷ {\chi, \psi}::Spinor(dimension=10, type=MajoranaWeyl).
▷ {\chi, \psi}::AntiCommuting.
▷ \bar{#}::DiracBar.
▷ \Gamma{#}::GammaMatrix.
▷ {\chi, \psi}::SortOrder.
▷ \bar{\psi} \Gamma_{m n p} \chi;
▷ @prodsort!(%);
  @prodsort: not applicable.
▷ @spinorsort!(%);
  \bar{\chi} \Gamma_{m n p} \psi;
```

Here `@prodsort` did not act because both the spinors and the gamma matrices have the `ImplicitIndex` property and there are thus no simple rules for their reordering. However, the `@spinorsort` algorithm did act, and took into account the fact that the components of the spinors are anti-commuting.

`::CommutingAsProduct`
This makes an object behave, for purposes of commutation rules, as a product. That is, when trying to move another object through it, the sign that is picked up is determined by moving the other object through all the non-index children.

In the example below,

```
▷ A(#)::CommutingAsProduct.
▷ {Q,X,Y,Z}::AntiCommuting.
▷ A(Y)(Z)(Q)*A(X);
  @prodsort!(%);
  (-1) A(X) A(Y)(Z)(Q);
```

The minus sign arises because it would also have been present when sorting `(Y*Z*Q)*X`.

Note that this does not work when the arguments are given as a list separated by commas, for instance as `A(Y,Z,Q)`.

`::CommutingAsSum`
This makes an object behave, for purposes of commutation rules, as a sum.

```
▷ A(#)::CommutingAsSum.
▷ {Q,X,Y,Z}::AntiCommuting.
▷ A(Y)(Z)*A(X);
  @prodsort!(%);
  (-1) A(X) A(Y)(Z);
```

The minus sign arises because it would also have been present when sorting the expression `(Y+Z)*X`.

Note that this does not (yet) work when the arguments are given as a list separated by commas, as in `A(Y,Z)`.

`::Traceless`
Indicates that the tensor is traceless: any internal index contraction makes the tensor vanish.

```
▷ A_{m n}::Traceless.
▷ A_{m}^{m};
▷ @canonicalise!(%);
  0;
```

`::KroneckerDelta`
Denotes a generalised Kronecker delta symbol. When the symbol carries two indices, it is the usual Kronecker delta. When the number of indices is larger, the meaning is

$$\delta_{m_1}{}^{n_1}{}_{m_2}{}^{n_2}{}_{\ldots m_k}{}^{n_k} = \delta_{[m1}{}^{n_1}\delta_{m_2}{}^{n_2}\cdots\delta_{m_k]}{}^{n_k}\,, \tag{5.1}$$

with unit weight anti-symmetrisation.

A symbol which is declared as a Kronecker delta has the property that it can be taken in and out of derivatives. The algorithm `@eliminate_kr` eliminates normal Kronecker deltas by appropriately renaming indices (in order to eliminate Kronecker deltas with more than two indices, first use `@breakgendelta`).

`::EpsilonTensor`**(***metric*=metric object*, delta*=Kronecker delta name**)**
A fully anti-symmetric tensor, defined by

$$\epsilon_{m_1\ldots m_k} := \varepsilon_{m_1\ldots m_k}\sqrt{|g|}\,, \tag{5.2}$$

where the components of $\varepsilon_{m_1\ldots m_k}$ are 0, $+1$ or -1 and $\varepsilon_{01\cdots k} = 1$, independent of the basis, and g denotes the metric determinant.

This property optionally takes a tensor which indicates the symbol which should be used as a `KroneckerDelta` symbol when writing out the product of two epsilon tensors. Additionally, it takes a tensor which is the associated metric, from which the signature can be extracted. See the documentation of `@epsprod2gendelta` for more information on the use of these optional arguments.

When the indices are in different positions it is understood that they are simply raised with the metric. This in particular implies

$$\epsilon^{m_1\ldots m_k} := g^{m_1 n_1}\cdots g^{m_k n_k}\epsilon_{n_1\ldots n_k} = \frac{\varepsilon^{m_1\ldots m_k}}{\sqrt{|g|}}\,, \tag{5.3}$$

again with $\varepsilon^{m_1\ldots m_k}$ taking values 0, $+1$ or -1 and $\varepsilon^{01\cdots k} = \pm 1$ depending on the signature of the metric.

`::SortOrder`
A list property which determines the preferred order of objects when a `@prodsort` command is used.

```
▷ {B,A,C,A_{m}}::SortOrder.
▷ A A_{m} B C;
  @prodsort!(%);
  B A C A_{m};
```

As indicated in the example above, any type of object can appear in the list.

`::Derivative`
An generic derivative object, satisfying the Leibnitz rule. These generic derivatives do not have to commute.

```
▷ D{#}::Derivative.
▷ D(A B C);
  @prodrule!(%);
  D(A) B C + A D(B) C + A B D(C);
```

Refer to the documentation of `PartialDerivative` on how to write derivatives with respect to coordinate indices or coordinates.

Make sure to declare the derivative either using the "with any arguments" notation as used above (using the hash mark), or by giving an appropriate pattern. The following does not work:

```
▷ D::Derivative.
▷ D(A B C);
  @prodrule!(%);
  @prodrule: not applicable.
```

The pattern `D` above does not match the expression `D(A B C)` and hence the algorithm does not know that `D(A B C)` is a derivative acting on the product of three objects.

`::PartialDerivative`
Makes an object a partial derivative, i.e. a derivative which commutes. The object on which it acts has to be a non-sub/superscript child, while all the sub- or superscript child nodes are interpreted to be the variables with respect to which the derivative is taken.

```
▷ \partial{#}::PartialDerivative.
▷ A_{\mu}::Depends(\partial).
▷ \partial_{\nu}{A_{\mu} B_{\rho}};
▷ @prodrule!(%);
  \partial_{\nu}{A_\mu} B_{\rho}
```

Note that derivative objects do not necessarily need to have a sub- or superscript child, they can be abstract derivatives as in

```
▷ D(d?)::PartialDerivative.
▷ D(c d e);
▷ @prodrule!(%);
  D(c) d e + c D(d) e + c d D(e);
```

If you want to write a derivative with respect to a coordinate (instead of with respect to an index, as in the first example above), refer to the `Coordinate` property.

algorithms:

`@prodflatten`
Removes brackets in a product, that is, makes changes of the type

$$a * (b * c) \to a * b * c\,. \tag{5.4}$$

In terms of code this reads

```
▷ a (b c);
▷ @prodflatten!(%);
  a b c;
```

The above is sometimes required because it is possible to write and keep nested products with brackets in order to improve readability of expressions.

`@sumflatten`
Removes brackets in a sum, that is, makes changes of the type

$$a + (b + c) \to a + b + c\,. \tag{5.5}$$

In terms of code this reads

```
▷ a + (b + c);
▷ @sumflatten!(%);
  a + b + c;
```

The above is sometimes required because it is possible to write and keep nested sums with brackets, like in the example above, in order to improve readability of expressions.

`@remove_indexbracket`
When a single symbol-with-indices is entered as

```
▷ (A)_{\mu\nu};
```

it gets wrapped in an `\indexbracket` node. In order to remove this node again, and produce

```
▷ A_{\mu\nu}
```

again, use `@remove_indexbracket`.

```
▷ (A)_{\mu\nu};
▷ @remove_indexbracket!(%);
  A_{\mu\nu};
```

This algorithm is also useful after the `@distribute` command has been called on an `\indexbracket` node which contained a sum; the `@remove_indexbracket` will then remove the superfluous brackets around the single symbols.

`@expand_power`
Expand powers into repeated products, i.e. do the opposite of `@collect_factors`. For example,

```
▷ (A B)**3:
▷ @expand_power!(%);
  (A * B) * (A * B) * (A * B);
▷ @prodflatten!(%):
▷ @prodsort!(%);
  A A A B B B;
▷ @collect_factors!(%);
  A**3 * B**3;
```

This command automatically takes care of index relabelling when necessary, as in the following example,

```
▷ {m,n,p,q,r}::Indices(vector).
▷ (A_m B_m)**3:
▷ @expand_power!(%):
▷ @prodflatten!(%);
  A_{m} * B_{m} * A_{n} * B_{n} * A_{p} * B_{p};
```

`@drop_weight`

Drop those terms for which a product has the indicated weight. Weights are computed by making use of the `Weight` property of symbols. This algorithm does the opposite of `@keep_weight`.

As an example, consider the simple case in which we want to drop all terms with 3 fields. This is done using

```
▷ {A,B}::Weight(label=field).
▷ A B B + A A A + A B + B:
▷ @drop_weight!(%){field}{3};
  A B + B;
```

However, you can also do more complicated things by assigning non-unit weights to symbols, as in the example below,

```
▷ {A,B}::Weight(label=field).
▷ C::Weight(label=field, value=2).
▷ A B B + A A A + A B + A C + B:
▷ @drop_weight!(%){field}{3};
  A B + B;
```

Weights can be "inherited" by operators by using the `WeightInherit` property. Here is an example using partial derivatives,

```
▷ {\phi,\chi}::Weight(label=small, value=1).
▷ \partial{#}::PartialDerivative.
▷ \partial{#}::WeightInherit(label=all, type=Multiplicative).
▷ \phi \partial_{0}{\phi} + \partial_{0}{\lambda}
                                    + \lambda \partial_{3}{\chi}:
▷ @drop_weight!(%){small}{1};
  \phi \partial_{0}{\phi} + \partial_{0}{\lambda};
```

`@keep_weight`

Keep only those terms for which a product has the indicated weight. Weights are computed by making use of the `Weight` property of symbols. This algorithm does the opposite of `@drop_weight`.

As an example, consider the simple case in which we want to keep all terms with 3 fields. This is done using

```
▷ {A,B}::Weight(label=field).
▷ A B B + A A A + A B + B:
▷ @keep_weight!(%){field}{3};
  A B B + A A A
```

However, you can also do more complicated things by assigning non-unit weights to symbols, as in the example below,

```
▷ {A,B}::Weight(label=field).
▷ C::Weight(label=field, value=2).
▷ A B B + A A A + A B + A C + B:
▷ @keep_weight!(%){field}{3};
  A B B + A A A + A C
```

Weights also apply to tensorial expressions. Consider e.g. a situation in which we have a polynomial of the type

$$c^a + c^a_b x^b + c^a_{bc} x^b x^c + c^a_{bcd} x^b x^c x^d; \tag{5.6}$$

and we want to keep only the quadratic term. This can be done using

```
▷ x^{a}::Weight(label=crd, value=1).
▷ c^{#}::Weight(label=crd, value=0).
▷ c^{a} + c^{a}_{b} x^{b} + c^{a}_{b c} x^{b} x^{c} + c^{a}_{b c d} x^{b} x^{c} x^{d};
▷ @keep_weight!(%){crd}{2};
  c^{a}_{b c} x^{b} x^{c};
```

Weights can be "inherited" by operators by using the `WeightInherit` property. Here is an example using partial derivatives,

```
▷ {\phi,\chi}::Weight(label=small, value=1).
▷ \partial{#}::PartialDerivative.
▷ \partial{#}::WeightInherit(label=all, type=Multiplicative).
▷ \phi \partial_{0}{\phi} + \partial_{0}{\lambda}
                                  + \lambda \partial_{3}{\chi}:
▷ @keep_weight!(%){small}{1};
  \lambda \partial_{3}{\chi};
```

If you want to use weights for dimension counting, in which operators can also carry a dimension themselves (e.g. derivatives), then use the `self` attribute,

```
▷ {\phi,\chi}::Weight(label=length, value=1).
▷
▷ x::Coordinate.
▷ \partial{#}::PartialDerivative.
  \partial{#}::WeightInherit(label=length, type=Multiplicative, self=-1).
▷
  \phi \partial_{x}{\phi} + \phi\chi + \partial_{x}{ \phi \chi**2 };
  @keep_weight!(%){length}{1};
```

This keeps only the first term.

`@distribute`

Rewrite a product of sums as a sum of products, as in

$$a\,(b+c) \to a\,b + a\,c\,. \tag{5.7}$$

This would read

```
▷ a (b+c);
▷ @distribute!(%);
  a b + a c;
```

The algorithm in fact works on all objects which carry the `Distributable` property,

```
▷ Op{#}::Distributable.
▷ Op(A+B);
▷ @distribute!(%);
  Op(A) + Op(B);
```

The primary example of a property which inherits the `Distributable` property is `PartialDerivative`. The `@distribute` algorithm thus also automatically writes out partial derivatives of sums as sums of partial derivatives,

```
▷ \partial{#}::PartialDerivative.
▷ \partial_{m}(A + B + C):
▷ @distribute!(%);
  \partial_{m}(A) + \partial_{m}(B) + \partial_{m}(C);
```

`@prodrule`

Apply the product rule or "Leibnitz identity" to an object which has the `Derivative` property, i.e.

$$D(f\,g) = D(f)\,g + f\,D(g)\,. \tag{5.8}$$

In terms of actual code this example would read

```
▷ D{#}::Derivative.
▷ D(f g);
▷ @prodrule!(%);
  D(f) g + f D(g);
```

This of course also works for derivatives which explicitly mention indices or components, as well as for multiple derivatives, as in the example below.

```
▷ D{#}::Derivative.
▷ D_{m n}(f g);
▷ @prodrule!(%);
```

```
▷ @distribute!(%);
▷ @prodrule!(%);
   D_{n}{D_{m}(f)} g + D_{m}(f) D_{n}{g}
               + D_{n}{f} D_{m}(g) + f D_{n}{D_{m}(g)};
```

`@sumsort`

Sort terms in a sum, taking into account any `SortOrder` properties, or else sorting lexographically. This is often useful in case sums appear as exponents; in this case it is necessary to first sort the sums before terms can be collected, as the following example shows.

```
▷ a**(-1+d) - a**(d-1);
▷ @collect_terms!(%);
  a**(-1+d) - a**(d-1);
▷ @sumsort!(%):
▷ @collect_terms!(%);
  0;
```

For more information on `SortOrder` see the documentation of `@prodsort`.

`@prodsort`

Sort factors in a product, taking into account any `SortOrder` properties. Also takes into account commutativity properties, such as `SelfCommuting`. If no sort order is given, it first does a lexographical sort based on the name of the factor, and if two names are identical, does a sort based on the number of children and (if this number is equal) a lexographical comparison of the names of the children.

The simplest sort is illustrated below,

```
▷ C B A D;
▷ @prodsort!(%);
  A B C D;
```

We can declare the objects to be anti-commuting, which then leads to

```
▷ {A, B, C, D}::AntiCommuting.
▷ C B A D;
▷ @prodsort!(%);
  (-1) A B C D;
```

For indexed objects, the anti-commutativity of components is indicated using the `SelfAntiCommuting` property,

```
▷ \psi_{m}::SelfAntiCommuting.
▷ \psi_{n} \psi_{m} \psi_{p};
```

```
▷ @prodsort!(%);
  (-1)  \psi_{m} \psi_{n} \psi_{p};
```

Finally, the lexographical sort order can be overridden by using the `SortOrder` property,

```
▷ {D, C, B, A}::SortOrder.
▷ {A, B, C, D}::AntiCommuting.
  C B A D;
  @prodsort!(%);
  (-1) D C B A;
```

`@collect_factors`
Collect factors in a product that differ only by their exponent. Note that factors containing sub- or superscripted indices do not get collected (i.e. $A_m A^m$ does not get reduced to $(A_m)^2$).

```
▷ A A B A B A;
▷ @collect_factors!(%);
  A**4 B**2;
```

Arbitrary powers can be collected this way,

```
▷ X X**(-1) X**(-4);
▷ @collect_factors!(%);
  X**(-3);
```

The exponent notation can be expanded again using `@expand_power`.

`@collect_terms`
Collect terms in a sum that differ only by their numerical pre-factor. This is called automatically on all new input, and also by some algorithms (in which case it will be indicated in the description of the command), but in general has to be called by hand.

Note that this command only collects terms which are identical, it does not collect terms which are different but mathematically equivalent. See `@sumsort` for an example.

`@factor_out`
Given a list of symbols, this algorithm tried to factor those symbols out of terms. As an example,

```
▷ a b + a c e + a d:
▷ @factor_out!(%){a};
  a ( b + c e + d );
```

In case you are familiar with FORM, `factor_out` is like its `bracket` statement. If you add more factors to factor out, it works as in

```
▷ a b + a c e + a c + c e + c d + a d:
▷ @factor_out!(%){a,c};
  a (b + d) + c ( e + d ) + a c (e + 1);
```

That is, separate terms will be generated for terms which differ by powers of the factors to be factored out.

The algorithm of course also works with indexed objects, as in

```
▷ A_{m} B_{m} + C_{m} A_{m};
▷ @factor_out!(%){A_{m}};
  A_{m} (B_{m} + C_{m}) ;
```

(this is still work in progress).

`@factor_in`

Given a list of symbols, this algorithm collects terms in a sum that only differ by pre-factors consisting of these given symbols. As an example,

```
▷ a b + a c + a d:
▷ @factor_in!(%){b,c};
  (b + c) a + a d;
```

The name is perhaps most easily understood by thinking of it as a complement to `factor_out`. Or in case you are familiar with FORM, `factor_in` is like its `antibracket` statement.

The algorithm of course also works with indexed objects, as in

```
▷ A_{m} B_{m} + C_{m} A_{m};
▷ @factor_in!(%){B_{n},C_{n}};
  (B_{m} + C_{m}) A_{m};
```

(this is still work in progress).

`@canonicalise`

Canonicalise a product of tensors, using the mono-term index symmetries of the individual tensors and the exchange symmetries of identical tensors. Tensor exchange takes into account commutativity properties of identical tensors.

Note that this algorithm does not take into account multi-term symmetries such as the Ricci identity of the Riemann tensor; those canonicalisation procedures require the use of `@young_project_tensor` or `@young_project_product`. Similarly, dimension-dependent identities are not taken into account, use `@decompose_product` for those.

In order to specify symmetries of tensors you need to use symmetry properties such as `Symmetric`, `AntiSymmetric` or `TableauSymmetry`. The following example illustrates this.

```
▷ A_{m n}::AntiSymmetric.
▷ B_{p q}::Symmetric.
▷ A_{m n} B_{m n};
▷ @canonicalise!(%);
  0;
```

If the various terms in an expression use different index names, you may need an additional call to `@rename_dummies` before `@collect_terms` is able to collect all terms together:

```
▷ {m,n,p,q,r,s}::Indices.
▷ A_{m n}::AntiSymmetric.
▷ C_{p q r}::AntiSymmetric.
▷ A_{m n} C_{m n q} + A_{s r} C_{s q r};
▷ @canonicalise!(%);
  A_{m n} * C_{q m n} - A_{r s} * C_{q r s};
▷ @rename_dummies!(%);
  A_{m n} * C_{q m n} - A_{m n} * C_{q m n};
▷ @collect_terms!(%);
  0;
```

If you have symmetric or anti-symmetric tensors with many indices, it sometimes pays off to sort them to the end of the expression (this may speed up the canonicalisation process considerably).

`@reduce`

NOT YET FUNCTIONAL

Reduce a sum of tensor monomials of the same type, taking into account mono-term as well as multi-term symmetries. This does not aim for a canonical form, but rather for a form in which no monomials occur which can be expressed (using mono- and multi-term symmetries) as a linear combination of other monomials appearing in the sum. An example makes this more clear:

```
▷ {m,n,p,q}::Indices(vector).
  {m,n,p,q}::Integer(0..10).
  R_{m n p q}::RiemannTensor.

  R_{m n q p} R_{m n p q} + R_{m p n q} R_{m n p q};
  @reduce!(%);
```

`@young_project`

Project the indicated expression onto a Young tableau representation. This includes

the normalisation factor, such that applying the operation twice does not change the result anymore. For example,

```
▷ A_{m n} B_{p}:
▷ @young_project!(%){2,1}{0,1,2};
    1/3 A_{m n} B_{p} + 1/3 A_{n m} B_{p}
  - 1/3 A_{p n} B_{m} - 1/3 A_{n p} B_{m};
▷ @young_project!(%){2,1}{0,1,2}:
▷ @sumflatten!(%):
  @collect_terms!(%);
    1/3 A_{m n} B_{p} + 1/3 A_{n m} B_{p}
  - 1/3 A_{p n} B_{m} - 1/3 A_{n p} B_{m};
```

The first argument gives the tableau shape, while the second argument gives the index position associated to each box in the Young tableau (similar to the way in which the `TableauSymmetry` property works; note that this algorithm does not require the tensors to have any specific symmetries). The index positions given in the second argument count from zero.

`@all_contractions`
Construct all full contractions of the given tensors, taking into account mono-term symmetries. Example,

```
▷ A_{m n}::Symmetric.
▷ A_{m n}::Traceless.
▷ {m,n,p,q,r,s,t,u}::Indices(vector).
▷ {m,n,p,q,r,s,t,u}::Integer(0..9).
▷ obj14:= A_{m n} A_{p q} A_{r s} A_{t u};
  @all_contractions(%){2};
  A_{m n} A_{m n} A_{p q} A_{p q} + A_{m n} A_{m p} A_{n q} A_{p q};
```

This command is not entirely correct at the moment: it determines the total number of singlets correctly, but does not generate parity-odd contractions (the ones involving an epsilon tensor). It also does not know about dimension-dependent identities. If this is a problem for your application, contact the author for an experimental version of cadabra.

5.4 Young tableaux

Young diagrams (with unlabelled boxes) and Young tableaux (with labelled boxes) can be manipulated directly, and cadabra contains routines to e.g. compute tensor products or to put tableaux in standard form using all their symmetries. In order to avoid confusion about the meaning of "diagram" and "tableau", a Young tableau with labelled boxes is called a "FilledTableau" in cadabra, while a tableau without any labels in the boxes is a "Tableau".

properties:

`::Tableau`**(***dimension*=integer*)*
Indicates that the object is a Young diagram, i.e. a tableau without labels in the boxes. The arguments of a `Tableau` property denote the lengths of the successive rows,

```
▷ \tab{#}::Tableau.
▷ \tab{4}{2}{1};
```

In the graphical interface this will display as

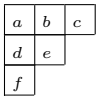
For several algorithms, the dimension of the permutation group associated to the tableau must be known; this can be added as in the example below,

```
▷ \tab{#}::Tableau(dimension=10).
```

`::FilledTableau`**(***dimension*=integer*)*
Indicates that the object is a Young tableau with labelled boxes. The arguments of a `Tableau` property denote the contents, given as lists,

```
▷ \ftab{#}::FilledTableau.
▷ \ftab{a,b,c}{d,e}{f};
```

In the graphical interface this will display as



See `Tableau` for information about adding the dimension of the permutation group.

algorithms:

`@tabdimension`
Compute the dimension of the representation corresponding to a tableau or filled tableau. This algorithm acts on objects with the `Tableau` or `FilledTableau` property, which should have the `dimension` argument set.

```
▷ \tableau{#}::Tableau(dimension=10).
▷ \tableau{2}{2};
▷ @tabdimension!(%);
  825;
▷ \tableau{#}::Tableau(dimension=8).
▷ \tableau{2}{2};
▷ @tabdimension!(%);
  336;
```

The computation uses the standard hook rule.

`@tabcanonicalise`
Canonicalise a tableau, that is, sort equal-length columns and sort boxes within columns. This algorithm acts on objects with the `FilledTableau` property.

```
▷ \tableau{#}::FilledTableau.
▷ \tableau{b,d}{c,a};
▷ @tabcanonicalise!(%);
  -\tableau{a,b}{d,c};
```

This algorithm only uses mono-term symmetries of Young tableaux, i.e. symmetries which relate the given tableau to one other tableau. Garnir symmetries (generalised Bianchi and Ricci identities) are not handled by `@tabcanonicalise` but require the use of `@tabstandardform`.

`@tabstandardform`
Bring a filled Young tableau to standard form, that is, to a form in which two horizontally adjacent boxes appear in sorted form. This requires the use of Garnir symmetries and generically produces a sum of tableaux. This algorithm acts on objects with the `FilledTableau` property.

```
▷ \tableau{#}::FilledTableau.
▷ \tableau{b,d}{c,a};
▷ @tabstandardform!(%);
  \tableau{a,c}{b,d} - \tableau{a,b}{c,d};
```

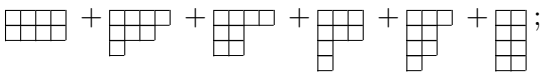
In the graphical interface this displays as

$$\begin{array}{|c|c|}\hline a & c \\ \hline b & d \\ \hline \end{array} - \begin{array}{|c|c|}\hline a & b \\ \hline c & d \\ \hline \end{array}\,; \tag{5.9}$$

`@lr_tensor`
Compute the tensor product of two tableaux or filled tableaux. The algorithm acts on objects which have the `Tableau` or `FilledTableau` property, through which it is possible to set the dimension. The standard Littlewoord-Richardson algorithm is used to construct the tableaux in the tensor product. An example with `Tableau` objects is given below.

```
▷ \tableau{#}::Tableau(dimension=10).
▷ \tableau{2}{2} \tableau{2}{2};
▷ @lr_tensor!(%);
  \tableau{4 4} + \tableau{4 3 1} + \tableau{4 2 2}
       + \tableau{3 3 1 1} + \tableau{3 2 2 1}
       + \tableau{2 2 2 2};
```

In the graphical interface the output will show up as proper Young tableaux,

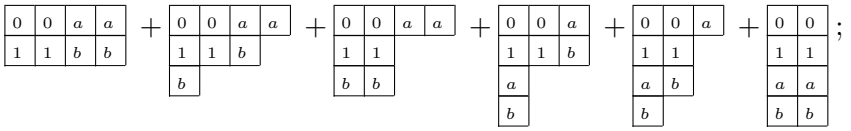


(5.10)

The same example, but now with `FilledTableau` objects, is

```
▷ \tableau{#}::FilledTableau(dimension=10).
▷ \tableau{0,0}{1,1} \tableau{a,a}{b,b}:
▷ @lr_tensor!(%);
```

This will again output a sum of `\tableau` objects. In the graphical interface they will be typeset as



(5.11)

`@decompose`

Decompose a tensor monomial on a given basis of monomials. The basis should be given in the second argument. All tensor symmetries, including those implied by Young tableau Garnir symmetries, are taken into account. Example,

```
▷ {m,n,p,q}::Indices(vector).
▷ {m,n,p,q}::Integer(0..10).
▷ R_{m n p q}::RiemannTensor.

▷ R_{m n q p} R_{m p n q};
▷ @decompose!(%)( R_{m n p q} R_{m n p q} );
  { -1/2 };
```

Note that this algorithm does not yet take into account dimension-dependent identities, but it is nevertheless already required that the index range is specified.

`@decompose_product`

Decompose a product of tensors by writing it out in terms of irreducible Young tableau representations, discarding the ones which vanish in the indicated dimension, and putting the results back together again. This algorithm can thus be used to equate terms which are identical only in certain dimensions.

If there are no dimension-dependent identities playing a role in the product, then `@decompose_product` returns the original expression,

```
▷ { m, n, p, q }::Indices(vector).
▷ { A_{m n p}, B_{m n p} }::AntiSymmetric.
▷ A_{m n p} B_{m n q} - A_{m n q} B_{m n p}.
▷ { m, n, p, q }::Integer(1..4).
▷ @decompose_product!(%);
▷ @canonicalise!(%);
▷ @collect_terms!(%);
  A_{m n p} B_{m n q} - A_{m n q} B_{m n p};
```

However, in the present example, a Schouten identity makes the expression vanish identically in three dimensions,

```
▷ { m, n, p, q }::Integer(1..3).
▷ @decompose_product!(%);
▷ @canonicalise!(%):
▷ @collect_terms!(%);
  0;
```

Note that `@decompose_product` is unfortunately computationally expensive, and is therefore not practical for large dimensions.

5.5 Lists and ranges

When using list algorithms, be aware of the fact that when they act on an argument, they actually replace the argument, not generate a new one. Therefore, to generate a new expression which contains the length of another expression,

```
▷ A+B+C+D;
  1:= A+B+C+D;
▷ @length[@(1)];
  2:= 4;
```

This is to be compared with

```
▷ A+B+C+D;
  1:= A+B+C+D;
▷ @length(1);
  1:= 4;
```

in which case the first expression gets replaced by its length.

algorithms:

`@length`
Replaces the expression with its length, that is, the number of terms in a sum, the number of factors in a product or the number of elements in a list.

```
▷ lst:= {a,b,c,d}:
▷ @length(%);
  lst:= 4;
▷ sum:= a + b + c + d + e:
▷ @length(%);
  sum:= 5;
▷ prod:= a*b*c:
▷ @length(%);
  prod:= 3;
```

Note that the original expression gets replaced; if you need a new expression use a square-bracket construction as in

```
▷ sum:= a + b + c + d;
▷ @length[@(sum)];
  2:= 4;
```

Here is a more complicated example which takes all terms except the first and last from an expression:

```
▷ a+b+c+d+e+f+g;
▷ @take(%){ 1..@collect_terms[@length[@(%)]-2] };
  b+c+d+e+f;
```

(remember that counting terms starts from zero).

If you just want to display the number of elements, use `@number_of_terms`; this is an output command and does not change the expression.

`@take`

Replace a sum, product or list with a subset of its terms, factors or elements. Example,

```
▷ A+B+C+D+E;
▷ @take(%){1,3};
  B+D;
▷ {A,B,C,D,E};
▷ @take(%){4};
  \{ E \};
▷ A*B*C*D*E;
▷ @take(%){2..\infty};
  C*D*E;
```

As usual, a range can be open by setting the second boundary to `\infty`. In order to select terms based on pattern matching, see `@take_match`

`@range`

Replaces a two-element list of integers with the elements in the range,

```
▷ {-3,2};
▷ @range(%);
  \{ -3,-2,-1,0,1,2 \}
▷ @range[{1,3}];
  \{ 1,2,3 \}
```

When the list has three elements, the algorithm generates a list consisting of a number of copies of the third element of the given list,

```
▷ @range[{1,3,c}];
  {c,c,c};
```

When the list contains four elements, the first one is used as a counter, and can appear in the fourth element to generate different list elements,

```
▷ @range[{i, 1, 5, c_{i} }];
  { c_1, c_2, c_3, c_4, c_5 };
```

`@inner`

Construct an inner product between two lists appearing in a list,

```
▷ lst1:= {a_m, b_m ,c_m ,d_m ,e_m }:
▷ lst2:= @range[{1, @length[@(lst1)]}];
  \{ 1,2,3,4,5 \};
▷ @inner[ { @(lst1), @(lst2) } ];
  a_m + 2 b_m + 3 c_m + 4 d_m + 5 e_m;
```

In order to generate a list of the sum of elements, use `@list_sum` instead.

`@list_sum`
In an expression containing sums of identical-length lists, create one new list constructed by adding the elements at the same position in each list. Example,

```
▷ {a, b, 7 c + q, d, e} - {-f, g, h, i, j}:
▷ @list_sum!(%):
▷ @sumflatten!(%);
  \{ a + f, b - g, 7c + q - h, d - i, e - j \};
```

To generate a list of the products of the elements, use `@inner` instead.

`@listflatten`
Turns nested lists into one list, as in

$$\{a, b, \{c, d\}\} \to \{a, b, c, d\}\,. \tag{5.12}$$

In terms of code this reads

```
▷ {a,b,{c,d}};
▷ @listflatten!(%);
  \{ a,b,c,d \};
```

`@permute`
Generic algorithm to generate permutations and combinations of elements in a list. It takes several arguments depending on the type of permutations which one wishes to generate.

The simplest permutations are generated by stating whether items can be taken from the original set more than once (multiple-pick or single-pick), and stating the length of the permutation sets to be generated. An example with single-pick is

```
▷ {a,b,c,d};
▷ @permute!(%){false}{3};
  {{a, b, c}, {a, b, d}, {a, c, d}, {b, c, d}};
```

while an example with multiple-pick is

```
▷ {a,b,c,d};
▷ @permute!(%){true}{2};
  {{a, a}, {a, b}, {a, c}, {a, d}, {b, b}, {b, c},
                           {b, d}, {c, c}, {c, d}, {d, d}};
```

The length parameter is in fact fixing the length of the subsets in the result for which the order of the elements does not matter. Therefore,

```
▷ {a,b,c};
▷ @permute!(%){false}{1,1};
  {{a, b}, {a, c}, {b, a}, {b, c}, {c, a}, {c, b}};
```

Finally, it is also possible to generate all permutations with a given maximal length,

```
▷ {a,b,c};
▷ @permute!(%){false}{<3};
  {{a}, {b}, {c}, {a, b}, {a, c}, {b, c}};
```

More complicated permutations can be generated by assigning weights to the objects in the original list. Here is an example in which a, b, c have weights $2, 1, 0$ respectively, and lists of length 3 are generated for which the total weight is 4.

```
▷ {a,b,c};
▷ @permute!(%){true}{3}{2,1,0}{4};
  {{a, a, c}, {a, b, b}};
```

Objects can be assigned more than one type of weight by simply repeating the two arguments associated to weights. For instance, if we in addition introduce another weight type, under which a, b, c have weight $0, 1, 0$ respectively, and also impose that this weight should add up to 2, we get

```
▷ {a,b,c};
▷ @permute!(%){true}{3}{2,1,0}{4}{0,1,0}{2};
  {{a, b, b}};
```

This functionality is useful when constructing field polynomials restricted to certain length dimension or Grassmann number.

5.6 Properties

properties:

`::PropertyInherit`
Indicates that an object should inherit all properties of the objects they wrap. In most cases, the use of `Accent` is more appropriate, since it also makes objects inherit indices.

`::IndexInherit`
Indicates that an object should inherit the indices of its child objects. This is useful mainly for operators. Matrix transposition, for instance, could be written as

```
▷ T{#}::IndexInherit.
▷ T(B_{m n}) + C_{m n};
```

Without the `IndexInherit` property, the object `T(B_{m n})` would be considered a scalar, without indices, and an index mismatch error would be reported. With the property, the first term has external indices `m` and `n`, just like the second term.

This property is automatically associated to all `Derivative` operators.

5.7 Dummy indices

Dummy indices are a subject by itself. There are various problems with most implementations of dummy indices in computer algebra software. In many cases, dummy indices are added as an afterthought, leading to situations were illegal expressions (e.g. with a more than twice repeated dummy index) can be entered. Even if this is flagged, most programs do not know about different index types (which leads to problems when there is more than one type of contraction, or a sum over only part of the index range). In cadabra, the system knows about dummy index *sets*, and all algorithms can request new dummy indices very easily. This is handled through the `dummies` module.

properties:

`::Indices(`*name*=set name*, parent*=parent set name*, position=free|fixed|independent)*
Declare index names to be usable for dummy index purposes. Typical usage is of the form

```
▷ {r,s,t}::Indices(vector).
▷ {a,b,c,d}::Indices(spinor).
```

This indicates the name of the index set ("vector" resp. "spinor" in the example above).

Indices can occur as subscripts or superscripts, and you may use this to indicate e.g. covariant and contravariant transformation behaviour. In this case, use the additional argument `position=fixed` to indicate that the position carries meaning. If you do not want cadabra to automatically raise or lower indices when canonicalising expressions, or if upper and lower indices are not related at all, use `position=independent`.

When you work with vector spaces which are subspaces of larger spaces, it is possible to indicate that a given set of indices take values in a subset of values of a larger set. An example makes this more clear. Suppose we have one set of indices A, B, C which take values in a four-dimensional space, and another set of indices a, b, c which take values in a three-dimensional subspace. This is declared as

```
▷ {A,B,C}::Indices(fourD).
▷ {a,b,c}::Indices(threeD, parent=fourD).
```

This will allow cadabra to canonicalise expressions which contain mixed index types, as in

```
▷ {A,B,C}::Indices(fourD).
▷ {a,b,c}::Indices(threeD, parent=fourD).
▷ M_{q? r?}::AntiSymmetric.
▷ M_{a A} + M_{A a}:
▷ @canonicalise!(%):
▷ @collect_terms!(%);
  0;
```

Note the way in which the symmetry of the M tensor was declared here.

algorithms:

`@rename_dummies`
Rename the dummy indices in an expression. This can be necessary in order to make various terms in a sum use the same names for the indices, so that they can be collected.

```
▷ {m,n,p,q,r,s}::Indices(vector).
▷ A_{m n} B_{m n} - A_{p q} B_{p q};
▷ @rename_dummies!(%);
  A_{m n} B_{m n} - A_{m n} B_{m n};
▷ @collect_terms!(%);
  0;
```

Note that the indices need to have been declared as being part of an index list, using the `Indices` property.

5.8 Symmetrisation and anti-symmetrisation

This module contains algorithms for the symmetrisation or anti-symmetrisation of expressions in their elements, and for the inverse procedure, whereby object trees are identified when they differ only by certain symmetrisations or anti-symmetrisations.

algorithms:

`@sym`
Symmetrise a product or tensor in the indicated objects. This works both with normal objects, as in

```
▷ A B C;
▷ @sym!(%){A,B,C};
  1/6 A B C + 1/6 A C B + 1/6 B A C
              + 1/6 B C A + 1/6 C A B + 1/6 C B A;
```

as well as with indices. When used with indices, remember to also indicate whether you want to symmetrise upper or lower indices, as in the example below.

```
▷ A_{m n} B_{p};
▷ @sym!(%){ _{m}, _{n}, _{p} };
  1/6 A_{m n} B_{p} + 1/6 A_{m p} B_{n} + 1/6 A_{n m} B_{p}
     + 1/6 A_{n p} B_{m} + 1/6 A_{p m} B_{n} + 1/6 A_{p n} B_{m};
```

Anti-symmetrisation (i.e. introducing a sign depending on the permutation of each term) is handled by the `@asym` algorithm.

`@asym`
Anti-symmetrise a product or tensor in the indicated objects. This works both with normal objects, as in

```
▷ A B C;
▷ @asym!(%){A,B,C};
  1/6 A B C - 1/6 A C B - 1/6 B A C
            + 1/6 B C A + 1/6 C A B - 1/6 C B A;
```

as well as with indices. When used with indices, remember to also indicate whether you want to symmetrise upper or lower indices, as in the example below.

```
▷ A_{m n} B_{p};
▷ @asym!(%){ _{m}, _{n}, _{p} };
  1/6 A_{m n} B_{p} - 1/6 A_{m p} B_{n} - 1/6 A_{n m} B_{p}
       + 1/6 A_{n p} B_{m} + 1/6 A_{p m} B_{n} - 1/6 A_{p n} B_{m};
```

Symmetrisation (i.e. using plus signs for all terms) is handled by the `@sym` algorithm.

`@canonicalorder`
Orders the indicated objects in the expression in canonical order. On a simple product of objects it works as a partial product sort,

```
▷ C B E D A;
▷ @canonicalorder!(%)(A, B, E);
  C A B D E;
```

It can, however, also be used to sort indices. Thereby, it facilitates imposing index symmetry on a tensor with open indices, as the following example illustrates.

```
▷ A^{m n p} B^{q r} + A^{q m} B^{n p r};
▷ @canonicalorder!(%)( ^{m}, ^{n}, ^{p}, ^{r}, ^{q} );
  A^{m n p} B^{r q} + A^{m n} B^{p r q};
```

`@impose_asym`
Impose anti-symmetry in external indices, that is, remove terms in the expression which have anti-symmetry imposed on more than one index sitting on a symmetric object.

`@asymprop`
Remove all traces of anti-symmetric tensors.

5.9 Field theory

properties:

`::SelfDual`
Make a tensor fully anti-symmetric as well as self-dual. The conventions used are that self-duality means

$$F_{\mu_1\cdots\mu_n} = \frac{1}{n!}\epsilon_{\mu_1\cdots\mu_n\mu_{n+1}\cdots\mu_{2n}}F_{\mu_{n+1}\cdots\mu_{2n}}\,. \tag{5.13}$$

This property used by the Young projection algorithms and the `@dualise_tensor` algorithm, among others.

`::AntiSelfDual`
Make a tensor fully anti-symmetric as well as anti-selfdual. The conventions used are that self-duality means

$$F_{\mu_1\cdots\mu_n} = -\frac{1}{n!}\epsilon_{\mu_1\cdots\mu_n\mu_{n+1}\cdots\mu_{2n}}F_{\mu_{n+1}\cdots\mu_{2n}}\,. \tag{5.14}$$

This property used by the Young projection algorithms and the `@dualise_tensor` algorithm, among others.

`::Depends(`*comma separated list of indices, coordinates, derivatives, accents)*
Makes an object implicitly dependent on other objects, i.e. assumes that the indicated object is a function of the arguments of the property. For example,

```
▷ x::Coordinate.
▷ \phi::Depends(x).
```

makes ϕ an implicit function of x. Instead of indicating the coordinate on which the object depends, it is also possible to indicate which derivatives would yield a non-zero answer, as in

```
▷ \nabla{#}::Derivative.
▷ \phi::Depends(\nabla).
```

Finally, it is possible to use an index name to indicate on which coordinates a field depends,

```
▷ {m,n,p,q}::Indices(vector).
▷ \phi::Depends(m).
```

Taking objects out of derivatives (because they do not depend on them) is handled using the `@unwrap` algorithm.

If you want to make an object depend on more than one thing, you need to specify them all in one `Depends` property. If you specify them in two separate properties, the last property will overwrite the previous one. Therefore, you get

```
▷ \hat{#}::Accent.
▷ \partial{#}::PartialDerivative.
▷ A::Depends(\hat).
▷ A::Depends(\partial).
▷ \hat{A};
▷ @unwrap!(%);
  A;
```

instead of `\hat{A}` which you might have expected.

`::DependsInherit`
Makes an object inherit all dependencies of the objects they contain. This makes it possible to define functions of objects which behave correctly inside derivatives.

```
▷ f{#}::DependsInherit.
▷ x::Coordinate.
  \partial{#}::PartialDerivative.
  \phi::Depends(x).
  \partial_{x}( g f(\phi) );
  @unwrap!(%);
  g \partial_{x}( f(\phi) );
```

`::Weight`**(***label*=name*, value*=rational*)*
Attach a labelled weight to an object, which can subsequently be used in the algorithms `@keep_weight` and `@drop_weight`. See the documentation of those algorithms for examples.

`::WeightInherit`**(***label*=name|*all, type*=*Additive*|*Multiplicative, self*=value*)*
Indicates that the object inherits all weights of its child nodes. The label indicates which weights are inherited; use `all` to inherit all weights. The type of inheritance determines whether weights of the child nodes should be added or multiplied.

Additive weight inheritance is used to assign a weight to sum-like objects. For example, if we declare

```
▷ f{#}::WeightInherit(label=all, type=Additive).
  A::Weight(label=field, value=3).
```

then the expression `f{A}{A}` has weight 3. This is what would happen if '`f`' would be a sum: the weight of the sum is the same as the weight of the terms (and is only defined if the weight of the all terms is equal).

Multiplicative weight inheritance is used to assign a weight to product-like objects. For example, if we declare

```
▷ f{#}::WeightInherit(label=all, type=Multiplicative).
▷ A::Weight(label=field, value=3).
```

then the expression `f{A}{A}` has weight 6. This is what would happen if 'f' would be a product: the weight of the product is the same as the sum of the weights of the factors.

Terms can be selected based on their weight by using the `@keep_weight` and `@drop_weight` algorithms.

For more information on related topics, see `IndexInherit` and `PropertyInherit`.

`::Symbol`
If you want to attach symbols to tensors in sub- or superscript, as in $A^\dagger$, you need to inform the system that this symbol is not an index (otherwise it may interpret $A^\dagger A^\dagger$ as two contracted vectors with vector indices †). This is done by associating the `Symbol` property to the symbol; in this example

```
▷ \dagger::Symbol.
▷ A^\dagger A^\dagger;
```

A related effect is induced by the `Coordinate` property.

`::Coordinate`
Declare a symbol to be a coordinate label (useful in combination with `Depends`). This is required if you want to write a derivative with respect to a coordinate: the input

```
▷ A(x,x') + \diff{B(x,x')}_{x};
```

will by default be seen as incorrect because the x in the second term will be considered an index label, not a coordinate. The input

```
▷ { x, x' }::Coordinate.
▷ A(x,x') + \diff{B(x,x')}_{x};
```

is allowed and interpreted in the right way.

`::Accent`
Turns a symbol into an accent. Accented objects inherit all properties and indices from the objects which they wrap. Here is an example with inherited coordinate dependence:

```
▷ \hat{#}::Accent.
▷ \partial{#}::PartialDerivative.
▷ A::Depends(\partial).
▷ \partial(A \hat{A} B):
▷ @prodrule!(%):
▷ @unwrap!(%);
  \partial(A) * A * B + A * \partial(\hat{A}) * B;
```

Without the accent property, `\hat{A}` object would be a completely independent object, unrelated to `A`.

`::SatisfiesBianchi`
Indicates that an object satisfies a (generalised) Bianchi identity. This is often used to link a derivative operator to a curvature tensor, as in

```
▷ R_{m n p q}::RiemannTensor;
▷ D{#}::Derivative.
▷ D_{m}{ R_{n p q r} }::SatisfiesBianchi.
▷ A^{m n p q}::AntiSymmetric.
▷ D_{m}{ R_{n p q r} } A^{m n p q};
▷ @canonicalise!(%);
  (-1) D_{m}{ R_{r n p q} } A^{m n p q};
▷ @impose_bianchi!(%);
  0;
```

This general method allows one to handle more than one type of derivative object.

algorithms:

`@unwrap`
Move objects out of `Derivatives` or `Accents` when they do not depend on these operators.

Accents will get removed from objects which do not depend on them, as in the following example:

```
▷ \hat{#}::Accent.
▷ \hat{#}::Distributable.
▷ B::Depends(\hat).

▷ \hat{A+B+C}:
▷ @distribute!(%);
  \hat{A} + \hat{B} + \hat{C};
▷ @unwrap!(%);
  A + \hat{B} + C;
```

Derivatives will be set to zero if an object inside does not depend on it. All objects which are annihilated by the derivative operator are moved to the front (taking into account anti-commutativity if necessary),

```
▷ \partial{#}::PartialDerivative.
▷ {A,B,C,D}::AntiCommuting.
  x::Coordinate.
▷ {B,D}::Depends(\partial).
▷
  \partial_{x}( A B C D ):
  @unwrap!(%);
  - A C \partial_{x}{B D};
```

Note that a product remains inside the derivative; to expand that use `@prodrule`.

Here is another example

```
▷ \del{#}::Derivative.
▷ X::Depends(\del).
▷ \del{X*Y*Z}:
▷ @prodrule!(%);
  \del{X} * Y * Z + X * \del{Y} * Z + X * Y * \del{Z};
▷ @unwrap!(%);
  \del{X}*Y*Z;
```

Note that all objects are by default constants for the action of `Derivative` operators. If you want objects to stay inside derivative operators you have to explicitly declare that they depend on the derivative operator or on the coordinate with respect to which you take a derivative.

`@eliminate_kr`
Eliminates Kronecker delta symbols by performing index contractions. Also replaces contracted Kronecker delta symbols with the range over which the index runs, if known.

```
▷ \delta_{m n}::KroneckerDelta.
▷ A_{m p} \delta_{p q} B_{q n};
▷ @eliminate_kr!(%);
  A_{m q} B_{q n};
```

The index range is set as usual with `Integer`,

```
▷ {m,n,p,q}::Integer(0..d-1).
▷ \delta_{m n}::KroneckerDelta.
▷ \delta_{p q} \delta_{p q};
▷ @eliminate_kr!(%);
  d;
```

In order to eliminate metric factors (i.e. to 'raise' and 'lower' indices) use the algorithm `@eliminate_metric`.

`@eliminateeps`
Given an epsilon tensor and a self-dual tensor contracted with it, rewrite the self-dual tensor as epsilon times itself.

```
▷ F_{m n p}::SelfDual.
▷ \eps_{m n p q r s}::EpsilonTensor.
  \eps_{m n p q r s} F_{q r s};
  @eliminateeps!(%){F_{m n p}};
```

At a further stage one can then use the command `@epsprod2gendelta` to remove the epsilons altogether the arguments contain a list of all tensors for which self-duality can be applied). Uses the the tensor with the largest number of indices in common with the epsilon symbol (can be more than one, in which case more terms are generated).

`@epsprod2gendelta`
Replace a product of two epsilon tensors with a generalised delta according to the expression

$$\epsilon^{r_1\cdots r_d}\epsilon_{s_1\cdots s_d} = \frac{1}{\sqrt{|g|}}\varepsilon^{r_1\cdots r_d}\sqrt{|g|}\varepsilon_{s_1\cdots s_d} = \operatorname{sgn} g\, d!\, \delta^{r_1\cdots r_d}_{s_1\cdots s_d}\,, \tag{5.15}$$

where $\operatorname{sgn} g$ denotes the signature of the metric g used to raise/lower the indices (see `EpsilonTensor` for conventions on the epsilon tensor). When the indices are not ocurring up/down as in this expression, and the index position is not free, metric objects will be generated instead.

Here is an example:

```
▷ {a,b,c,d}::Indices.
▷ {a,b,c,d}::Integer(1..3).
▷ \delta{#}::KroneckerDelta.
▷ \epsilon_{a b c}::EpsilonTensor(delta=\delta).
▷ \epsilon_{a b c} \epsilon_{a b d};
▷ @epsprod2gendelta!(%);
  2 \delta_{c d};
```

In order for this algorithm to work, you need to make sure that the epsilon symbols in your expression are declared as `EpsilonTensor` *and* that these declarations involve a specification of the `delta` and/or `metric` symbols.

As you can see from this example, contracted indices inside the generalised delta are automatically eliminated, as the command `@reduce_gendelta` is called automatically; if you do not want this use the optional argument “noreduce”.

Note that the results typically depend on the signature of the space-time, which can be introduced through the optional `metric` argument of the `EpsilonTensor` property. Compare the two examples below:

```
▷ {a,b,c,d}::Indices.
▷ {a,b,c,d}::Integer(1..3).
▷ \delta{#}::KroneckerDelta.
▷ \epsilon_{a b c}::EpsilonTensor(delta=\delta, metric=g_{a b}).

▷ g_{a b}::Metric(signature=-1).
▷ \epsilon_{a b c} \epsilon_{a b c};
▷ @epsprod2gendelta!(%);
  -6;

▷ g_{a b}::Metric(signature=+1).
▷ \epsilon_{a b c} \epsilon_{a b c};
```

```
▷ @epsprod2gendelta!(%);
  6;
```

Note that you need to specify the full symbol for the metric, including the indices, whereas the Kronecker delta argument only requires the name without the indices (because a contraction can generate generalised Kronecker delta symbols with any number of indices).

`@dualise_tensor`
Dualises tensors which have been declared `SelfDual` according to the formula

$$F_{\mu_{n+1}\cdots\mu_d} \quad \to \quad *F_{\mu_{n+1}\cdots\mu_d} = \frac{1}{n!}\epsilon_{\mu_{n+1}\cdots\mu_d}{}^{\mu_1\cdots\mu_n} F_{\mu_1\cdots\mu_n}\,. \tag{5.16}$$

In order for this to work the indices on the tensor have to be declared with `Indices` and their range should have been specified with `Integer`.

```
▷ {m,n,p,q,r#}::Indices.
▷ {m,n,p,q,r#}::Integer(0..5).
▷ F_{m n p}::SelfDual.
▷ A_{q} F_{m n p} F_{m n p};
▷ @dualise_tensor!(%);
  1/36 A_{q} \epsilon_{m n p r1 r2 r3} F_{r1 r2 r3}
             \epsilon_{m n p r4 r5 r6} F_{r4 r5 r6};
```

As indices are by default position-free, all indices on the epsilon tensors are generated as lower indices in the example above. Here is a modification which takes index positions into account:

```
▷ {m,n,p,q,r#}::Indices(position=fixed).
▷ {m,n,p,q,r#}::Integer(0..5).
▷ F_{m n p}::SelfDual.
▷ F^{m n p}::SelfDual.
▷ A_{q} F_{m n p} F^{m n p};
  @dualise_tensor!(%);
  1/36 A_{q} \epsilon_{m n p}^{r1 r2 r3} F_{r1 r2 r3}
             \epsilon^{m n p r4 r5 r6} F_{r4 r5 r6};
```

Products of epsilon tensors can be expanded in terms of Kronecker delta symbols with `@epsprod2gendelta`.

`@reduce_gendelta`
Convert generalised delta symbols which contain contracted indices to deltas with fewer indices, according to the formula

$$n!\,\delta^{a_1\cdots a_n}_{b_1\cdots b_n}\,\delta^{b_1}_{a_1}\cdots\delta^{b_m}_{a_m} = \Big[\prod_{i=1}^{m}\big(d-(n-i)\big)\Big]\,(n-m)!\,\delta^{a_{m+1}\cdots a_n}_{b_{m+1}\cdots b_n}\,. \tag{5.17}$$

Here is an example

```
▷ \delta{#}::KroneckerDelta.
▷ {m,n,q}::Integer(0..3).
  \delta_{m}^{n}_{n}^{q};
  @reduce_gendelta!(%);
  (-3/2) \delta_{m}^{q};
```

Note that this requires that the indices on the Kronecker delta symbol also carry an `Integer` property to specify their range.

`@product_shorthand`
Rewrites the product of two fully symmetric or anti-symmetric tensors in a compact form by removing the contracting dummy indices.

```
▷ F_{m n p q}::Symmetric.
▷ G_{m n p q}::Symmetric.
▷ @product_shorthand![ F_{a b c d} G_{a b d f} ]{F}{G};
  F_{c} G_{f};
```

The two arguments denote the tensors to which the algorithm should apply. Expanding the product again in terms of the full tensors can be done with the algorithm `@expand_product_shorthand`.

`@expand_product_shorthand`
Reverse of the `@product_shorthand` algorithm. In addition to the two tensor names, it takes a number denoting the total number of indices.

```
▷ F_{m n p q}::Symmetric.
▷ G_{m n p q}::Symmetric.
▷ @product_shorthand![ F_{a b c d} G_{a b d f} ]{F}{G};
  F_{c} G_{f};
▷ @expand_product_shorthand!(%){F}{G}{4};
  F_{c a b d} G_{f a b d};
```

Note that writing a product as a shorthand and then expanding again may re-order the indices, as in the example above.

`@pintegrate`
Perform a partial integration. This requires an expression with an object carrying a `PartialDerivative` property. The command should have the name of the partial derivative as an argument:

```
▷ \partial{#}::PartialDerivative.
▷ \partial_{m}{A} B C D;
  @pintegrate!(%){\partial};
  (-1) \partial_{m}(B C D) A;
  @prodrule!(%):
  @distribute!(%);
  - \partial_{m}{B} C D A - B \partial_{m}{C} D A - B C \partial_{m}{D} A;
```

See the tutorials and the manual for `@take_match` and `@replace_match` for examples in which only particular terms in an expression are selected and partially integrated.

`@impose_bianchi`
Removes terms which are proportional to the (Garnir generalised) Bianchi identity. It removes all products for which a set of indices in Garnir hook form is contracted with an anti-symmetric set. Here is a simple example with a Riemann tensor,

```
▷ A^{m n p}::AntiSymmetric.
▷ R_{m n p q}::RiemannTensor.
▷ R_{m n p q} A^{m n p};
▷ @impose_bianchi!(%);
  0;
```

Here is a more complicated one using a `TableauSymmetry` property,

```
▷ \epsilon^{m n p q}::EpsilonTensor.
▷ B_{m n p}::TableauSymmetry(shape={2,1}, indices={0,1,2}).
▷ \epsilon^{m n p q} B_{m n p};
▷ @impose_bianchi!(%);
  0;
```

An alternative way to obtain the same result is to project each tensor using the generic `@young_project_tensor` algorithm, and then use `@distribute` followed by `@canonicalise`, but that generically takes much more time.

`@einsteinify`
In an expression containing dummy indices at the same position (i.e. either both subscripts or both superscripts), raise one of the indices.

```
▷ A_{m} A_{m};
▷ @einsteinify!(%);
  A^{m} A_{m};
```

If an additional argument is given to this command, it instead inserts "inverse metric" objects, with the name as indicated by the additional argument.

```
▷ {m,n}::Indices.
▷ A_{m} A_{m};
  @einsteinify!(%){\eta};
  A_{m} A_{n} \eta^{m n};
```

Note that the second form requires that there are enough dummy indices defined through the use of `Indices`.

`@combine`

Combine two consecutive objects with indexbrackets and consecutive contracted indices into one object with an indexbracket. An example with two contracted matrices:

```
▷ (\Gamma_r)_{\alpha\beta} (\Gamma_{s t u})_{\beta\gamma};
▷ @combine!(%);
  (\Gamma_r \Gamma_{s t u})_{\alpha\gamma};
```

An example with a matrix and a vector:

```
▷ (\Gamma_r)_{\alpha\beta} v_{\beta};
▷ @combine!(%);
  (\Gamma_{r} v)_{\alpha};
```

The inverse is done by `@expand`.

`@expand`

Write out products of matrices and vectors inside indexbrackets, inserting new dummy indices for the contraction. This requires that the objects inside the index bracket are properly declared to have `Matrix` or `ImplicitIndex` properties.

Here is an example with multiple matrices:

```
▷ {a,b,c,d,e}::Indices.
▷ {A,B,C,D}::Matrix.
▷ (A B C D)_{a b};
  @expand!(%);
  A_{a c} B_{c d} C_{d e} D_{e b};
```

Compare the above to the following example, in which one of the objects inside the bracket is no longer a matrix:

```
▷ {a,b,c,d,e}::Indices.
▷ {A,B,D}::Matrix.
▷ (A B C D)_{a b};
  @expand!(%);
  A_{a c} B_{c d} C D_{d b};
```

Finally, an example with matrices carrying additional labels, as well as a vector object:

```
▷ {\alpha,\beta}::Indices.
▷ \Gamma{#}::Matrix.
  v::ImplicitIndex.
  (\Gamma_{r} v)_{\alpha};
  @expand!(%);
  (\Gamma_{r})_{\alpha \beta} v_{\beta};
```

Note that in all cases, the indices on the indexbracket have to be part of a large enough set so that the dummy indices can be generated.

5.10 Output routines

The core of cadabra only knows how to output its internal tree format in infix notation, and by default displays the result of every input line in this way (labelled by the expression number and label, if any). For more complicated output, for instance to feed cadabra expressions to other programs, or to see the internal tree form, routines are available in the `output` module. Note that these modules do not transform the tree or generate new expressions; they *only* generate output.

Related but *different* functionality is present in the `convert` module, which contains transformation algorithms which transform the internal tree to other conventions (e.g. in order to convert a cadabra tree to a tree which can be understood by Maple or Mathematica).

properties:

`::LaTeXForm`(valid LaTeX expression)
Changes the way in which symbols are displayed in the graphical interface. Example:

```
▷ \del{#}::LaTeXForm("\partial").
▷ \del_{m}(A);
```

This prints $\partial_m(A)$; in the notebook, despite the fact that `\del` is not a LaTeX command.

If you use this property to make a symbol printable, make sure to declare it *before* any other properties are declared, otherwise the notebook will not know how to display the symbol and produce an error message.

Note that the property is attached to a pattern (`\del{#}` in this case) which matches the expression in which the replacement has to be made. If the pattern matches, the replacement will be done on the head symbol (`\del` in this case). A pattern `\del` without the argument wildcard `#` would only replace when `\del` occurs without any arguments (as in e.g. `\del + A`).

These settings have no effect in the command-line version.

algorithms:

`@depprint`
Display an expression together with all properties of the symbols that appear in this expression. This is a very useful command if you want to extract a result from a long calculation and write it in a separate file for further processing.

```
▷ \Gamma{#}::GammaMatrix.
▷ {m,n,p,q}::Indices.
  \Gamma_{m n p};
  @depprint(%);
```

`@print`
Construct output from strings and cadabra expressions. Here is an example:

```
▷ \Gamma{#}::GammaMatrix(metric=\delta).
▷ @print["The result is : " ~ @join[\Gamma^{a b}\Gamma_{c}] ~ ";"];
  The result is : (\Gamma^{a b}_{c} + 2 * \Gamma^{a} * \delta^{b}_{c});
```

As you can see in this example, expressions are joined together into one printable string by using the tilde character.

Note that `@print` completely overrides the normal output of expressions: there is no expression number for instance. Also note that all active nodes are (as usual) completely expanded before the print algorithm is called.

`@number_of_terms`
Displays the number of terms in a sum.

```
▷ A + B + C + D;
▷ @number_of_terms(%);
  4;
```

Note that this leaves the expression unmodified; if you want to use the number instead of just seeing it displayed, use `@length`.

`@proplist`
Show all properties and the patterns to which they are associated. Note that this includes all default properties for objects like $\backslash prod$.

```
▷ {A,B}::AntiCommuting.
▷ A_{m n}::Symmetric.
  @proplist;
  \sum{#}::CommutingAsSum
  \prod{#}::IndexInherit
  \prod{#}::Distributable
  ...
  {A, B}::AntiCommuting
  A_{m n}::Symmetric
```

A list of all available properties (not necessarily attached to any object) is instead obtained using `@properties`.

`@assert`
Checks whether the indicated expression equals zero. If not, the program will be terminated. This command is used e.g. in the test suite.

```
▷ 1 - B B**(-1);
▷ @collect_factors!(%);
  @collect_terms!(%);
```

```
@assert(%);
Assert check passed.
```

If the expression is not zero an error message will be produced, and the program will be terminated.

Furthermore, there are a few algorithms which are mainly useful for debugging purposes, as they display information about the internal representation of the expression tree. More information on the internal data structures can be found in section 6.2.

algorithms:

`@tree`
Display the internal tree form of a given expression. Can be used on a single expression or on an entire tree:

```
▷ A_{m n}:
▷ B_{m n}:
▷ @tree(1);
  (1): \history  (0)
    1:{\expression}  (1)
    2:  {A}  (2)
    3:    _{m}  (3)
    4:    _{n}  (3)
    5:  \asymimplicit  (2)
▷ @tree;
  (1): {\expression}  (1)
    1:  {A}  (2)
    2:    _{m}  (3)
    3:    _{n}  (3)
    4:  \asymimplicit  (2)

  (2): {\expression}  (1)
    1:  {B}  (2)
    2:    _{m}  (3)
    3:    _{n}  (3)
    4:  \asymimplicit  (2)
```

Most expressions lead to extremely lengthy output, so it is useful to redirect it to a file using output redirection.

`@indexlist`
Displays a list of all indices in an expression, together with their type (dummy or free).

5.11 General relativity

This module deals with special tensors relevant for general relativity. See in particular also the `canonicalise` algorithm in section 5.3.

properties:

`::Metric`**(***signature*=integer**)**
Labels the object as a symmetric tensor, and optionally gives it the indicated signature through the `signature` parameter.

```
▷ g_{m n}::Metric(signature=1).
▷ g_{m n} A^{n p};
▷ @eliminate_metric!(%);
  A_{m}^{p};
```

Objects declared as `Metrics` can be used to automatically raise or lower indices using the `@eliminate_metric` command.

`::InverseMetric`
This property is the partner of `Metric`. It makes the associated two-tensor symmetric and also indicates that it can be used to raise or lower indices.

```
▷ g^{m n}::InverseMetric.
▷ g_{m n}::Metric.
▷ g_{m q} g^{m n} A_{n p};
  @eliminate_metric!(%);
  A_{q p};
```

For more information, see `Metric`.

`::Vielbein`
Indicates that an object can be used to convert indices from one type to another, i.e. is a vielbein or tetrad. For more information, see `@eliminate_vielbein`.

`::InverseVielbein`
The partner of `Vielbein`. See the `@eliminate_vielbein` command for more information on typical usage patterns.

`::RiemannTensor`
Gives an object the symmetry properties of a Riemann tensor.

```
▷ R_{m n p q}::RiemannTensor;
▷ A^{m n p}::AntiSymmetric.
  A^{m n p} R_{m n p q};
  @impose_bianchi!(%);
  0;
```

Various other algorithms, such as `@canonicalise`, also take into account this property.

`::WeylTensor`
Gives an object the symmetry properties of a Weyl tensor, i.e. a Riemann tensor with an additional traceless condition. For more information, see `RiemannTensor`.

algorithms:

`@rewrite_indices`
Rewrite indices on an object by contracting it with a second object which contains indices of both the old and the new type (a vielbein, in other words, or a metric). A vielbein example is

```
▷ {m,n,p}::Indices(flat).
▷ {\mu,\nu,\rho}::Indices(curved).
▷ T_{m n p};
▷ @rewrite_indices!(%){ T_{\mu\nu\rho} }{ e_{\mu}^{n} };
  T_{\mu \nu \rho} e_{\mu}^{m} e_{\nu}^{n} e_{\rho}^{p};
```

If you want to raise or lower an index with a metric, this can also be done with as an index rewriting command, as the following example shows:

```
▷ {m,n,p,q,r,s}::Indices(curved, position=fixed).
▷ H_{m n p};
▷ @rewrite_indices!(%){ H^{m n p} }{ g_{m n} };
  H^{q r s} g_{m q} g_{n r} g_{p s};
```

As these examples show, the desired form of the tensor should be given as the first argument, and the conversion object (metric, vielbein) as the second object.

`@riemann_index_regroup`
Given a set (or multiple sets) of indices in which to anti-symmetrise, this routine determines whether the indicated Riemann tensor has an index distribution with two of the anti-symmetrised indices on two different pairs. If so, it performs the substitution

$$R_{m[p|\,n|q]} \to \frac{1}{2} R_{mnpq}\,,$$

which is valid by virtue of the Riccy cyclic identity $R_{m[npq]} = 0$. The example above translates to

```
▷ R_{m n p q}::RiemannTensor.
▷ A^{m n}::AntiSymmetric.
▷ R_{m p n q} A^{p q};
▷ @riemann_index_regroup!(%);
   1/2 R_{m n p q} A^{p q};
```

Note that this also works on Weyl tensors.

`@split_index`
Replace a sum by a sum-of-sums, abstractly. Concretely, replaces all index contractions of a given type by a sum of two terms, each with indices of a different type. Useful for Kaluza-Klein reductions and the like. An example makes this more clear:

```
▷ {M,N,P,Q,R}::Indices(full).
▷ {m,n,p,q,r}::Indices(space1).
▷ {a,b,c,d,e}::Indices(space2).
▷ A_{M p} B_{M p};
▷ @split_index(%){M,m,a};
  A_{m p} B_{m p} + A_{a p} B_{a p};
▷ @pop(%);
▷ @split_index(%){M,m,4};
  A_{m p} B_{m p} + A_{4 p} B_{4 p};
```

Note that the two index types into wich the original indices should be split can be either symbolic (as in the first case above) or numeric (as in the second case).

`@split_index`
Replace a sum by a sum-of-sums, abstractly. Concretely, replaces all index contractions of a given type by a sum of two terms, each with indices of a different type. Useful for Kaluza-Klein reductions and the like. An example makes this more clear:

```
▷ {M,N,P,Q,R}::Indices(full).
▷ {m,n,p,q,r}::Indices(space1).
▷ {a,b,c,d,e}::Indices(space2).
▷ A_{M p} B_{M p};
▷ @split_index(%){M,m,a};
  A_{m p} B_{m p} + A_{a p} B_{a p};
▷ @pop(%);
▷ @split_index(%){M,m,4};
  A_{m p} B_{m p} + A_{4 p} B_{4 p};
```

Note that the two index types into wich the original indices should be split can be either symbolic (as in the first case above) or numeric (as in the second case).

`@eliminate_vielbein`
Eliminates vielbein objects.

```
▷ { m, n, p }::Indices(flat).
▷ { \mu, \nu, \rho }::Indices(curved).
▷ e_{m \mu}::Vielbein.
▷ H_{m n p} e_{m \mu};
▷ @eliminate_vielbein!(%){H_{\mu\nu\rho}};
  H_{\mu n p};
```

Other elimination commands of a similar type are `@eliminate_kr` for Kronecker delta symbols and `@eliminate_metric` for metric and inverse metric objects.

`@eliminate_metric`
Eliminates metric and inverse metric objects.

```
▷ {m, n, p, q, r}::Indices(vector, position=fixed).
▷ {m, n, p, q, r}::Integer(0..9).
▷ g_{m n}::Metric.
▷ g^{m n}::InverseMetric.
▷ g_{m}^{n}::KroneckerDelta.
▷ g^{m}_{n}::KroneckerDelta.
▷ g_{m p} g^{p m};
▷ @eliminate_metric!(%);
  g^{p}_{p};
▷ @eliminate_kr!(%);
  10;
```

Other elimination commands of a similar type are `@eliminate_kr` for Kronecker delta symbols and `@eliminate_vielbein` for vielbeine.

5.12 Substitution

Substitution of expressions into other ones is a subtle issue. One problem, namely that of relabelling of dummy indices, is addressed by the core routines and has been discussed in section 5.7. Here we will discuss how cadabra handles the issue of pattern matching.

algorithms:

`@`

Given an expression number of label, this node replaces itself with a copy of this expression. As with any active node, this one takes care of dummy index relabelling automatically (see section 5.7).

`@rename`

(documentation no longer correct!) Renames objects and numbered objects, but only on a fixed level, i.e. does not do pattern matching at all. For instance,

```
▷ F_{m1 m2 m3};
  F_{m1 m2 m3}
▷ @rename!(%){m}{r};
  F_{r1 r2 r3}
```

It will not prevent you from introducing dummy pairs:

```
▷ F_{m n};
▷ @rename!(%){m}{n};
  F_{n n};
```

However, it will relabel already existing dummy pairs if you rename an open index in such a way that it would conflict with the dummy pair:

```
▷ {m,n,p,q,r,s}::Indices(vector).
▷ F_{m n} G_{n q};
▷ @rename!(%){m}{n};
  F_{n p} G_{p q};
```

`@substitute`

Generic substitution command. Takes a rule or a set of rules according to which an expression should be modified. If more than one rule is given, it tries each rule in turn, until the first working one is encountered, after which it continues with the next node.

```
▷ G_{mu nu rho} + F_{mu nu rho};
▷ @substitute!(%)( F_{mu nu rho} -> A_{mu nu} B_{rho} );
  G_{mu nu rho} + A_{mu nu} B_{rho};
```

```
▷ A_{mu nu} B_{nu rho} C_{rho sigma};
▷ @substitute!(%)( A_{m n} C_{p q} -> D_{m q} );
  D_{mu sigma} B_{nu rho};
```

This command takes full care of dummy index relabelling, as the following example shows:

```
▷ {m,n,q,d1,d2,d3,d4}::Indices(vector).
▷ a_{m} b_{n};
▷ @substitute!(%)( a_{q} -> c_{m n} d_{m n q} );
  c_{d1 d2} * d_{d1 d2 m} * b_{n};
```

By postfixing a name with a question mark, it becomes a pattern.

The substitution algorithm can do very complicated things; for more detailed information on substitution, see the manual.

`@vary`

Generic variation command. Takes a rule or a set of rules according to which the terms in a sum should be varied. In every term, apply the rule once to every factor.

```
▷ A B + A C;
▷ @vary(%)( A -> \epsilon D,
            B -> \epsilon C,
            C -> \epsilon A - \epsilon B );
  \epsilon D B + A \epsilon C + \epsilon D C
                     + A (\epsilon A - \epsilon B);
```

It also works when acting on powers, in which case it will use the product rule to expand them.

```
▷ A**3;
▷ @vary(%)( A -> \delta{A} );
  3 A**{2} \delta{A};
```

This algorithm is thus mostly intended for variational derivatives (subsequent partial integrations can be done using `@pintegrate`).

Note: In the examples above, the command is applied only at the top level (there is no exclamation mark used in the call of `@vary`). To understand why this is important, compare the following two examples. The first one works as expected,

```
▷ A B;
▷ @vary(%)( A -> a, B -> b);
  a * B + A * b;
```

In the second one, we add an exclamation mark,

```
▷ A B;
▷ @vary!(%)( A -> a, B -> b);
  0;
```

The reason why we now get a zero is that in the first step, the vary command acts in each of the individual factors, producing

```
a b;
```

It then acts once more at the level of the product. But now there are no uppercase symbols left anymore, and the variation produces zero.

`@take_match`
Select a subset of terms in a sum or list which match the given pattern.

```
▷ A + B D G + C D A;
▷ @take_match(%)( D Q?? );
  B D G + C D A;
```

In particular, note that the `Q??` is necessary to ensure that the pattern matches a product of `D` with something else. However, the algorithm has selected the entire term, not just the part matched by the pattern; compare the similar

```
▷ A + B D G + C D A;
▷ @substitute!(%)( D Q?? -> 1);
  A + G + A;
```

in which the replacement is done on the pattern, not on the entire term which contains the pattern.

This algorithm is particularly useful in combination with a copy operation on the expression. It allows one to take out certain terms from an expression, do manipulations on it, and then substitute it back using `@replace_match`.

The following example shows how this works by taking out the term which contains a χ factor, doing a transformation on the A_{mn} tensor in that term, and then substituting back using `@replace_match`.

```
▷ expr:= A_{m n} \chi B^{m}_{p} + \psi A_{n p};
▷ @take_match[@(%)]( \chi Q?? );
  A_{m n} \chi B^{m}_{p};
▷ @substitute!(%)( A_{m n} -> C_{m n} );
▷ @replace_match!(expr)( \chi Q?? -> @(%) );
  C_{m n} \chi B^{m}_{p} + \psi A_{n p};
```

The `@replace_match` pattern matching rules are identical to those in `@take_match`: a match always matches an entire term, not just a factor of it.

`@replace_match`

Replaces a subset of terms in a sum or list which match the given pattern. This is like a substitute, but always acting on entire terms.

```
▷ A + B D G + C D A;
▷ @replace_match!(%)( D Q?? -> 1);
  A + 1;
```

Note the difference with `@substitute`,

```
▷ A + B D G + C D A;
▷ @substitute!(%)( D Q?? -> 1);
  A + G + A;
```

See `@take_match` for further details on how to use this "select/modify/replace" mechanism.

5.13 Linear algebra

algorithms:

`@lsolve`
Solve a system of linear equations over integers. Example,

```
▷ { a0+2*a2 + a3= 3, -a0 - a2 + a3= - (8/3) , a3 = 3};
▷ @lsolve(%){a0,a2,a3};
  {a0 = 34/3, a2 = (-17/3), a3 = 3};
```

Underdetermined and inconsistent systems are handled as expected: either some coefficients are left unfixed or the system is returned and an error message is printed.

5.14 Gamma matrix algebra and fermions

This module deals with manipulations of the Clifford algebra, spinor representations of the Lorentz group and related issues. Cadabra follows the conventions of Sevrin's appendix 1.5, except for the fact that we call the time-like component zero and the product of all gamma matrices $\tilde{\Gamma}$.

properties:

`::Spinor`**(***dimension*=integer*, type=Weyl|Majorana|MajoranaWeyl, chirality=Positive|Negative)*
Declares an object to be a spinor, i.e. transforming in one of the spinor representations of the orthogonal or Lorentz group. The declaration should involve an indication of the dimension, as in the example below. It can optionally have type indicators (these should be `Majorana`, `Weyl` or `MajoranaWeyl`) and chirality indicators for Weyl spinors (`Positive` or `Negative`, indicating the eigenvalue with respect to the generalised γ_5 matrix). Here is an example:

```
▷ \psi::Spinor(dimension=11, type=Majorana).
```

This property is taken into account by various algorithms such as `@fierz` and `@spinorsort`.

`::DiracBar`
Declares an object to be the operator which applies the Dirac bar to a spinor, i.e. an operator which acts according to

$$\bar{\psi} = i\psi^\dagger\Gamma^0\,. \tag{5.18}$$

For more information see e.g. `@spinorsort` or `@fierz`.

`::GammaMatrix`**(***metric*=metric name*)*
A generalised generator of a Clifford algebra. With one vector index, it satisfies

$$\{\Gamma^m,\Gamma^n\} = 2\,\eta^{mn}\,. \tag{5.19}$$

The objects with more vector indices are defined as

$$\Gamma^{m_1\ldots m_n} = \Gamma^{[m_1}\cdots\Gamma^{m_n]}\,, \tag{5.20}$$

where the anti-symmetrisation includes a division by $n!$. If you intend to use the `@join` algorithm, you have to add a key/value pair `metric` to set the name of the tensor which acts as the unit element in the Clifford algebra.

`::GammaTraceless`
Declares a spinor object with a vector index to be zero when it is contracted with a gamma matrix.

`::SigmaMatrix`
These are the invariant tensors relating the $(\frac{1}{2},\frac{1}{2})$ to the vector representation of

SO(3,1). Cadabra uses the Wess & Bagger conventions, which means that the metric has signature $\eta = \mathrm{diag}(-1,1,1,1)$ and

$$(\sigma^\mu)_{\alpha\dot\beta} = (-\mathbb{1}, \vec\sigma)_{\alpha\dot\beta}\,, \quad (\bar\sigma^\mu)^{\dot\alpha\beta} = (-\mathbb{1}, -\vec\sigma)^{\dot\alpha\beta}\,. \tag{5.21}$$

When the objects carry two vector indices, they are understood to be

$$(\sigma^{mn})_\alpha{}^\beta \equiv \frac{1}{4}(\sigma^m\bar\sigma^n - \sigma^n\bar\sigma^m)_\alpha{}^\beta\,, \qquad (\bar\sigma^{mn})^{\dot\alpha}{}_{\dot\beta} \equiv \frac{1}{4}(\bar\sigma^m\sigma^n - \bar\sigma^n\sigma^m)^{\dot\alpha}{}_{\dot\beta}\,. \tag{5.22}$$

See below for algorithms dealing with the conversion from indexed to index-free notation.

`::SigmaBarMatrix`
See `SigmaMatrix` for details.

algorithms:

`@join`
Join two fully anti-symmetrised gamma matrix products according to the expression

$$\Gamma^{b_1\ldots b_n}\Gamma_{a_1\ldots a_m} = \sum_{p=0}^{\min(n,m)} \frac{n!m!}{(n-p)!(m-p)!p!}\Gamma^{[b_1\ldots b_{n-p}}{}_{[a_{p+1}\ldots a_m}\eta^{b_{n-p+1}\ldots b_n]}{}_{a_1\ldots a_{m-p}]}\,. \tag{5.23}$$

This is the opposite of `@gammasplit`.

Without further arguments, the anti-symmetrisations will be left implicit. The argument "`expand`" instead performs the sum over all anti-symmetrisations, which may lead to an enormous number of terms if the number of indices on the gamma matrices is large. Compare

```
▷ \Gamma{#}::GammaMatrix(metric=g).
▷ \Gamma_{m n} \Gamma_{p};
  @join!(%);
  \Gamma_{m n p} + 2 \Gamma_{m} g_{n p};
```

with

```
▷ \Gamma{#}::GammaMatrix(metric=g).
▷ \Gamma_{m n} \Gamma_{p};
  @join!(%){expand};
  \Gamma_{m n p} + \Gamma_{m} g_{n p} - \Gamma_{n} g_{m p};
```

Note that the gamma matrices need to have a metric associated to them in order for this algorithm to work.

In order to reduce the number somewhat, one can instruct the algorithm to make use of generalised Kronecker delta symbols in the result; these symbols are defined as

$$\delta^{r_1}{}_{s_1}{}^{r_2}{}_{s_2}\ldots{}^{r_n}{}_{s_n} = \delta^{[r_1}{}_{s_1}\delta^{r_2}{}_{s_2}\ldots{}^{r_n]}{}_{s_n}\,. \tag{5.24}$$

Anti-symmetrisation is implied in the set of even-numbered indices. The use of these symbols is triggered by the "`gendelta`" option,

```
▷ {m,n,p,q}::Indices(position=fixed).
▷ \Gamma{#}::GammaMatrix(metric=\delta).
  \Gamma_{m n} \Gamma^{p q};
  @join!(%){expand}{gendelta};
   \Gamma_{m n}^{p q} + \Gamma_{m}^{q} \delta_{n}^{p}
      - \Gamma_{m}^{p} \delta_{n}^{q} - \Gamma_{n}^{q} \delta_{m}^{p}
      + \Gamma_{n}^{p} \delta_{m}^{q} + 2 \delta_{n}^{p}_{m}^{q};
```

Finally, to select only a single term (for a given p) in this expansion, give the join an argument with the value of p.

```
▷ \Gamma{#}::GammaMatrix(metric=g).
▷ \Gamma_{m n} \Gamma_{p};
  @join!(%){expand}{3};
  \Gamma_{m n p};
```

This option can also be combined with `gendelta` if required.

`@gammasplit`

The opposite of `@join`: splits off a gamma matrix from a totally anti-symmetrised product, e.g.

$$\Gamma^{mnp} = \Gamma^{mn}\Gamma^{p} - 2\,\Gamma^{[m}\eta^{n]}{}_{p}\,. \tag{5.25}$$

In code this reads

```
▷ \Gamma{#}::GammaMatrix(metric=\eta).
▷ \Gamma^{m n p};
  @gammasplit(%);
   \Gamma^{m n} \Gamma^{p} - \Gamma^{m} \eta^{n p}
                                  + \Gamma^{n} \eta^{m p};
```

By default it splits of the one from the back, like in the example above, but with argument `front` it will split from the front:

```
▷ \Gamma{#}::GammaMatrix(metric=\eta).
▷ \Gamma^{m n p};
  @gammasplit(%){front};
  \Gamma^{m} \Gamma^{n p} - \Gamma^{p} \eta^{m n}
                                  + \Gamma^{n} \eta^{m p};
```

`@rewrite_diracbar`

Rewrite the Dirac conjugate of a product of spinors and gamma matrices as a product of Dirac and hermitean conjugates. This uses

$$\bar{\psi} = i\psi^{\dagger}\Gamma^{0}\,, \tag{5.26}$$

together with

$$\Gamma_m^\dagger = \Gamma_0 \Gamma_m \Gamma_0 \,. \tag{5.27}$$

For example,

```
▷ \bar{#}::DiracBar.
▷ \psi::Spinor(dimension=10).
▷ \Gamma{#}::GammaMatrix.
▷ \bar{\Gamma^{m n p} \psi};
▷ @rewrite_diracbar!(%);
  \bar{\psi} \Gamma^{m n p};
```

`@projweyl`

Projects an expression onto Weyl spinors of positive chirality (this algorithm only works in even dimensions). On such a subspace, we have

$$\Gamma^{r_1\cdots r_d}\Big|_{\text{Weyl}} = \frac{1}{\sqrt{-g}}\epsilon^{r_1\cdots r_d}\,, \quad \epsilon^{0\cdots(d-1)} = +1\,, \tag{5.28}$$

and therefore all gamma matrices with more than $d/2$ indices can be converted to their "dual" gamma matrices. By repeated contraction of (5.28) with gamma matrices on the left one deduces that

$$\Gamma^{r_1\cdots r_n}\Big|_{\text{Weyl}} = \frac{1}{\sqrt{-g}}\frac{(-1)^{\frac{1}{2}n(n+1)+1}}{(d-n)!}\Gamma_{s_1\cdots s_{d-n}}\Big|_{\text{Weyl}}\epsilon^{s_1\cdots s_{d-n}r_1\cdots r_n}\,. \tag{5.29}$$

Here is an example:

```
▷ {m,n,p,q,r,s,t}::Indices.
▷ {m,n,p,q,r,s,t}::Integer(0..5).
  \Gamma{#}::GammaMatrix.
  \Gamma_{m n p q};
  @projweyl!(%);
```

`@spinorsort`

Sorts Majorana spinor bilinears using the Majorana flip property, which for anti-commuting spinors takes the form

$$\bar\psi_1\Gamma_{r_1\cdots r_n}\psi_2 = \alpha\beta^n(-)^{\frac{1}{2}n(n-1)}\,\bar\psi_1\Gamma_{r_1\cdots r_n}\psi_2\,. \tag{5.30}$$

Here α and β determine the properties of the charge conjugation matrix,

$$\mathcal{C}^T = \alpha\mathcal{C}\,, \quad \mathcal{C}\Gamma_r\mathcal{C}^{-1} = \beta\Gamma_r^T\,. \tag{5.31}$$

Here is an example.

```
▷ {\chi, \psi, \psi_{m}}::Spinor(dimension=10, type=MajoranaWeyl).
▷ {\chi, \psi, \psi_{m}}::AntiCommuting.
▷ \bar{#}::DiracBar.
▷ \Gamma{#}::GammaMatrix.
▷ {\psi_{m}, \psi, \chi}::SortOrder.
▷ \bar{\chi} \Gamma_{m n} \psi;
▷ @spinorsort!(%);
  (-1) \bar{\psi} \Gamma_{m n} \chi;
```

5.15 Conversion from/to other formats

Cadabra's syntax is a superset of the Maple and Mathematica formats, in the sense that the internal tree representation of cadabra can store input for both of these systems. However, this may not always be the most convenient way of storage, as cadabra often has more compact or expressive ways to store tensors. The `convert` module provides conversion algorithms from and to these two formats.

algorithms:

`@run`
Run an external program on the input expression, replacing it with the standard output of the program. The expression is the first command line argument of the program to be called.

Suppose we have a simple script `test.sh` containing

```
#!/bin/sh
echo "$1" | sed -e 's/A/B/g' -e '/;/q'
```

This script replaces 'A' with 'B' in the first command line argument, until it encounters a semi-colon. We can then call this script by using e.g.

```
▷ A B C A D E;
▷ @run(%){"./test.sh"};
  B B C B D E;
```

The external program should make sure that it produces valid cadabra input.

`@maxima`
Feed an expression through the Maxima computer algebra system. This allows you to e.g. do trigonometric simplification or other 'scalar computer algebra' for which cadabra does not have any built-in support.

```
▷ trigsimp( sin(x)**2 + cos(x)**2 ):
▷ @maxima(%);
  1;
▷ integrate( \sin(x)**2, x ):
▷ @maxima(%);
  (1/2 * x - 1/4 * \sin(2 * x));
```

Note that reserved names, like the `\sin` above, get converted to Maxima notation automatically (and they get translated again when the result is read back into cadabra).

This command is preliminary and not yet fully functional.

`@maple`
Feed an expression through the Maple computer algebra system. This allows you to e.g. do trigonometric simplification or other 'scalar computer algebra' for which cadabra does not have any built-in support.

```
▷ simplify( sin(x)**2 + cos(x)**2 ):
▷ @maple(%);
  1;
▷ integrate( \sin(x)**2, x ):
▷ @maple(%);
  - 1/2 \sin(x) \cos(x) + 1/2 x;
```

Note that reserved names, like the `\sin` above, get converted to Maple notation automatically (and they get translated again when the result is read back into cadabra).

This command is preliminary and not yet fully functional.

`@from_math`**[expression]**
Reads mathematica input format as used by GAMMA [7]. Fully anti-symmetric tensors are denoted with

```
Tensor[name,{indices}]
```

Tensors with property "weyltensor" are printed as

```
▷ W_{r1 r2 r3 r4}
  Weyl[{r1,r2}, {r3,r4}]
```

(warning, this removes the tensor name).

`@to_math`**[expression]**
Converts the mathematica expression to a cadabra one. See above for details about the conversion process.

Core functionality

6.1 Source file description

The files in the `src` directory build up the core of the program and are described below. Algorithm-specific code is in the `src/modules` subdirectory and a description of those files can be found in section 5. The files below should not be changed when adding new modules.

`main.cc`
Startup routines; initialises the I/O streams, sets up the signal handlers for control-C and window resize signals and starts the main event loop.

`manipulator.cc`, `manipulator.hh`
Contains the object that processes the input line by line, activates the parser on it, and scans the tree for active nodes. A few very low-level commands (see section 5.1) are handled in this class, while the other ones are dispatched to external modules.

`preprocessor.cc`, `preprocessor.hh`
The code for conversion of human editable infix input to the tree-form handled by the other parts of the program. This is a text-to-text filter forming the first pass of the parser. All functionality can be tested using `test_preprocessor.cc`.

`storage.cc`, `storage.hh`
The node and tree classes for storage of the tree form of expressions. These are the classes on which actual symbolic manipulation takes place. See also the separate `tree.hh` class.

`parser.cc`, `parser.hh`
The class which contains the parsing algorithms that turns output of the preprocessor class into a tree form.

`algorithm.cc`, `algorithm.hh`
The base class for all the classes defined in the modules subdirectory. Plus the implementation of scanning for command arguments and the necessary logic to create the undo information, apply an algorithm until it no longer changes the expression and other things which are independent of the particular command.

`combinatorics.hh`
General routines for the generation of combinations and permutations of elements in a vector. Consists of a single class `combinations<...>` templated over the type of the objects to be permuted. This class acts as a container holding both the original vector (in the `storage` member), the permutation patterns (in the `sublengths` vector, more on this later) as well as the resulting permuted vectors (accessible through `operator[]`; the total number of combinations is given by `size()`).

6.2 Tree representation of expressions

Expressions are internally stored in a tree container class, based on the `tree.hh` library [8]. The present section describes the tree format in some more detail.

Let us start with a couple of remarks on the motivation which has led to the current implementation. An important requirement for the storage format is that symbols should not have a fixed meaning, but rather be like variables in a computer program: they can be declared to represent objects of a certain type. Similarly, the user should be free to use arbitrary bracket types and sub- and superscript indicators (these should in addition be kept during the algebraic manipulations). The first issue is resolved with the property system, but the second one needs support directly in the storage tree.

Secondly, storage should be such that a trivial (often occurring) extensions from one object to another should not lead to blowup. For instance, the difference between `A_\mu` and `A_{\mu \nu}` should preferably not introduce a new layer between the `A` node and the `\mu` node. This is achieved by adding the bracket type to the *child* node rather than the parent (in the example above, the curly brackets are part of the `\mu` and `\nu` nodes). This applies also to e.g. sums: in `(A+B)` (which is converted by the preprocessor to `\sum(A)(B)`), the round brackets are part of the child nodes of the `\+` node.[7] Similarly, extending `q` to `3q` should not introduce new intermediate layers to group the two symbols together. This has been achieved by giving each node a multiplier field (this holds true even for products, i.e. `3(a+b)` is represented as a sum node with multiplier 3 (in addition, such numerical factors are always commuted to the front of the expression).[8]

It should also be possible to make expressions arbitrarily deeply nested; one should be able to write `a+b` as well as `\sin(a+b)`. This is in contrast to programs such as `Abra` and `FORM` which store their input as sums of products of factors, i.e. without further nesting.

Finally, tt should be posssible to store the history of an expression, in order to implement unlimited undo.

The data stored at each tree node is further explained in section 7.4, see in particular the excerpt from the `storage.hh` file in figure 2. The structure of the actual tree is illustrated in figure 1. Each expression starts with a `history` node. The label of the expression is given in a subnode `label`. All remaining child nodes of the `history` nodes represent the history of the expression. The most often encountered node here is the `expression` node, which contains the actual expression. However, there are other nodes as well, for instance `asymimplicit` which contains information about the symmetry structure of an expression (this feature is not yet enabled).

6.3 Nested sums, products and pure numbers

Nested sum and product nodes require some explanation, as the way in which cadabra handles and stores these determines how brackets can appear in sums and products. As a general rule, sums within sums which are such that all child nodes have the same, non-visible bracket are not allowed, and automatically converted to a single sum. In other words

```
(\sum{a}{b}+\sum{c}{d})
```

[7] Subsequent child nodes with the same bracket type associated to them are taken to be part of the same group; the input `A_{\mu}_{\nu}` will therefore be printed as `A_{\mu \nu}`. Similarly, there is no way to represent $(a)+(b)$ since the internal storage format automatically turns this into $(a+b)$.

[8] The multiplier field is a rational, and is intended to stay that way. It is not a good idea to make it complex, otherwise we will end up with octonionic multipliers at some stage.

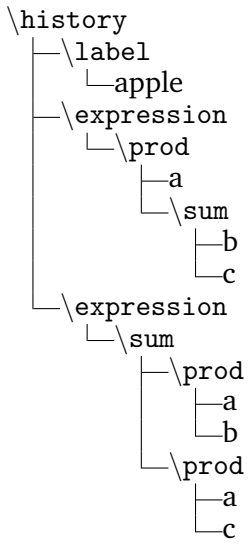


Figure 1: Example history tree for a single expression. Here, the program started with the user entering the expression $a(b+c)$. This expression was then manipulated by applying the `@distribute` algorithm. The nodes at the top of the expression tree are all of the `history` type.

is not allowed, since it would print as `(a+b+c+d)` which is indistinguishable from

```
(\sum{a}{b}{c}{d}) .
```

The first expression is therefore internally always converted to the second one before it is stored. A similar situation occurs with products, where

```
\prod{a}{\prod{b}{c}}
```

is not allowed and "flattened" to the simpler

```
\prod{a}{b}{c} .
```

The general rule is that nested sums and products are converted to flattened form if these two forms are indistinguishable in the standard output format. Nested sums and products can occur, but they have to have different or non-empty bracket types for their children. An example is

```
\prod{a}{\prod(b)(c)}
```

which prints as `a*(b*c)`. This is not automatically converted to `a*b*c`. Note that this form is *not* represented as

```
\prod{a}(b)(c)    (wrong)
```

This is a consequence of the second rule: all child nodes of product or sum nodes must have the same bracket type. In summary, automatic flattening of nested sum or product trees do not automatically occur over bracket boundaries. You can then write and keep $(\sin(x)+\cos(x))+(\sin(x)+3\cos(x))$, which will be stored as

```
\sum{\sum(sin(x))(cos(x))}{\sum(sin(x))(cos(x))}
```

In order to remove brackets, one uses the `@sumflatten` and `@prodflatten` commands, see section 5.3.

Products and sums with sub- or superscripts which apply to one or more of the child nodes are internally represented using `\indexbracket` nodes. That is, the input

```
q (a+b)_\mu
```

is represented as

```
\prod{q}{\indexbracket{\sum(a)(b)}_{\mu}}
```

(note the way in which the bracket types are inherited). In fact, such indexbracket constructions apply for any content inside the brackets; in the example above, an indexbracket would be generated even without the `\sum` node: the input

```
q (\Gamma^r)_{a b}
```

is stored as

```
\prod{q}{\indexbracket(\Gamma^r)_{a b}}
```

The result of the indexbracket way of storing expressions is that "index free" manipulations apply directly inside the indexbracket argument and no special algorithms are needed. Input will automatically be converted to this form. Getting rid of indexbrackets around single symbols is done using `@remove_indexbracket`.

Another issue is the storage of numbers. Since pure numerical factors are only allowed to appear in the multiplier field of a node, a pure number is stored as "1" with its multiplier equal to the number. Product nodes with two arguments of which one is a number are not allowed and will be converted to a single node with multiplier field, automatically.

In order to eliminate ambiguities concerning multipliers in sum and product nodes, the following rules should be obeyed: sum nodes always have unit multiplier (i.e. nontrivial multipliers are associated to the children) and children of product nodes should also always have unit multiplier (i.e. the multipliers of all the children should in this case be collected in the prod node itself).

6.4 External scalar engine interface

Cadabra is a tensor computer algebra system, without much functionality for simplification or manipulation of scalar expressions. Other systems exist which handle this much better, and it is a waste to duplicate those efforts. Therefore, the `algorithm` class contains functionality to isolate scalar factors from a tensorial expression, manipulate them using an external scalar CAS, and re-insert them into the internal cadabra tree.

Scalar expressions are defined to be sums or (subsets of factors of) products which contain no objects with subscript or superscript children (i.e. contain no tensors).

Exporting a scalar expression to an external CAS requires a rewriting step before and after the export. At present only Maxima's format is supported.

Module implementation guide

7.1 Storage of numbers and strings

There is a special memory area for the storage of rational numbers (with arbitrary-length integers for the numerator and denominator) as well as for the storage of strings.

7.2 Accessing indices

Indices are stored as ordinary child nodes. However, there are various ways to iterate over all indices in a factor, which are more useful than a simple iteration over all child nodes.

7.3 Object properties

The properties attached to a node in the tree can be queried by using a lookup method in the `properties` namespace. Object properties are child classes of the `property` class, and contain the property information in a parsed form. The following code snippet illustrates how properties are dealt with:

```
AntiSymmetry *s=properties::get<AntiSymmetry>(it)
if(s) {
   // do something with the property info
   }
```

Here we used a property called `AntiSymmetry`, which is declared in the the module header `modules/algebra.hh` (see the other header files of the various modules for information about the available property types). If the node pointed to by `it` has this property associated to it, the pointer `s` will be set to point to the associated property node. **[KP: Better to use a property example where the property node contains some additional information, which can then be used.]**

It is also possible to do this search in the other direction, that is, to find an object with a given property.

```
properties::property_map_t::iterator
               it=properties::get_pattern<GammaMatrix>();
if(it!=properties::props.end()) {
   // Now *(it->first) contains the name of a GammaMatrix object.
   }
```

[KP: Instead of checking these things, it is better to throw exceptions, since that will make the code more readable and also the error handling more uniform (it can go in one place instead of being repeated).]

Sums and products inherit, in a certain way, properties of the terms and factors from which they are built up. This is true also for other functions. If you want to know whether a given node has such *inherited properties*, a simple call to `get<...>` is not sufficient. Instead, use the following construction:

```
Matrix *m=properties::get_composite<Matrix>(it)
if(m) {
   // do something with the property info
   }
```

These inheritance rules for composite objects are at present hard-coded but will be user-defineable at a later stage.

In addition to querying for object properties, some algorithms also act on objects which encode their property in a special reserved node name. As an example, the `@prodrule` acts on objects which carry a `Derivative` property, but it also acts on `@cdb_Derivative` objects. This allows e.g. the `@vary` algorithm to wrap a derivative-like object around powers and have `@prodrule` expand that

7.4 Writing a new algorithm module

All functionality of cadabra that goes beyond storage of the expression trees is located in separate modules, stored in the `src/modules` subdirectory. New functionality can easily be added by writing new modules. All algorithm objects derive from the `algorithm` object defined in `src/algorithm.hh`, and only a couple of virtual members have to be implemented.

The class interface for the algorithm object is shown in figure 3. The members which any class derived from `algorithm` should implement are described below.

`constructor(exptree&, iterator)`
: Algorithm objects get initialised with a reference `tr` to the expression tree on which they act, together with an iterator `this_command` pointing to the associated command node. By looking at the child nodes of the latter, the arguments of the command can be obtained. The simplest way to do this is to use the sibling iterators returned by `args_begin()` and `args_end()`; these automatically skip the argument which refers to the equation number (i.e. they skip the `(5)` argument in `@command(5){arg1}{arg2}`). The constructor is a good place to parse the command arguments, since that will in general only be required once. Do not do anything in the constructor that has to be done on every invocation of the same command on different expressions.

  If you discover that the arguments passed to an algorithm are not legal, you have to throw an `algorithm::constructor_error` exception.

`description()`
: Should display an explanation of the functionality of the algorithm on the `txtout` stream. Do not add `std::endl` newlines by hand, these will be inserted automatically.

`bool can_apply(iterator)`
`bool can_apply(sibling_iterator, sibling_iterator)`
: Determines whether the algorithm has a chance of acting on the given expression. This is an *estimate* but is required to return `true` if there is a possibility that the algorithm would yield a change.

```
class str_node {
   public:
      ...
};
```

Figure 2: The node class.

```
class algorithm {
  public:
    typedef exptree::iterator         iterator;
    typedef exptree::sibling_iterator sibling_iterator;

    algorithm(exptree&, iterator&);

    virtual void     description() const=0;
    virtual bool     can_apply(iterator&);
    virtual bool     can_apply(sibling_iterator&, sibling_iterator&);
    virtual result_t apply(iterator&);
    virtual result_t apply(sibling_iterator&, sibling_iterator&);

    sibling_iterator args_begin() const;
    sibling_iterator args_end() const;
    unsigned int     number_of_args() const;

    exptree&  tr;
    iterator this_command;
};
```

Figure 3: The relevant members of the class interface for the `algorithm` object. The virtual members and the constructor have to be implemented in new algorithm objects; see the main text for an explanation of the two versions of `can_apply` and `apply`.

```
class exptree {
  public:
    sibling_iterator arg(iterator, unsigned int) const;
    unsigned int     arg_size(sibling_iterator) const;
    multiplier_t     arg_to_num(sibling_iterator, unsigned int) const;
};
```

Figure 4: The relevant members of the class interface for the `exptree` object.

It takes two sibling_iterators as arguments, which indicate a node or a range of nodes on which the algorithm is supposed to act.

Since `can_apply` is guaranteed to be called before `apply`, it is allowed for the `can_apply` member to store results for later use in `apply`. However, multiple calls to can be made to this combo, so any data stored should be erased upon each call to `can_apply`.

`apply_result apply(iterator&)`
`apply_result apply(sibling_iterator&, sibling_iterator&)`

This is where the actual algorithm is applied. It is allowed to modify the iterator if necessary. No elements in the tree which sit above the entry point should ever be changed.

Before exiting this function, the member variable `expression_modified` has to be set according to whether or not the apply call made any changes to the expression.

If the algorithm can take a long time to complete and you want to give the user the option of interrupting it with control-C, you can check (at points where the algorithm can be interrupted) the global variable `interrupted`. If it is set, you should immediately throw an exception of type `algorithm_interrupted`, preferably with a short string describing the location or algorithm at which the interrupt occurred. The exception will be handled at the top level in the `manipulator.cc` code, where the current expression will be removed.

Algorithms can write progress information to the stream `txtout` and debug information (which will go into a separate file on disk) to `debugout`. These are global objects. Do not use the C++ streams `std::cout` and `std::cerr` for this purpose.

An algorithm is *never* allowed to modify the tree above the node or range of sibling nodes which were passed to the `apply` call. In other words, siblings and parent nodes should never be touched, and the nodes should not be removed from the tree directly. If you want to remove nodes, this can be done by setting their multiplier field to zero; the core routines will take care of actually removing the nodes from the tree. It is allowed to inspect the tree above the given nodes, but this is discouraged.

Upon any exit from an algorithm module, the tree is required to be in a valid state. In many cases, it is useful to clean up modifications to the tree by calling the utility function `cleanup_expression` (see section 7.9 for more information).

7.5 Adding the module to the system

Once you have written the `.hh` and `.cc` files associated to your new algorithm, it has to be added to the build process and the core program. First, add the header to the master header file `src/modules/modules.hh` file; this one looks like

```
src/modules/modules.hh:

#include "algebra.hh"
#include "pertstring.hh"
#include "select.hh"
...
```

You also have to add it to the `src/Makefile.in`. Near the top of this file you find a line looking somewhat like

```
src/Makefile.in:

...
MOBJS=modules/algebra.o modules/pertstring.o modules/convert.o \
      modules/field_theory.o modules/select.o modules/dummies.o \
      modules/properties.o modules/relativity.o modules/substitute.o
...
```

and this is where your module has to be added. Finally, you have to make the algorithm visible to the core program. This is done in the `src/manipulator.cc` file. In the constructor of the `manipulator` class (defined somewhere near the top), you see a map of names to algorithm classes. A small piece is listed below:

```
src/manipulator.cc:

...
// pertstring
algorithms["@aticksen"]       =&create<aticksen>;
algorithms["@riemannid"]      =&create<riemannid>;

// algebra
algorithms["@distribute"]     =&create<distribute>;
 ...
```

Add your algorithm here; it does not have to have the same name as the object class name. Once this is all done, rerun configure and make.

7.6 Adding new properties

[KP: Discuss the various member functions of property, in particular the way in which parse is supposed to behave, and the way in which properties should be registered.] Since algorithms rely crucially on a *fixed* set of properties which they provide themselves, we store these things in a C++ way, rather than using strings. Another motivation: we do not want to parse (and check) these property trees every time they are accessed.

Properties typically carry key/value pairs which further specify the characteristics of the property. When the property object is created, the core calls the `parse()` method of the property, in which you are responsible for scanning through the list of key/value pairs. This is rather simple: just call the `find()` member of the `keyval` argument,

```
keyval_t::iterator ki=keyvals.find("dimension");
if(ki!=keyvals.end()) {
    // now ki->second is an iterator to an expression subtree
    }
```

```
class property\_base{
  public:
    virtual bool        parse(exptree&, iterator pat, iterator prop);
    virtual std::string name() const=0;
    virtual void        display(std::ostream&) const;
};

class property      : public property\_base {};
class list\_property : public property\_base {};

class labelled\_property : public property {
  public:
    std::string label;
};
```

Figure 5: The relevant members of the class interface for the `property_base` object and the derived objects.

You are supposed to remove any handled keyval pair from the list by using the `erase()` member, so that the system can keep track of unhandled pairs. Do not forget to call the `parse()` method of all parent classes at the end.

Once this is all done, register the property with the system by adding an appropriate call to the module's `register_properties()` method, found at the top of each module.

7.7 Throwing errors

If, at any stage, an inconsistent expression is encountered, the safe way to return to the top level of the manipulator is to throw the exception class `consistency_error` (defined in `algorithm.hh`).

7.8 Manipulating the expression tree

The `exptree` class contains a number of tree access routines which enhance the functionality of the `tree.hh` library.

`index_iterator`

`index_map_t`

Acting on products or single terms

use `is_single_term` inside `can_apply` to test for this case. Then in `apply` call `prod_wrap_single_term` (which will be harmless for proper products). At the very end of `apply`, call `prod_unwrap_single_term` to remove the product again.

7.9 Utility functions

There are also various useful helper functions in the `exptree` class, mainly dealing with the conversion of expression numbers or labels to iterators that point to the actual expression in the tree.

`cleanup_expression(exptree&)`
`cleanup_expression(exptree&, iterator)`
Converts an expression (or part of it below the indicated node) to a valid form. That is, it calls various simplification algorithms on the expression, among which `sumflatten` and `prodcollectnum`.

`cleanup_nests`
When called on a product inside a product, or a sum inside a sum, and the bracket types of both node and parent are the same, flatten the tree. Adjusts the iterator to point to the product or sum node which remains.

`equation_by_number_or_name`
Given an iterator to a node that represents an expression number or expression label, this member returns an iterator that points to the actual node (to be precise: to the `\history` node of the expression).

`arg_size(sibling_iterator)`
`arg(iterator, unsigned int)`
Many algorithms require one *or more* objects as arguments. These are usually passed as lists, and therefore end up being of two forms: either they are single objects or they are `comma` objects representing a list. In order to avoid having to check for these two cases, one can use these two member functions.

The iterator arguments of these two member functions should point to the child which is either a single argument or a `\comma` node.

`pushup_multiplier`
Can be called on any node. If the node has a non-unit multiplier and the storage conventions exclude this, the algorithm will push the multiplier up the tree.

7.10 Writing a new output module

[KP: Structure is different now: one inherits from `output_exptree` and then has printing stuff at ones disposal. Still would be nice to be able to change the printing objects for a given node type, i.e. modify the map of node names to printing objects.]

[KP: Also do something intelligent with the Mathematica output flags].

7.11 Namespaces and global objects

The following objects are global:

`modglue::opipe txtout`
The normal output stream, to which you should communicate with the user.

`std::ofstream debugout`
The debug output stream, which normally writes to a file, and should only be used to write output which is useful for debugging purposes.

`nset_t name_set`
The global collection of all strings. Nodes in the expression tree contain an iterator into this map (see in particular figure 2).

`rset_t rat_set`
The global collection of all rational numbers. Nodes in the expression tree contain an iterator into this map (see in particular figure 2).

`bool interrupted`
A global flag which indicates that the user has requested the computation to be interrupted. See section 7.4 for more information on this variable.

`stopwatch globaltime`
The wall time since program start.

`unsigned int size_x, size_y`
The size of the current output window.

Index

References

[1] J. Martín-García, "xPerm and xAct", http://metric.iem.csic.es/Martin-Garcia/xAct/index.html.

[2] A. Cohen, M. van Leeuwen, and B. Lisser, "LiE v. 2.2", 1998, http://wwwmathlabo.univ-poitiers.fr/~maavl/LiE/.

[3] S. A. Fulling, R. C. King, B. G. Wybourne, and C. J. Cummins, "Normal forms for tensor polynomials. 1: The Riemann tensor", *Class. Quant. Grav.* **9** (1992) 1151.

[4] F. De Jonghe, K. Peeters, and K. Sfetsos, "Killing-Yano supersymmetry in string theory", *Class. Quant. Grav.* **14** (1997) 35–46, hep-th/9607203.

[5] J. van der Hoeven, "GNU TeXmacs: A free, structured, wysiwyg and technical text editor", in "Le document au XXI-ième siècle", D. Flipo, ed., vol. 39–40, pp. 39–50. Metz, 14–17 mai 2001. Actes du congrès GUTenberg. http://www.texmacs.org.

[6] J. A. M. Vermaseren, "New features of FORM", math-ph/0010025.

[7] U. Gran, "GAMMA: A Mathematica package for performing gamma-matrix algebra and Fierz transformations in arbitrary dimensions", hep-th/0105086, http://fy.chalmers.se/~gran/GAMMA/.

[8] K. Peeters, "Tree.hh", http://tree.phi-sci.com/.