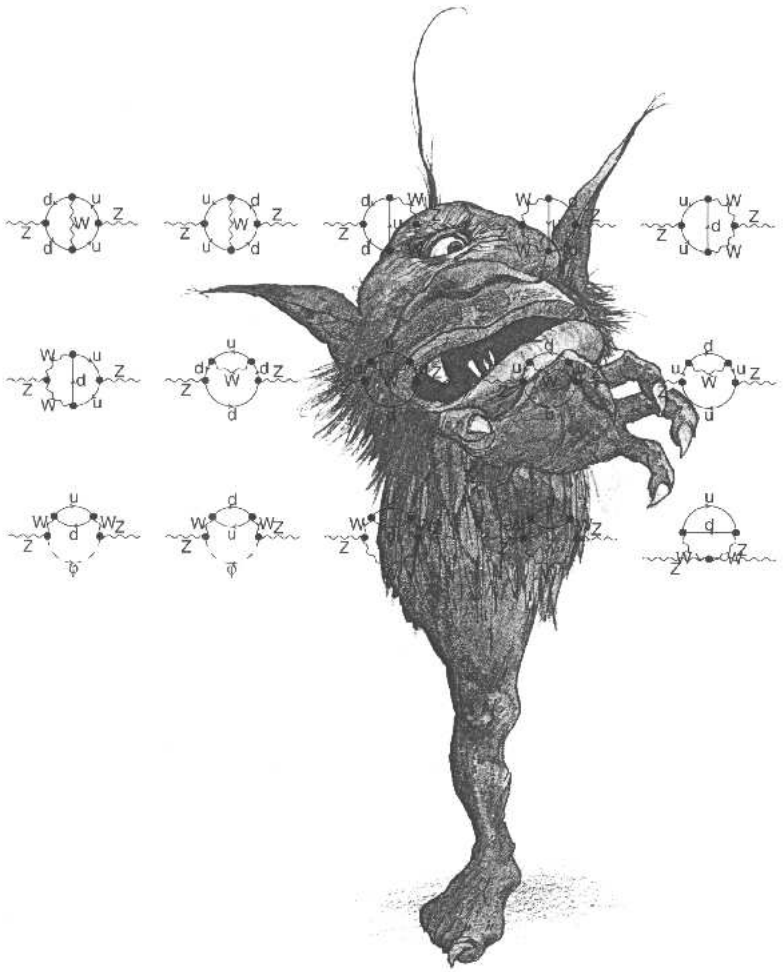


FeynArts 2.2 User's Guide

August 3, 1998 Thomas Hahn

The dreadful legal stuff: *FeynArts* is free software, but is not in the public domain. Instead it is covered by the GNU library general public license. In plain English this means:

1) We don't promise that this software works. (But if you find any bugs, please let us know!)

2) You can use this software for whatever you want. You don't have to pay us.

3) You may not pretend that you wrote this software. If you use it in a program, you must acknowledge somewhere in your documentation that you've used our code.

If you're a lawyer, you will rejoice at the exact wording of the license at `http://www.fsf.org/copyleft/lgpl.html`.

FeynArts is available from `ftp://ftp.physik.uni-wuerzburg.de/pub/hep/index.html`. If you make this software available to others please also provide them with this manual. To join the *FeynArts* mailing list where updates will be announced, send a mail (any text) to `hahn-feynarts-subscribe@particle.physik.uni-karlsruhe.de`.

If you find any bugs, or want to make suggestions, or just write fan mail, address it to:

Thomas Hahn
Institut für Theoretische Physik
Universität Karlsruhe
e-mail: `hahn@particle.physik.uni-karlsruhe.de`

Contents

1 Getting Started **5**

2 Creating the Topologies **7**

2.1 Topological Objects 7

2.2 `CreateTopologies` 8

2.3 `CreateCTTopologies` 9

2.4 Excluding Topologies 10

2.5 Selecting Starting Topologies 12

3 Inserting Fields into Topologies **15**

3.1 The Three-Level Fields Concept 15

3.2 `InsertFields` 17

3.3 Imposing Restrictions 19

3.4 Structure of the Inserted Topologies 21

4 Drawing Feynman Graphs **22**

4.1 Shaping Topologies 24

5 Creating the Analytic Expressions **27**

5.1 Representation of Feynman Amplitudes 27

5.2 `CreateFeynAmp` 28

5.3 Interpreting the Results 32

5.4 Picking Levels 33

5.5 General Structure of the Amplitudes 34

5.6 On Fermion Chains and Signs 35

5.7 Compatibility with *FeynArts* 1 36

6 Definition of a New Model **37**

6.1 The Generic Model 37

6.2 The Classes Model 44

A The Lorentz Formalism **52**

B The Electroweak Standard Model **53**

C Changes in Version 2.2 **55**

1 Getting Started

FeynArts is a system for generation and visualization of Feynman diagrams and amplitudes based on *Mathematica*. It was conceived in 1990 by Hagen Eck and Sepp Küblbeck. Implementing some decisive new ideas, Hagen Eck developed a second version in 1995. The current version 2.2 was written by Thomas Hahn in 1998. It introduces no new concepts but was nevertheless reprogrammed almost entirely. It has become much faster and user-friendly and also a lot of bugs were removed.

The generation of diagrams is possible at three levels, generic fields, classes of fields, or specific particles. *FeynArts* takes the information about the model from two special files: the generic model file contains the information about the kinematical quantities and the classes model file the information about the particle content. Since the user can create own model files, the applicability of *FeynArts* is virtually unlimited within perturbative quantum field theory. As generic model the Lorentz formalism (`Lorentz.gen`) and as classes model the electroweak Standard Model (`SM.mod`) are supplied. For practising purposes there are also `QED.gen` and `QED.mod` which are slimmed-down versions of `Lorentz.gen` and `SM.mod`, respectively.

Installation

To run *FeynArts* you need *Mathematica* 2.2 or above and `gcc`, the GNU C compiler. Furthermore, the topology editor will work only on an X11 platform.

FeynArts comes in a compressed tar archive `FeynArts22.tar.gz`. To install it, create a directory for *FeynArts*—`FA22`, say—and unpack the archive there:

```
mkdir FA22
cd FA22
gunzip -c ../FeynArts22.tar.gz | tar xvf -
./compile
```

The last line invokes a script to compile the C programs needed by *FeynArts*. `compile` creates binaries whose name has the system type appended, so the binaries for different platforms can coexist in one directory.

The directory structure after installing *FeynArts* should be

`FA22/`	the *FeynArts* directory
`FA22/FeynArts.m`	the main program
`FA22/SetUp.m`	initialization file
`FA22/FeynArts/`	directory containing the *FeynArts* code
`FA22/Models/`	directory containing the model files
`FA22/GraphInfo/`	directory containing the shapes of topologies
`FA22/TopEdit.tm`	topology editor source code
`FA22/TopEdit_`*sysid*	`TopEdit` executable
`FA22/mpslatex.c`	converter from *Mathematica* PostScript to LaTeX
`FA22/mpslatex_`*sysid*	`mpslatex` executable
`FA22/mathdict.pro`	PostScript prologue file for use with LaTeX
`FA22/compile`	shell script to compile `TopEdit.tm` and `mpslatex.c`
`FA22/README`	additional information about this release
`FA22/HISTORY`	general blurb about the evolution of *FeynArts*

sysid is a string that identifies the platform; it is the value of the environment variable `HOSTTYPE` in the Bourne shell (`sh`) and typically has values such as `alpha`, `i486`, etc.

Some options of *FeynArts* functions take a list of objects. If there is only one element in this list, the list can usually be omitted, e.g. `ExcludeTopologies -> Tadpoles` instead of `ExcludeTopologies -> {Tadpoles}`. One notable exception is the specification of levels, see Sect. 3.1.

Some *FeynArts* functions write messages to the screen to indicate their progress. These messages can be partially or completely suppressed by setting `$Verbose` to 0 (no messages) or 1 (summary messages only). The default is 2 (all messages).

Permanent changes of parameters, options, etc. can be placed in `SetUp.m`. Patching the *FeynArts* code directly is discouraged since it is inherently unportable.

The program `mpslatex` converts files in the *Mathematica* PostScript format (MPS) to LaTeX `picture` environments in which the graphical part is contained in a PostScript `\special` and the labels are in LaTeX code. `mpslatex` is called internally by `Paint` when using the option `TeXLabels -> True` (cf. Sect. 4) but can just as well be used for other *Mathematica* figures. The syntax is

`mpslatex_`*sysid* *width* *height* `<` *infile.mps* `>` *outfile.tex*

where *width* and *height* specify the size of the LaTeX figure in inches.

2 Creating the Topologies

2.1 Topological Objects

For the purposes of *FeynArts*, a *topology* is a set of lines (propagators) connecting a set of points (vertices). Furthermore, topologies in *FeynArts* are restricted to be *connected* topologies, where every part of the topology is connected to the rest with at least one propagator.

A vertex is characterized by two numbers: its adjacency and its counter-term order. The *adjacency* is the number of propagators that run into the vertex.

`Vertex[`*adj*`][`*n*`]`	vertex with adjacency *adj*, counter-term order 0, and number *n*
`Vertex[`*adj*`,` *cto*`][`*n*`]`	vertex with adjacency *adj*, counter-term order *cto*, and number *n*

A propagator connects two vertices, possibly carrying a field.

`Propagator[`*t*`][`*from*`,` *to*`]`	propagator of type *t* running from *from* to *to*
`Propagator[`*t*`][`*from*`,` *to*`,` *field*`]`	propagator of type *t* running from *from* to *to* carrying field *field*
possible types of propagators:	
`Incoming, Outgoing`	external propagator flowing in or out
`External`	undirected external propagator
`Internal`	internal propagator which is not part of a loop
`Loop[`*n*`]`	internal propagator on loop *n*[*]

The propagators, then, are collected into topologies.

`Topology[`p_1`,` p_2`, ...]`	representation of a topology with propagators p_i
`Topology[`*s*`][...]`	the same with combinatorical factor $1/s$
`TopologyList[`t_1`,` t_2`, ...]`	a list of topologies t_i
`TopologyList[`*info*`][...]`	the same with an additional information field

[*] The *n* in `Loop[`*n*`]` is not the actual number of the loop—which in general cannot be determined unambiguously—but the number of the one-particle irreducible conglomerate of loops.

2.2 CreateTopologies

The basic function to generate topologies is `CreateTopologies`. It creates topologies based on how many loops and external legs they have.

`CreateTopologies[`*l*`,` *i* `-> o]`	create topologies with *l* loops, *i* incoming, and *o* outgoing legs
`CreateTopologies[`*l*`,` *e*`]`	create topologies with *l* loops and *e* external legs

Load *FeynArts*.

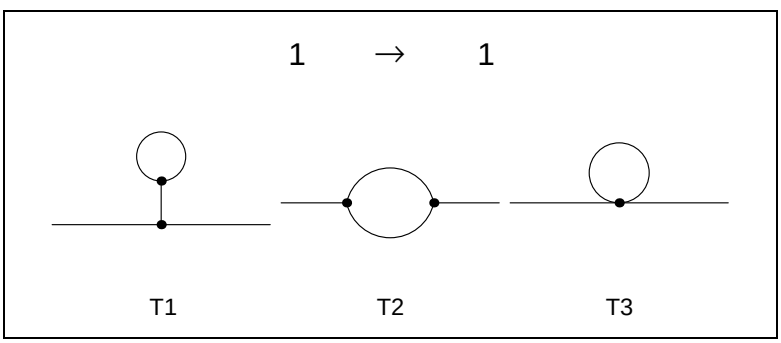
```
In[1]:= <<FeynArts.m

FeynArts 2.2
by Hagen Eck and Sepp Kueblbeck
last revision: 3 Aug 98 by Thomas Hahn
```

Create all topologies with one loop, one incoming, and one outgoing line. The results are collected in a `TopologyList`.

```
In[2]:= CreateTopologies[1, 1 -> 1]
Out[2]= TopologyList[
  Topology[2][
    Propagator[Incoming][Vertex[1][1], Vertex[3][101]],
    Propagator[Outgoing][Vertex[1][2], Vertex[3][101]],
    Propagator[Internal][Vertex[3][2], Vertex[3][101]],
    Propagator[Loop[1]][Vertex[3][2], Vertex[3][2]] ],
  Topology[2][
    Propagator[Incoming][Vertex[1][1], Vertex[3][2]],
    Propagator[Outgoing][Vertex[1][2], Vertex[3][101]],
    Propagator[Loop[1]][Vertex[3][2], Vertex[3][101]],
    Propagator[Loop[1]][Vertex[3][2], Vertex[3][101]] ],
  Topology[2][
    Propagator[Incoming][Vertex[1][1], Vertex[4][2]],
    Propagator[Outgoing][Vertex[1][2], Vertex[4][2]],
    Propagator[Loop[1]][Vertex[4][2], Vertex[4][2]] ] ]
```

The painted version of these topologies is much easier to understand than the list form.

```
In[3]:= Paint[%]
```



CreateTopologies uses a recursive algorithm that generates topologies with n legs from topologies with $n-1$ legs [KüBD91]. The recursion iterates down to zero external legs where it is stopped by a pre-defined set of *starting topologies*.

Several options influence the behaviour of CreateTopologies:

option	*default value*	
Adjacencies	{3, 4}	allowed adjacencies of the vertices
CountertermOrder	0	counter-term order of the topologies
ExcludeTopologies	{}	list of filters for excluding topologies
StartingTopologies	All	list to starting topologies to use

Adjacencies gives the allowed adjacencies the vertices may have, e.g. Adjacencies -> {4} generates only topologies with 4-vertices. In renormalizable field theories, 3 and 4 are the only possible adjacencies.

CountertermOrder specifies at which counter-term order topologies are generated. Note that CreateTopologies creates counter-term topologies for *exactly* the given counter-term order.

2.3 CreateCTTopologies

Beyond one loop, one generally needs counter terms of more than one order, e.g. in a two-loop calculation the second-order counter terms on tree topologies as well as the first-order counter terms on one-loop topologies are needed. This more comprehensive task is handled by CreateCTTopologies. Its options are the same as for CreateTopologies except that CountertermOrder is ignored.

CreateCTTopologies[l, i -> o]	create all counter-term topologies up to order l with i incoming and o outgoing legs
CreateCTTopologies[l, e]	create all counter-term topologies up to order l with e external legs

Once again: use CreateTopologies to generate topologies with l loops and counter-term order *cto*; use CreateCTTopologies to generate counter-term topologies for calculations of order l. Specifically, CreateCTTopologies[l, i -> o] creates all counter-term topologies needed for the topologies created by CreateTopologies[l, i -> o].

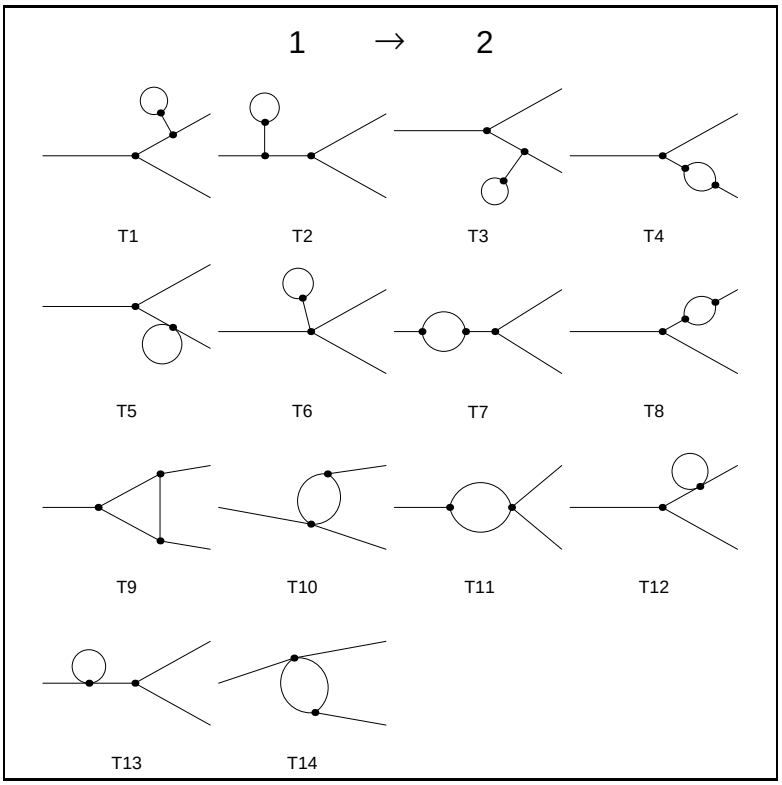
2.4 Excluding Topologies

With the option `ExcludeTopologies` a list of topology exclusion filters can be specified. Such a filter is a special function whose outcome when applied to a topology—`True` or `False`—determines whether the topology is kept or discarded. Some filters are supplied with *FeynArts*, others can be defined. The pre-defined filters work on topologies of any loop number.

	exclude topology if it contains...
`Tadpoles` `TadpoleCTs`	loops counter terms of adjacency 1
`SelfEnergies` `SelfEnergyCTs`	loops counter terms of adjacency 2
`Triangles` `TriangleCTs`	loops counter terms of adjacency 3
`AllBoxes` `AllBoxCTs`	loops counter terms of adjacency $\geqslant 4$
`Boxes[`n`]` `BoxCTs[`n`]`	loops counter terms of adjacency n
`WFCorrections` `WFCorrectionCTs`	self-energy or tadpole loops counter terms on ext. legs
`Internal`	propagators of type `Internal`, i.e. if the topology is one-particle reducible

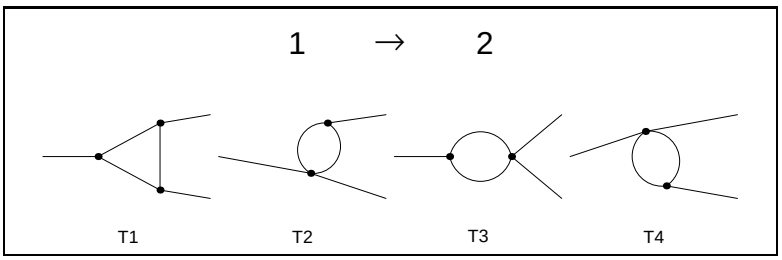
Create all $1 \to 2$ topologies.

In[4]:=
```
CreateTopologies[1, 1 -> 2];
```
In[5]:=
```
Paint[%, ColumnsXRows -> 4]
```



Now the same for irreducible topologies.

In[6]:=
```
CreateTopologies[1, 1 -> 2,
        ExcludeTopologies -> Internal];
```
In[7]:=
```
Paint[%, ColumnsXRows -> 4]
```



To extend the filtering capabilities, you may define your own filter functions. Here is how the `Triangles` filter is defined:

```
$ExcludeTopologies[ Triangles ] = FreeQ[ToTree[#], Centre[3]]&
```

The filter function *must* be defined as a pure function (`func[#]&`) since it will be grouped together with other filter functions by `CreateTopologies`. The function will be passed a topology as its argument. This is important to know for structural operations, e.g. the

following filter excludes topologies with 4-vertices on external legs:

```
$ExcludeTopologies[ V4onExt ] =
  FreeQ[ Cases[#, Propagator[External][__]], Vertex[4] ]&
```

`$ExcludeTopologies[`*name*`] = `*func*`[#]&`	
	defines the filter *name*
`ToTree[`*top*`]`	returns the topology *top* with each loop shrunk to a point named `Centre[`*adj*`][`*n*`]` where *adj* is the adjacency of loop *n*
`Centre[`*adj*`][`*n*`]`	represents the remains of loop *n* with adjacency *adj* after it has been shrunk to a point by `ToTree`

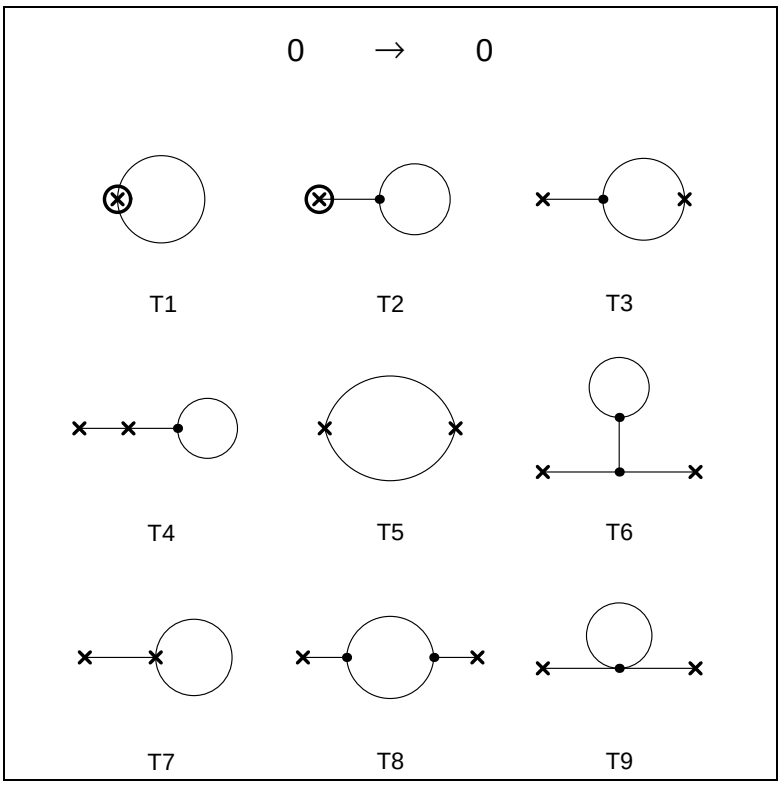
2.5 Selecting Starting Topologies

Using the option `StartingTopologies` to select particular starting topologies may significantly speed up `CreateTopologies`, especially for higher-loop topologies. For example, it is much faster to start without the reducible three-loop starting topologies than to exclude the reducible topologies afterwards. The default setting `All` evaluates to all starting topologies for the given loop number and counter-term order. It is allowed to use patterns, e.g. `StartingTopologies -> Three[_]` selects only the irreducible three-loop starting topologies.

`Theta, Eight, Bicycle`	two-loop starting topologies
`Three[1...8]`	irreducible three-loop starting topologies
`ThreeRed[1...7]`	reducible three-loop starting topologies
`CT[`*l*`,` *cto*`][`*n*`]`	counter-term starting topologies, currently defined for loop number and counter-term order (*l*, *cto*) = (0, 2), (0, 3), (1, 1), (1, 2), and (2, 1)
`StartTop[`*l*`,` *cto*`]`	the starting topologies for loop number *l* and counter-term order *cto*
`SymmetryFactor[`*t*`]`	find the combinatorical factor of the starting topology *t*

You can also draw the starting topologies. The cross marks a first-order and the circled cross a second-order counter term.

In[8]:= `Paint[StartTop[1, 2]]`



Up to three loops all starting topologies including the counter terms are supplied with *FeynArts*. If you need others, you must enter them yourself: Edit `Topology.m` and locate the definition of `StartTop`. There more starting topologies can be appended. Since entering starting topologies is not an everyday job, some restrictions have been imposed that enable *FeynArts* to work with much faster algorithms.

1) There is an important distinction between positive and negative vertex identifiers (the v in `Vertex[`e`][`v`]`). Vertices with negative identifiers are so-called permutable vertices. They are used for weeding out topologically equivalent topologies. The algorithm is roughly the following: The topologies are sorted into some canonical order, and then compared. This simple method, however, fails whenever a graph has a symmetry. In that case, the indices of the symmetrical vertices have to be permuted to give all topologically equivalent versions. It is this "power set" of each topology that is actually compared. If you're not sure which vertices should be permutables, make them *all* permutables. This will be slower, but safer.

2) For the correct functioning of the `ExcludeTopologies` filters it is essential that the propagators on an irreducible conglomerate of loops have the *same* loop number

(the *n* in `Loop[`*n*`]`), no matter how many loops there actually are. For example, the two-loop starting topology `Theta` has only `Loop[1]` propagators.

3) Vertex identifiers must always be unique, e.g. having both a `Vertex[3][1]` and a `Vertex[4][1]` within the same topology is forbidden.

4) To give the starting topology a name which can be used with `StartingTopologies`, define it with `define[`*name*`] = Topology[`*s*`][...]`. To make *name* accessible outside of `Topology.m` it must either be declared in `FeynArts.m` via *name*`::usage` or else live in the `Global‘` context (i.e. `define[Global‘`*name*`] = ...`). If there is only one starting topology, or one always wants to use all of the starting topologies, the `define[...]` can be omitted.

5) To determine the (inverse) symmetry factor (the *s* in `Topology[`*s*`][...]`), enter the topology with an arbitrary factor first (e.g. `Topology[1][...]`), then apply `SymmetryFactor` to find the right symmetry factor, and with it supplement the initial definition.

For example, the two-loop starting topologies are defined as

```
StartTop[2, 0] = TopologyList[
  define[Theta] = Topology[12][
    Propagator[Loop[1]][Vertex[3][-2], Vertex[3][-1]],
    Propagator[Loop[1]][Vertex[3][-2], Vertex[3][-1]],
    Propagator[Loop[1]][Vertex[3][-2], Vertex[3][-1]] ],
  define[Eight] = Topology[8][
    Propagator[Loop[1]][Vertex[4][1], Vertex[4][1]],
    Propagator[Loop[1]][Vertex[4][1], Vertex[4][1]] ],
  define[Bicycle] = Topology[8][
    Propagator[Loop[1]][Vertex[3][-1], Vertex[3][-1]],
    Propagator[Loop[2]][Vertex[3][-2], Vertex[3][-2]],
    Propagator[Internal][Vertex[3][-2], Vertex[3][-1]] ]
]
```

3 Inserting Fields into Topologies

3.1 The Three-Level Fields Concept

FeynArts distinguishes three different levels of fields, generic fields, classes of fields, and specific particles. Field information becomes more and more specific with these levels.

`Generic, Classes, Particles`	field levels in *FeynArts*

Generic fields are the abstract field types.

`F, S, V, U`	basic field types: fermion, scalar, vector, and ghost fields
`SV`	scalar–vector mixing field

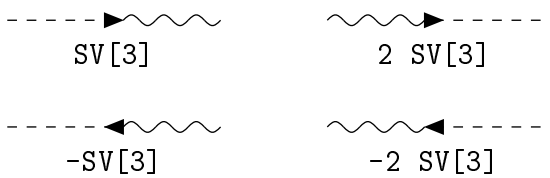
Classes fields represent sets of fields with common properties such as behaviour under charge conjugation. A class is a generic field type with a class number, e.g. `F[1]`. The class specifies which further indices (if any) the class members possess and the range of these indices.

Particles fields are then class members with definite indices, e.g. `F[1, {1, 2}]` if the class `F[1]` has two indices. For classes fields without further indices, classes and particles fields are the same.

Antiparticles (charge-conjugate fields) are denoted by a minus sign in front of the field, e.g. if `F[2, {1}]` is the electron, `-F[2, {1}]` is the positron.

Apart from simple fields, *FeynArts* can also handle mixing fields. A mixing field propagates like any other field, but has no couplings of its own. Instead, it couples like one simple field on the left side and like another simple field on the right side, e.g. if the scalar–vector mixing field `SV[3]` has the mixing partners `{S[3], V[3]}`, it couples as if it were a `S[3]` on the left and a `V[3]` on the right.

Whereas simple fields can have at most two states, the field and its antifield, a mixing field can occur in four states, the mixing field, its antifield, the reversed mixing field, and its antifield, e.g.



Note that, unlike for simple fields, the antifield of `SV[3]` is `-2 SV[3]` and the antifield of `-SV[3]` is `2 SV[3]`. Self-conjugate fields cannot have negative coefficients, i.e. in that case the two possibilities in the bottom row are absent.

Scalar–vector mixing is a special case because it requires different handling internally already at generic level. It is disabled by default.

`$SVMixing = True (False)`	enable (disable) scalar–vector mixing fields (must be set *before* the model is initialized)

At generic level there are no antiparticles or reversed mixing propagators. (Exception: `VS` is used as the reversed `SV` field.)

`F`	generic fermion field
`F[`n`]`	fermion class n, e.g. `F[2]` is the class of leptons in `SM.mod`
`F[`n`, {`i`, ...}]`	member of fermion class n, e.g. `F[2, {1}]` is the electron in `SM.mod`
-field	charge-conjugate of *field*
`2` *mixingfield*	*mixingfield* with left and right partner reversed

Using different levels of fields is a natural concept in perturbative field theory. The kinematical structure of a coupling is determined at the generic level. Consider the scalar–fermion–fermion coupling in the Lorentz formalism. Its kinematical structure is

$$C_{\mathrm{SFF}} = G^{\omega_-}_{\mathrm{SFF}}\,\omega_- + G^{\omega_+}_{\mathrm{SFF}}\,\omega_+$$

($\omega_\pm$ being the chirality projectors). $G^{\omega_-}_{\mathrm{SFF}}$ and $G^{\omega_+}_{\mathrm{SFF}}$ are generic coupling constants which carry two kinds of indices: the fields they belong to and the kinematical object they appear with. At classes level, the coupling constants are resolved but not the indices. E.g. for the $\mathrm{H}\ell_i\ell_j$ coupling in the electroweak Standard Model (Higgs–lepton–lepton with generation indices i and j, i.e. $\ell_1 = \mathrm{e}$, $\ell_2 = \mu$, $\ell_3 = \tau$) they happen to be identical:

$$G^{\omega_-}_{\mathrm{H}\ell_i\ell_j} = G^{\omega_+}_{\mathrm{H}\ell_i\ell_j} = -\frac{\mathrm{i}e}{2\sin\theta_{\mathrm{W}}\,M_{\mathrm{W}}}\delta_{ij}m_{\ell_i}$$

Finally, at particles level, the generation indices i and j are resolved, so for instance the $\mathrm{He}\mu$ coupling is zero because of the δ_{ij}.

The *FeynArts* functions `InsertFields`, `CreateFeynAmp`, and `Paint` can operate at different fields levels. The level specification is analogous to the usual *Mathematica* level

specifications with e.g. `{3}` vs. `3`. If, for example, `InsertFields` is used with the option `InsertionLevel -> {Particles}`, the result will contain insertions *only at* particles level. In contrast, `InsertionLevel -> Particles` (no braces) will produce insertions *down to* particles level. More than one level may be specified, e.g. `{Generic, Particles}` skips the classes level.

3.2 `InsertFields`

The computer algebraic generation of Feynman diagrams corresponds to the distribution of fields over topologies in such a way that the resulting diagrams contain only couplings allowed by the model. The function for this is `InsertFields`.

`InsertFields[`t`, {`$i_1, i_2, \ldots$`} -> {`$o_1, o_2, \ldots$`}]`

insert fields into the `TopologyList` t where the incoming fields are $i_1, i_2, \ldots$ and the outgoing fields are $o_1, o_2, \ldots$

Create irreducible $1 \to 2$ topologies.

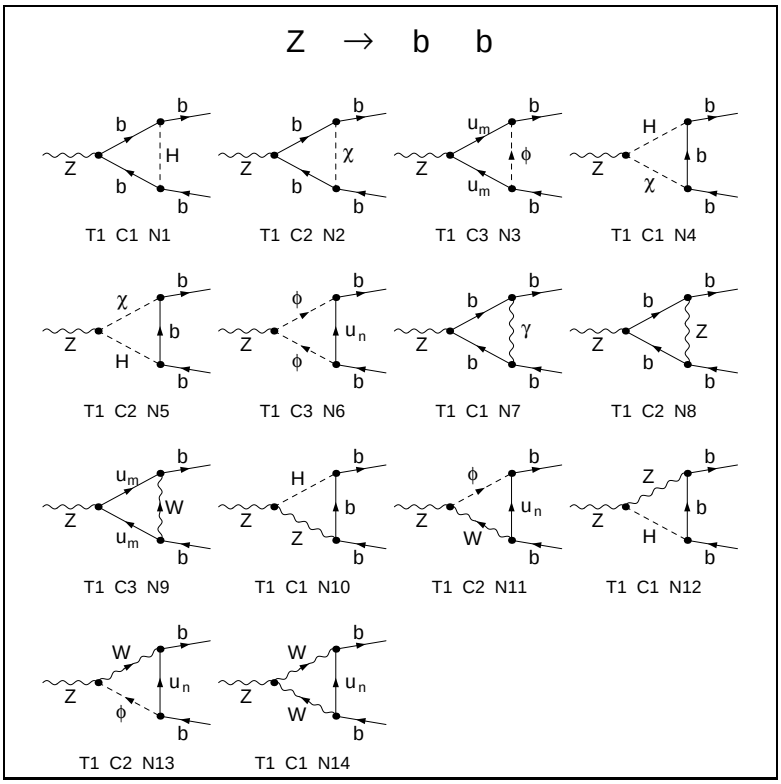
```
In[9]:= t12 = CreateTopologies[1, 1 -> 2,
            ExcludeTopologies -> Internal];
```

Insert $Z \to b\bar{b}$.

```
In[10]:= InsertFields[ t12,
             V[2] -> {F[4, {3}], -F[4, {3}]} ];

initializing generic model file Models/Lorentz.gen
> $SVMixing is OFF
generic model Lorentz initialized

initializing classes model file Models/SM.mod
> 46 particles (incl. antiparticles) in 16 classes
> $CounterTerms are ON
> 88 vertices
> 108 counter terms of order 1
> 1 counter terms of order 2
classes model SM initialized

inserting at level(s) {Generic, Classes}
> Top. 1: 6 Generic, 14 Classes insertions
> Top. 2: 0 Generic, 0 Classes insertions
> Top. 3: 0 Generic, 0 Classes insertions
> Top. 4: 0 Generic, 0 Classes insertions
in total: 6 Generic, 14 Classes insertions
```

`Paint` works also with inserted topologies.

```
In[11] := Paint[%, ColumnsXRows -> 4,
                   PaintLevel -> {Classes}]
```



`InsertFields` accepts the following options.

option	*default value*	
`InsertionLevel`	`Classes`	level specification (see end of Sect. 3.1)
`GenericModel`	`"Lorentz"`	generic model to use
`Model`	`"SM"`	classes model to use
`ProcessName`	`Automatic`	name for the process; used as a filename for graphics output
`RemoveEmptyTops`	`True`	whether to remove topologies for which there are no insertions
`ExcludeFieldPoints`	`{}`	couplings to exclude
`ExcludeParticles`	`{}`	fields to exclude
`Restrictions`	`{}`	restrictions for diagram generation
`LastSelections`	`{}`	field patterns which must or must not appear in the final output

`GenericModel` specifies the model file containing the generic propagators and couplings (the extension `.gen` is always added to the file name). `Model` specifies the classes model file containing the classes definitions and couplings (extension `.mod`). The model name is a string but may be given as a symbol if this is possible (e.g. a name like `2HD` cannot be represented by a symbol because it starts with a digit).

`ProcessName -> Automatic` concocts a process name from the particles involved in the process. For example, `V[2] -> {F[4, {3}], -F[4, {3}]}` produces `V2F4f4`. This name has no purpose other than to be used as a file name by `Paint` for saving graphics output.

3.3 Imposing Restrictions

It is often necessary to restrict the number of diagrams generated by `InsertFields`. This can be done in several ways.

`ExcludeParticles -> {`*fields*`}` excludes insertions containing *fields*. Patterns are permitted in *fields*, e.g. `ExcludeParticles -> F[_, {2 | 3}]` excludes all second and third generation fermion fields. Note that excluding a field at a particular level automatically excludes derived fields at lower levels including their antiparticles. Excluding e.g. the classes field `F[1]` will also exclude `-F[1]`, `F[1, {1}]`, `-F[1, {1}]`, ... Furthermore, `ExcludeParticles` has no effect on the external particles. (After all, it would be rather senseless to specify certain external fields and exclude them in the same line.)

`ExcludeFieldPoints -> {`*couplings*`}` excludes insertions containing *couplings*. A coupling of counter-term order *cto* is specified as `FieldPoint[`*cto*`][`*fields*`]`. Patterns are allowed, e.g. `FieldPoint[_][-F[2, {1}], F[1, {1}], S[3]]`. Here, too, the exclusion of a field point entails the exclusion of more specific ones derived from it.

`Restrictions -> {`*exclp, exclfp*`}` is a convenient way of specifying abbreviations for `ExcludeParticles` and `ExcludeFieldPoints` statements. For example, `SM.mod` defines

```
NoElectronHCoupling =
  ExcludeFieldPoints -> {
    FieldPoint[0][-F[2, {1}], F[1, {1}], S[3]],
    FieldPoint[0][-F[2, {1}], F[2, {1}], S[1]],
    FieldPoint[0][-F[2, {1}], F[2, {1}], S[2]] }
```

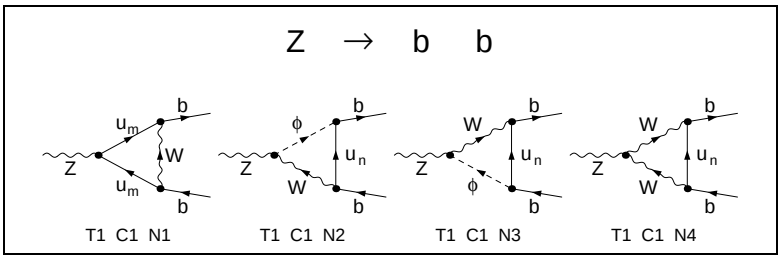
as a short-hand to exclude the electron–Higgs couplings. In `InsertFields` it is used as `Restrictions -> NoElectronHCoupling`.

`LastSelections` is an alternative method to specify field patterns that must or must not appear in the insertions. For example, `LastSelections -> {S, !F[_, {2}]}` forces that the insertions must contain a scalar field and must not contain fermions of the second generation. Like `ExcludeParticles`, `LastSelections` does not affect the external fields. The individual criteria are combined with the logical 'and', i.e. an insertion is only permitted if it fulfills all criteria simultaneously.

While `LastSelections` may seem more general, it works by first generating all diagrams and *afterwards* selecting those that match the given criteria (hence the name). In contrast, `ExcludeParticles` and `ExcludeFieldPoints` work by eliminating particles and couplings *before* starting the insertion process and can thus be significantly faster. They can, on the other hand, only exclude but not force the presence of fields or field points.

Here is the $Z \to b\bar{b}$ example again. Now we select only diagrams in which a W boson occurs.

```
In[12]:= InsertFields[ t12,
           V[2] -> {F[4, {3}], -F[4, {3}]},
           InsertionLevel -> {Classes},
           LastSelections -> V[3] ];

inserting at level(s) {Classes}
> Top. 1: 4 Classes insertions
> Top. 2: 0 Classes insertions
> Top. 3: 0 Classes insertions
> Top. 4: 0 Classes insertions
in total: 4 Classes insertions

In[13]:= Paint[%, ColumnsXRows -> 4]
```



3.4 Structure of the Inserted Topologies

Usually it is not necessary to tamper with (or even see) the `InsertFields` output. All the same, if you want to modify inserted topologies, it is helpful to have an understanding of their structure. Essentially, a topology gets enclosed in a hierarchy of rules with particles insertion nested inside classes insertions nested inside generic insertions.

It is possible to select specific levels out of this hierarchy with `PickLevel`. This function is described in Sect. 5.4.

```
TopologyList[info][
  Topology[s][props] ->
    Insertions[Generic][
      Graph[s_g1, Generic == 1][Field[1] -> F, ...] ->
        Insertions[Classes][
          Graph[s_c1, Classes == 1][Field[1] -> F[1], ...] ->
            Insertions[Particles][
              Graph[s_p1, Particles == 1][Field[1] -> F[1, {1}], ...],
              Graph[s_p2, Particles == 2][Field[1] -> F[1, {2}], ...],
              more particles insertions
            ],
          Graph[s_c2, Classes == 2][Field[1] -> F[2], ...] -> ...,
          more classes insertions
        ],
      Graph[s_g2, Generic == 2][Field[1] -> V, ...] -> ...,
      more generic insertions
    ],
  more topologies
]
```

4 Drawing Feynman Graphs

The drawing routine `Paint` has already been used several times to illustrate the examples in this manual.

`Paint[`t`]`	paint the (bare or inserted) topologies t

`Paint` accepts either a `TopologyList` or a `Topology` as argument. Note however that `InsertFields` adds an information field to `TopologyList` which is used by `Paint`. Thus, if you want to paint a single *inserted* topology, it is better to paint a topology list with one element (viz. `Take[`*toplist*`, {1}]`) than a single topology (viz. *toplist*`[[1]]`) in order to preserve that information field.

option	*default value*	
`PaintLevel`	`InsertionLevel`	level specification (see end of Sect. 3.1)
`ColumnsXRows`	`3`	number of diagrams per column/row, may also be given as a list $\{n_c, n_r\}$
`Destination`	`Screen`	where to put the output (`Screen`, `File`, or `All`)
`TeXLabels`	`False`	whether to produce `.tex` files containing PostScript graphics and LaTeX labels
`FileBaseName`	`ProcessName`	first part of the file name used for saving graphics output
`SheetHeader`	`Automatic`	title for each sheet of graphics
`Numbering`	`True`	whether to display a numbering underneath each diagram
`AutoEdit`	`True`	whether to call the topology editor when encountering an unshaped topology

The `PaintLevel` is not relevant for bare topologies. For inserted topologies, the level they were inserted at is used by default.

`ColumnsXRows` specifies the number of diagrams displayed in each column and row of a sheet. It may be given as a single integer, e.g. `ColumnsXRows -> 4` which means 4 rows

of 4 diagrams each on a sheet, or as a list of two integers, e.g. `ColumnsXRows -> {3, 5}` which means 5 rows of 3 diagrams each.

`Destination -> Screen`, `File`, or `All` displays the graphics on the screen, writes it to a file, or both, respectively. The files produced by `Paint` are in the MPS format which is Level-1 PostScript plus a few special commands that *Mathematica* uses. They must be run through `psfix` (a utility that comes with *Mathematica*) to become ordinary PostScript files. Letting `Paint` produce the files is not the same as saving the screen graphics with `Display` since `Display` does not take care of multipage output.

`TeXLabels -> True` will make `Paint` produce LaTeX files instead of an MPS file. Each page of output is written to a separate file which has the page number appended to its name, e.g. `V2F4f4_1.tex`. The files contain the graphical part in a PostScript `\special` and the labels as LaTeX code embedded in a `picture` environment. Apart from having genuine LaTeX labels in the output, it is in this way easily possible to touch up the labels e.g. before submitting the diagrams in a publication. The files are meant to be included in a document and therefore have no front or back matter. The following line must be added to the preamble of a LaTeX document in which *FeynArts*-made `.tex` files are included to provide the PostScript extensions used by *Mathematica*.

```
\special{header=mathdict.pro}
```

The file `mathdict.pro` resides originally in the *FeynArts* directory. It must be copied to the same directory as the document which includes it. In a kpathsea distribution of TeX it may alternatively be put into the PS-headers directory, which usually has a path like `/usr/local/teTeX/texmf/dvips/base`; this in general requires superuser privileges.

The figure size is 4×4 inches by default (*Mathematica*'s standard size for EPS figures) but can be changed by assigning a different value to the variable `$TeXPictureSize`. It is given in inches either by a number or by a list of two numbers, e.g. `$TeXPictureSize = 5` makes the size 5×5 inches, or `{8, 10}` results in figures 8×10 inch large. The constant `Centimeters` can be used to specify the size in centimeters, like in `$TeXPictureSize = {5, 6} Centimeters`.

`FileBaseName -> "`*string*`"` makes *string* the first part of the file name used for saving graphics output. The default is to use `ProcessName`, the name that `InsertFields` constructs from the process information. The file name always gets an extension appended indicating its format, i.e. `.mps` or `.tex`, and for the single-page output (`TeXLabels -> True`) a page number, e.g. `V2F4f4_1.tex`.

`SheetHeader` specifies the title of each sheet of diagrams. With `Automatic` or `True` the default header is used ("particles$_{\text{in}} \to$ particles$_{\text{out}}$" for inserted or "$n_{\text{in}} \to n_{\text{out}}$" for bare topologies), `False` disables the header, and everything else is taken literally as a header, e.g. `SheetHeader -> "Self-energy diagrams"`.

`Numbering -> False` omits the default numbering of the diagrams. The numbering is of the form `T1 C8 N15` (= topology 1, classes insertion 8, running number 15), or, if `TeXLabels -> True` is set, just `#15`.

`AutoEdit` determines whether `Shape` is called when an unshaped topology is found. This option is useful if your diagrams involve lots of unshaped topologies (e.g. try painting 2-loop, $2 \to 4$ diagrams) and you want a quick (pre)view of the diagrams without going through the effort of shaping them at that moment. *FeynArts* uses a rather sophisticated autoshaping algorithm, but one should not expect too much: autoshaping offers a reasonable starting point, but for publishing-quality diagrams it is usually necessary to refine the shapes by hand.

Other constants affecting the graphical appearance (font size, arrow length, etc.) are defined at the beginning of `Graphics.m` but probably need not be changed.

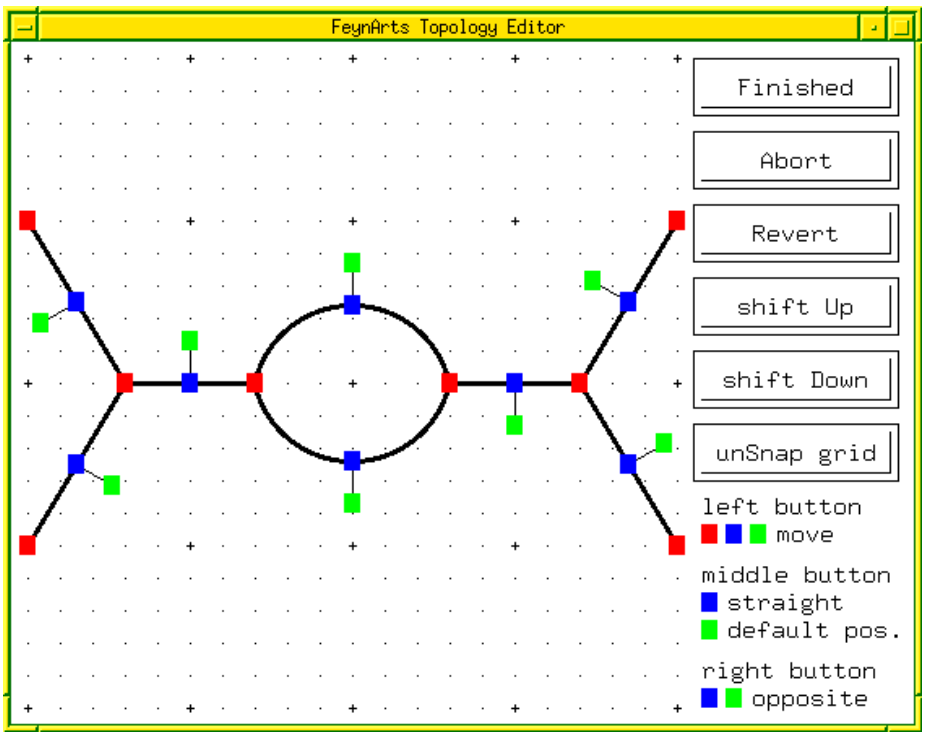
4.1 Shaping Topologies

A topology does not by itself provide information on how to draw it. While the human eye is usually quite skilled in figuring out a nice shape for a topology, at least for not too complex ones, this is a tremendously difficult task for the computer even in simple cases. Indeed, the autoshaping routine is the longest single function in the whole program.

Whenever `Paint` encounters a topology for which is does not yet know the shape, it first does its best to autoshape the topology, and then calls the topology editor (unless `AutoEdit -> False` is set) so that the user can refine the shape. This shaping function can also be invoked directly.

`Shape[`t`]`	invoke the topology editor to shape the topology or topology list t

The topology editor pops up a window which looks like



The *red squares* mark the vertices. Click and drag a red square with the left mouse button to move the respective vertex.

The *blue squares* mark the propagators. Click and drag a blue square with the left mouse button to change the respective propagator's curvature. Click on it with the middle mouse button to make the propagator straight again. Click on it with the right mouse button to reverse the curvature, i.e. to make the line curved in the opposite direction.

The *green squares* mark the label positions (they look like little tags attached to the center of the propagator). Click and drag a green square with the left mouse button to move the respective label. Click on it with the middle mouse button to put the label back to its default position. Click on it with the right button to flick the label to the opposite side of the propagator.

The buttons on the right of the window are largely self-explanatory:

FINISHED exits and takes over the changes.

ABORT exits and discards all changes.

SHIFT UP and SHIFT DOWN shift the whole diagram vertically by $\pm\frac{1}{2}$ a unit.

REVERT reverts to the initial layout of the topology.

(UN)SNAP GRID switches between moving points freely and moving them only onto

grid points.

The topologies need to be shaped only once. The shapes changed during a *Mathematica* session are saved at the end of the session in the directory specified by the variable `$TopologyDataDir`, which by default is the `GraphInfo` directory in the *FeynArts* home directory. They are stored in files named e.g. `Inc2Out1Int2Loop1.m`, meaning that this file contains the shapes of topologies with 2 incoming, 1 outgoing, 2 internal, and 1 loop propagator. If for any reason you do not want to save the shapes, you can set `$ModifiedGraphInfos = {}` before quitting *Mathematica*.

The actual topology editor is a program written in C (`TopEdit.tm`) that communicates with *Mathematica* via the *MathLink* protocol. It must be compiled before running *FeynArts* (see Sect. 1).

5 Creating the Analytic Expressions

5.1 Representation of Feynman Amplitudes

The basic object for a Feynman amplitude is `FeynAmp`. Corresponding to the three insertion levels there are also three amplitude levels.

for a single level:	
`FeynAmp[`*n*`,` *mom*`,` *amp*`]`	Feynman amplitude with name *n*, integration momenta *mom*, and analytic expression *amp*
for multiple levels:	
`FeynAmp[`*n*`,` *mom*`,` *amp*`,` *ru*`]`	Feynman amplitude with name *n*, integration momenta *mom*, and generic analytic expression *amp* including replacement rules *ru* to obtain the other levels
`FeynAmpList[`*info*`][`*amps*`]`	list of `FeynAmps`
`GraphName["`*name*`",` *num*`]`	name (label and numbering) of an amplitude
`Integral[`q_1`,` q_2`, ...]`	representation of the integration momenta

A `FeynAmp` containing a single level has only a single analytic expression *amp*: the amplitude at that level. In case more than one level is present, *amp* is the generic-level amplitude, which can be transformed into the individual amplitudes of the deeper levels by applying the replacement rules *ru* with `PickLevel` (see Sect. 5.4).

FeynArts always produces purely symbolic expressions for Feynman amplitudes, e.g. `PropagatorDenominator[`p`,` m`]` is used to denote $1/(p^2 - m^2)$, simply to prevent *Mathematica* from performing any automatic simplification on it. In fact, *FeynArts* does not attempt to simplify *anything* in the amplitude (e.g. evaluate traces) because this would limit its applicability to a certain class of theories.

Apart from the relatively few symbols *FeynArts* uses by itself in amplitudes, all symbols defined by the model can of course appear in the amplitude.

`Mass[`f`]`	mass of field f
`GaugeXi[`f`]`	gauge parameter of field f
`G[±1][`cto`][`f`][`kin`]`	symmetric ($+1$) or antisymmetric (-1) generic coupling of counter-term order cto involving the fields f and corresponding to the kinematical object kin of the coupling vector (see Sect. 6 for details)
`FourMomentum[`t`, `n`]`	nth momentum of type t (`Incoming`, `Outgoing`, `External`, or `Internal`)
`PropagatorDenominator[`p`, `m`]`	symbolic expression for $1/(p^2 - m^2)$ where p is the momentum and m the mass of the propagator
`FeynAmpDenominator[...]`	collection of `PropagatorDenominators` belonging to loops
`MatrixTrace[`o_1`, `o_2`, ...]`	closed fermion chain involving the noncommuting objects $o_1, o_2, \ldots$
`FermionChain[`o_1`, `o_2`, ...]`	open fermion chain involving the noncommuting objects $o_1, o_2, \ldots$

5.2 `CreateFeynAmp`

Once the possible combinations of fields have been determined by `InsertFields`, the Feynman rules must be applied to produce the actual amplitudes. The function for this is `CreateFeynAmp`.

`CreateFeynAmp[`i`]`	translate the inserted topologies i into Feynman amplitudes

option	*default value*	
AmplitudeLevel	InsertionLevel	level specification (see end of Sect. 3.1)
PreFactor	-I (2 Pi)^ (-4 LoopNr)	overall factor of the amplitude
Truncated	False	whether to apply the model's truncation rules to remove external wave functions
GaugeRules	GaugeXi[_] -> 1	rules to enforce a particular choice of gauge
HoldTimes	False	whether to keep factors from different parts of the amplitude separate (used for tracing errors in the model file)

The default for AmplitudeLevel is to use the same level the topologies were inserted at.

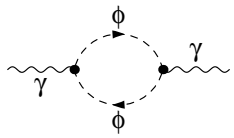
PreFactor specifies the overall factor every diagram is multiplied with. It may contain the symbol LoopNr which is substituted by the actual number of loops of a topology.

If Truncated -> True is set, CreateFeynAmp applies the truncation rules defined in the model file (M$TruncationRules) to the final result. These rules should discard external wave functions, typically spinors or polarization vectors.

To be able to produce amplitudes in an arbitrary gauge, the model file must of course contain the full gauge dependence of the propagators and couplings. For example, Lorentz.gen and SM.mod contain the gauge-dependent propagators (V, S, U) and couplings (SUU) for an arbitrary R_ξ gauge. However, most people prefer to work in the Feynman gauge which is the most convenient one for calculating radiative corrections. Therefore, the default GaugeRules set all gauge parameters to unity. On the other hand, if one wants to check e.g. gauge independence of an expression, GaugeRules -> {} will keep the gauge parameters untouched.

HoldTimes is a useful option for tracing the origin of coupling constants and factors from propagators when debugging a model file. With HoldTimes -> True the constants from couplings and propagators are stored in a function Mult instead of being multiplied. Mult preserves the order and does not cancel common factors. Keeping the individual contributions separated makes it easier to disentangle sources of errors in a wrong amplitude.

Create one diagram from the photon self-energy:




```
In[14]:= t11 = CreateTopologies[1, 1 -> 1][[2]];
In[15]:= AA = InsertFields[t11, V[1] -> V[1],
            ExcludeParticles -> {F, V, U}];

Excluding 3 Generic, 20 Classes, and 36 Particles fields

inserting at level(s) {Generic, Classes}
> Top. 1: 1 Generic, 1 Classes insertions
in total: 1 Generic, 1 Classes insertions

Restoring 3 Generic, 20 Classes, and 36 Particles fields
```

Generate the amplitudes.

```
In[16]:= CreateFeynAmp[AA]

creating amplitudes at level(s) {Generic, Classes}
> Top. 1: 1 Generic, 1 Classes amplitudes
in total: 1 Generic, 1 Classes amplitudes

Out[16]= FeynAmpList[
    Model -> SM, GenericModel -> Lorentz,
    InsertionLevel -> Classes, Restrictions -> {},
    ExcludeParticles -> {F, V, U},
    ExcludeFieldPoints -> {}, LastSelections -> {},
    RemoveEmptyTops -> True, ProcessName -> V1V1,
    Process -> {{V[1], p1, 0}} -> {{V[1], k1, 0}}][
  FeynAmp[
    GraphName[V1V1, Topology == 1, Generic == 1],
    Integral[q1],
    (I/32 RelativeCF
      FeynAmpDenominator[1/(q1^2 - Mass[S[G3]]^2),
        1/((-p1 + q1)^2 - Mass[S[G4]]^2)]
      (p1 - 2 q1)[L1] (-p1 + 2 q1)[L2]
      ep[V[1], p1, L1] ep*[V[1], k1, L2]
      G_SSV^(0)[(Mom[1] - Mom[2])[KI1[3]]]
      G_SSV^(0)[(Mom[1] - Mom[2])[KI1[3]]]) / Pi^4,
    {Mass[S[G3]], Mass[S[G4]],
      G_SSV^(0)[(Mom[1] - Mom[2])[KI1[3]]],
      G_SSV^(0)[(Mom[1] - Mom[2])[KI1[3]]], RelativeCF} ->
      Insertions[Classes][{MW, MW, I EL, -I EL, 2}] ] ]
```

The CreateFeynAmp output is by default displayed on the screen in a special human-readable format. Its full internal representation is shown here.

In[17]:= `InputForm[%]`

Out[17]//InputForm=

```
FeynAmpList[Model -> "SM", GenericModel -> "Lorentz",
  InsertionLevel -> Classes, Restrictions -> {},
  ExcludeParticles -> {F, V, U},
  ExcludeFieldPoints -> {}, LastSelections -> {},
  RemoveEmptyTops -> True, ProcessName -> "V1V1",
  Process -> {{V[1], FourMomentum[Incoming, 1], 0}} ->
    {{V[1], FourMomentum[Outgoing, 1], 0}}][
FeynAmp[
  GraphName["V1V1", Topology == 1, Generic == 1],
  Integral[FourMomentum[Internal, 1]],
  (I/32*RelativeCF*
    FeynAmpDenominator[
      PropagatorDenominator[FourMomentum[Internal, 1],
        Mass[S[Index[Generic, 3]]]],
      PropagatorDenominator[-FourMomentum[Incoming, 1] +
          FourMomentum[Internal, 1],
        Mass[S[Index[Generic, 4]]]] ]*
    FourVector[FourMomentum[Incoming, 1] -
      2*FourMomentum[Internal, 1], Index[Lorentz, 1]]*
    FourVector[-FourMomentum[Incoming, 1] +
      2*FourMomentum[Internal, 1], Index[Lorentz, 2]]*
    PolarizationVector[V[1],
      FourMomentum[Incoming, 1], Index[Lorentz, 1]]*
    Conjugate[PolarizationVector][V[1],
      FourMomentum[Outgoing, 1], Index[Lorentz, 2]]*
    G[-1][0][-S[Index[Generic, 3]],
      -S[Index[Generic, 4]], V[1]][
      FourVector[Mom[1] - Mom[2], KI1[3]]]*
    G[-1][0][S[Index[Generic, 3]],
      S[Index[Generic, 4]], V[1]][
      FourVector[Mom[1] - Mom[2], KI1[3]]])/Pi^4,
    {Mass[S[Index[Generic, 3]]],
      Mass[S[Index[Generic, 4]]],
      G[-1][0][-S[Index[Generic, 3]],
        -S[Index[Generic, 4]], V[1]][
        FourVector[Mom[1] - Mom[2], KI1[3]]],
      G[-1][0][S[Index[Generic, 3]],
        S[Index[Generic, 4]], V[1]][
        FourVector[Mom[1] - Mom[2], KI1[3]]],
      RelativeCF} ->
    Insertions[Classes][{MW, MW, I*EL, -I*EL, 2}] ] ]
```

5.3 Interpreting the Results

Although the analytical expression of an amplitude (the third element of a `FeynAmp`) may look complicated as a whole, the origin of the individual parts can easily be recounted. In the preceding example the respective terms have the following meaning:

`I/32` and `Pi^-4` together are the product of all scalar factors from the (generic) couplings and propagators including the prefactor of the diagram. The fact that both terms are displayed apart in the output is a peculiarity of *Mathematica*.

The symbol `RelativeCF` stands for the *relative c*ombinatorical *f*actor of a classes or particles amplitude with respect to the generic amplitude. It is inserted by the replacement rules for the deeper levels.

`FeynAmpDenominator` collects all denominators belonging to a loop, i.e. those whose momentum is not fixed by conservation of the external momenta. If the structure of the propagators is more intricate (e.g. gauge-boson propagators in a general gauge), a sum of several `FeynAmpDenominators` can appear in one amplitude.

`(p1 - 2 q1)[L1]` and `(-p1 + 2 q1)[L2]` are the kinematical objects—four-vectors in this case—that come from the generic coupling structure, the first from the left and the second from the right vertex. `L1` and `L2` are Lorentz indices. The external momenta are assigned such that the total momentum flows in through the incoming particles, and out through the outgoing particles, i.e. $\sum_{in} p_i - \sum_{out} p_i = 0$, not $\sum_{all} p_i = 0$.

`ep[V[1], p1, L1]` and `ep*[V[1], k1, L1]` are the polarization vectors of the incoming and outgoing photon, the latter of which is complex conjugated.

The two factors $\mathtt{G}^{(0)}_{\mathtt{SSV}}$`[(Mom[1] - Mom[2])[KI1[3]]]` are the generic coupling constants of the scalar–scalar–vector coupling associated with the kinematical object `(Mom[1] - Mom[2])[KI1[3]]`, a four-vector. (The `SSV` coupling is proportional to only one kinematical object.) The superscript `(0)` refers to counter-term order 0. The two `G`'s (one for the left and one for the right vertex) look identical but are not the same internally because in the human-readable output format some indices are suppressed.

The replacement rules (the fourth element of a `FeynAmp`) particularize the unspecified quantities in the generic expression. For the single classes insertion of the example in the last section the following substitutions are made. (The full form of the generic expressions is written out below in small print to show that all substitutions are distinct.)

```
Mass[S[G3]] -> MW
```
full: `Mass[S[Index[Generic, 3]]]`

```
Mass[S[G4]] -> MW
```
full: `Mass[S[Index[Generic, 4]]]`

$G^{(0)}_{SSV}$`[(Mom[1] - Mom[2])[KI1[3]]] -> I*EL`
full: `G[-1][0][-S[Index[Generic, 3]], -S[Index[Generic, 4]], V[1]][FourVector[Mom[1] - Mom[2], KI1[3]]]`

$G^{(0)}_{SSV}$`[(Mom[1] - Mom[2])[KI1[3]]] -> -I*EL`
full: `G[-1][0][S[Index[Generic, 3]], S[Index[Generic, 4]], V[1]][FourVector[Mom[1] - Mom[2], KI1[3]]]`

```
RelativeCF -> 2
```

5.4 Picking Levels

The replacement rules are not of a form which can directly be applied with *expr* `/.` *rules*. Instead, `PickLevel` has to be used to select a particular level.

`PickLevel[`*lev*`][`*amp*`]`	pick level *lev* from the amplitude *amp*
`PickLevel[`*levs*`][`*toplist*`]`	pick level(s) *levs* from the list of inserted topologies *toplist*

For the case of an amplitude, only one level may be specified. `PickLevel` replaces the generic amplitude and the rules with the amplitude at the selected level (i.e. the `FeynAmp` contains only three elements after `PickLevel`) and adds a running number of the form `Number ==` *n* to the `GraphName`. Once a particular level has been selected, it is no longer possible to choose other levels (even deeper ones), e.g. it is not possible to first select classes level and subsequently particles level from a `FeynAmpList`.

With inserted topologies, `PickLevel` works somewhat different. First, it preserves the structure of the inserted topologies. While in a `FeynAmp` the fourth element (the replacement rules) is deleted, an inserted topology is still of the form *topology* `->` *insertions* after `PickLevel`. Second, more than one level may be picked, with one exception: the generic level cannot be removed because it is needed to create the generic amplitudes. However, the syntax differs from the usual level specification: `PickLevel[`*lev*`][`*t*`]` selects *only* level *lev* from *t* (apart from the generic level); `PickLevel[{`*leva*`,` *levb*`}]` selects levels *leva* and *levb*.

If `CreateFeynAmp` finds that amplitudes at only one level are requested (either directly by `AmplitudeLevel`, or if the topologies were inserted at only one level), it automatically calls `PickLevel` so that the output will contain only the amplitudes at the requested level and no replacement rules.

5.5 General Structure of the Amplitudes

The three-level concept has a big advantage when simplifying the amplitudes. Since the kinematical structure is already determined at the generic level, certain algebraic simplifications can be performed for the generic amplitude only and need not be done over and over for every insertion. For example, a fermionic trace needs to be calculated only for the generic amplitude, from which the expressions for the individual fermion classes or particles are then obtained by applying respectively the classes- or particles-level replacement rules.

Owing to fast computers and powerful software for evaluating amplitudes at one-loop level, however, this conceptually superior technique has not been used much so far. Nevertheless, for higher-loop calculations it will likely become an inevitable modus operandi given the number of diagrams already a two-loop calculation typically involves.

To be able to simplify the amplitudes at different levels it is essential to understand how they are structured. The structure is similar to that of inserted topologies, i.e. an expression embedded in a hierarchical list of insertion rules. The main difference is that the basic ingredient is not a topology but the generic amplitude.

Note that only amplitudes containing more than one level are structured in this way. If only one level is selected in `CreateFeynAmp` or with `PickLevel`, the `FeynAmps` contain only the analytic expression of the amplitude at the chosen level and no rules.

```
FeynAmpList[info][
  FeynAmp[gname, mom, generic amp,
    {Mass[F[Index[Generic, 1]]], ...} ->
      Insertions[Classes][
        {MLE[2], ...} ->
          Insertions[Particles][
            {ME, ...}, {MM, ...}, more particles insertions
          ],
        {MLE[3], ...} -> ...,
        more classes insertions
      ],
  ],
  more amplitudes
]
```

5.6 On Fermion Chains and Signs

One of the most asked-for details of amplitude generation is how the signs of fermionic diagrams are determined.

FeynArts generally uses the methods from [DeEHK92]. That is basically, the fermion flow is fixed at generic level and so-called flip rules are applied if it turns out that in the classes or particles insertion the flow is reversed. Open fermion chains—the ones terminating on external fermions—are ordered opposite to the fermion flow, i.e. the leftmost field in the chain is an antifermion, `-F`.

For amplitudes containing anticommuting fields (fermions or ghosts) the following signs are added:

- a minus for every closed fermion or ghost loop and
- the signature of the permutation that brings the ordinal numbers of the external fermions as connected by the fermion chains into descending order (descending because the fermion chains are ordered opposite to the fermion flow).

  Consider these diagrams:

  Following the fermion chains opposite to the arrows, the external particles are connected as $\{2\text{–}1, 3\text{–}4\}$ in the left diagram, and as $\{3\text{–}1, 2\text{–}4\}$ in the right diagram. The number of permutations needed to bring $\{2\,1\,3\,4\}$ into descending order $\{4\,3\,2\,1\}$ is odd, but even for $\{3\,1\,2\,4\}$. Hence the *s*-channel diagram gets an additional minus sign.

  Note that the signature of the permutation is invariant under interchange of any two whole fermion chains because each fermion chain contributes an even number (two) of external fermions.

In renormalizable theories at most two fermion fields can appear at a vertex, and therefore the construction of the fermion chains in totally unambiguous. This is not so in other, non-renormalizable theories, for instance in the Fermi model. ***FeynArts* cannot correctly build the fermion chains if vertices involving more that two fermions appear** because this information is simply not available from the Feynman rules. In such

a case `$FermionLines = False` must be set and the fermion fields must carry an additional kinematic index (e.g. a Dirac index) with which it is *afterwards* possible (i.e. *FeynArts* does not do this) to find the right concatenation. Also, the above mentioned signs are not resolved in this case.

5.7 Compatibility with *FeynArts* 1

Many programs based on *FeynArts* still use the simpler conventions of version 1. For them to work with *FeynArts* 2.2, a function is provided to translate the `CreateFeynAmp` output into the old format.

`ToFA1Conventions[`*expr*`]`	convert *expr* into the conventions of *FeynArts* 1

Note that `ToFA1Conventions` only renames some symbols. The results may thus not be 100% *FeynArts* 1 compatible since certain kinds of expressions could not be generated with *FeynArts* 1 at all. (That, after all, is why the conventions had to be changed.) In particular, amplitudes containing more than one level will probably not be very useful in programs made for processing *FeynArts* 1 output.

6 Definition of a New Model

6.1 The Generic Model

In the generic model file the following items are defined:

- the kinematic indices of the fields (`KinematicIndices`),
- the generic propagators (`M$GenericPropagators`),
- the generic coupling vectors (`M$GenericCouplings`),
- the fermion flip rules (`M$FermionFlipRule`),
- the truncation rules (`M$TruncationRules`),
- some optional "final-touch" rules (`M$LastGenericRules`).

Apart from these required definitions, the generic model file is a perfectly ordinary *Mathematica* input file and may contain any number of additional statements. For example, `Lorentz.gen` includes `Format` directives with which the objects it introduces are displayed in a nicer form in the output.

Probably the best way to learn how to set up a generic model file is by going through one of the provided generic models, `Lorentz.gen` or `QED.gen`.

Kinematic Indices

A kinematic index is an index transported along a propagator. Due to the special property of renormalizable theories to possess vertices which join at most two fermion fields, spinor indices are not necessary because *FeynArts* can construct the fermion chains itself (cf. Sect. 5.6).

`KinematicIndices[`f`] = {`i`}`	definition of the kinematic indices for field f

For example, in the usual representation of the Poincarè group, only the vector bosons carry a Lorentz index. `Lorentz.gen` thus defines the following kinematic indices.

```
KinematicIndices[ F ] = {};
KinematicIndices[ V ] = {Lorentz};
KinematicIndices[ S ] = {};
KinematicIndices[ SV ] = {Lorentz};
KinematicIndices[ U ] = {}
```

There seems to be a mismatch for the scalar–vector mixing field `SV`: it is declared to have a Lorentz index even though the scalar half has no index at all. However, `CreateFeynAmp` knows about this special case and discards the index on the scalar side when resolving a coupling.

Generic Propagators

A generic propagator defines the kinematical structure of a propagator. In the generic model file the generic propagators are contained in the list `M$GenericPropagators`. Inside this list each propagator is declared by an equation for `AnalyticalPropagator`.

`M$GenericPropagators`	name of the list of generic propagators
`AnalyticalPropagator[`*t*`][`*f*`] == `*expr*	definition of the propagator of type *t* (`Internal` or `External`) for field *f*

The nomenclature perhaps needs explanation: An `External` "propagator" is what is traditionally called external wave function, e.g. a spinor in the case of a fermion field. `Internal` propagators are the propagators in the usual sense connecting two internal field points.

The simplest case are scalar fields which have no external wave function, and propagator

$$\langle S_i(k)|S_i(k)\rangle = \frac{\mathrm{i}}{k^2 - \xi_{S_i} M_{S_i}^2}\,.$$

The corresponding statements in `Lorentz.gen` are

```
AnalyticalPropagator[External][ s S[i, mom] ] == 1,
AnalyticalPropagator[Internal][ s S[i, mom] ] ==
  I PropagatorDenominator[mom, Sqrt[GaugeXi[S[i]]] Mass[S[i]]]
```

When initializing the generic model, *FeynArts* transforms these equations into definitions and for this purpose augments the symbols like `s`, `i`, and `mom` by patterns. It is

important to realize that `CreateFeynAmp` appends the momentum and kinematic indices to the field while resolving the propagators and couplings. These additional elements do not appear in the final output, but must be taken into account in the definition of `AnalyticalPropagator`.

f`[`i`,` mom`, {`μ_1`, ...} -> {`ν_1`, ...}]`	maximal internal pattern layout of field f with classes and particles indices summarized in i, momentum mom, and kinematic indices $\mu_1, \ldots$ on the left and $\nu_1, \ldots$ on the right side of the propagator

For instance, a vector field `V[1]` might intermediately be extended by `CreateFeynAmp` to `V[1, FourMomentum[Internal, 1], {Index[Lorentz, 1]} -> {Index[Lorentz, 5]}]`.

The vector-boson propagator is defined accordingly:

```
AnalyticalPropagator[External][ s V[i, mom, {li2}] ] ==
  PolarizationVector[V[i], mom, li2],
AnalyticalPropagator[Internal][ s V[i, mom, {li1} -> {li2}] ] ==
  -I PropagatorDenominator[mom, Mass[V[i]]] *
    (MetricTensor[li1, li2] - (1 - GaugeXi[V[i]]) *
       FourVector[mom, li1] FourVector[mom, li2] *
       PropagatorDenominator[mom, Sqrt[GaugeXi[V[i]]] Mass[V[i]]])
```

The `s` stands for a possible prefactor (a sign for antiparticles or ± 2 for mixing fields), and is transformed to the pattern `s_.` in the final definition. `i` summarizes the class index and all possible particles indices (final pattern: `i__`) and `mom` stands for the momentum (final pattern: `mom_`). For an internal propagator, `li1` and `li2` are the kinematic indices on the left and right side of the propagator, respectively (final patterns: `li1_` and `li2_`), and an external propagator uses only one set of kinematic indices, `li1`. The names of the patterns (`s`, `i`, `mom`, etc.) can of course be freely chosen, but must be unique. The most common errors in this respect are assignments of `i` or `j` prior to employing them as patterns.

Noncommuting objects appearing in the kinematic vector (typically elements from Dirac algebra) must be wrapped in `NonCommutative` so that their order does not get destroyed by `CreateFeynAmp`. This makes sense only for fermion (and perhaps ghost)

fields since there is no well-defined order for other types of field. The external spinors are for example noncommutative objects:

```
AnalyticalPropagator[External][ s F[i, mom] ] ==
  NonCommutative[
    If[SelfConjugate[F[i]], MajoranaSpinor, DiracSpinor][
      -mom, Mass[F[i]], Sequence@@ Drop[{i}, 1] ]
  ]
```

The example shows furthermore how more complex definitions can be constructed from the knowledge of the pattern structure, e.g. `Sequence@@ Drop[{i}, 1]` selects only the list of particles indices by discarding the first (classes) index.

Generic Couplings

Analogous to a generic propagator, a generic coupling defines the kinematical structure of a coupling. The generic couplings are contained in the generic model file in the list `M$GenericCouplings`. Inside this list each coupling is declared by an equation for `AnalyticalCoupling`.

`M$GenericCouplings`	name of the list of generic couplings
`AnalyticalCoupling[`f_1`,`f_2`,...] == G[`s`][`f_1`,`f_2`,...] . {`k_1`,`k_2`,...}`	definition of the coupling of fields f_1, f_2, ... with coupling-constant vector `G[`s`][`f_1`,`f_2`,...]` and kinematic vector `{`k_1`,`k_2`,...}`
`G[±1][`f_1`,`f_2`,...]`	generic coupling-constant vector, symmetric for $+1$ and antisymmetric for -1

The kinematically extended field structure of the fields must be taken into account similar to the case of generic propagators. Whereas however a field on an internal propagator has two sets of kinematic indices `{li1} -> {li2}` for the left and right side each (provided the field carries kinematic indices at all), a field involved in a coupling obviously needs at most one set of indices.

For example, the quartic gauge-boson coupling is defined in `Lorentz.gen` as

```
AnalyticalCoupling[ s1 V[i, mom1, {li1}], s2 V[j, mom2, {li2}],
    s3 V[k, mom3, {li3}], s4 V[l, mom4, {li4}] ] ==
```

```
G[1][s1 V[i], s2 V[j], s3 V[k], s4 V[l]] .
  { MetricTensor[li1, li2] MetricTensor[li3, li4],
    MetricTensor[li1, li3] MetricTensor[li2, li4],
    MetricTensor[li1, li4] MetricTensor[li3, li2] }
```

The generic coupling "constant" `G` refers actually to a vector of coupling constants, one for each element of the kinematic vector. The `VVVV` coupling would appear in a textbook as

$$C\left(V_\mu^i(p_1), V_\nu^j(p_2), V_\rho^k(p_3), V_\sigma^\ell(p_4)\right) = G^{(1)}_{V_iV_jV_kV_\ell}\, g_{\mu\nu}g_{\rho\sigma} + G^{(2)}_{V_iV_jV_kV_\ell}\, g_{\mu\rho}g_{\nu\sigma} + G^{(3)}_{V_iV_jV_kV_\ell}\, g_{\mu\sigma}g_{\rho\nu}$$
$$= \vec{G}_{V_iV_jV_kV_\ell} \cdot \begin{pmatrix} g_{\mu\nu}g_{\rho\sigma} \\ g_{\mu\rho}g_{\nu\sigma} \\ g_{\mu\sigma}g_{\rho\nu} \end{pmatrix} .$$

When looking up a coupling, the fields of a vertex in general have to be permuted to fit the definition in the model file. For this to work it is necessary that the kinematic vector closes under permutations of fields of one type. In the example above this means that for any permutation of $\{\mu, \nu, \rho, \sigma\}$ (all four fields are of the same type here) one must end up with another (or the same) element of the kinematic vector, which is indeed the case.

There is one exception from this closure requirement: if a permutation of the fields yields either an element of the kinematic vector or the negative of an element of the kinematic vector, the coupling is said to be antisymmetric and specified as `G[-1]`. An example of this is the triple gauge-boson coupling in `Lorentz.gen`.

```
AnalyticalCoupling[ s1 V[i, mom1, {li1}], s2 V[j, mom2, {li2}],
    s3 V[k, mom3, {li3}] ] ==
  G[-1][s1 V[i], s2 V[j], s3 V[k]] .
    { MetricTensor[li1, li2] FourVector[mom2 - mom1, li3] +
      MetricTensor[li2, li3] FourVector[mom3 - mom2, li1] +
      MetricTensor[li3, li1] FourVector[mom1 - mom3, li2] }
```

The kinematic vector has in this case only one element,

$$C\left(V_\mu^i(p_1), V_\nu^j(p_2), V_\rho^k(p_3)\right) = G_{V_iV_jV_k}\left(g_{\mu\nu}(p_2 - p_1)_\rho + g_{\nu\rho}(p_3 - p_2)_\mu + g_{\rho\mu}(p_1 - p_3)_\nu\right) .$$

Because the kinematic vector contains the differences of the momenta, a permutation of $\{(p_1, \mu), (p_2, \nu), (p_3, \rho)\}$ will produce either the kinematic vector itself or its negative.

Fermion Flip Rules

Matters are more complicated for fermionic couplings. When two fermion fields are not in the order in which the coupling is defined in the model file, simple permutation of the fields does not suffice, but in addition an operation called "flipping of fermionic couplings" has to be performed on the kinematic vector. Symbolically, this flip operation can be written as

$$\overline{\psi}\,\Gamma\,\psi \to \overline{\widetilde{\psi}}\,\Gamma'\,\widetilde{\psi}$$

where $\widetilde{\psi}$ is the charge-conjugate of the fermion field ψ and Γ is a product of Dirac matrices. The interesting question is of course how Γ and Γ' are related. This is described in detail in [DeEHK92]. Basically, the only necessary substitution is

$$\gamma_\mu\omega_\pm \to -\gamma_\mu\omega_\mp\,.$$

where $\omega_\pm = (1 \pm \gamma_5)/2$ are the chirality projectors. This is more than just flipping the sign of the whole expression (which could be effected with `G[-1]`) since also the other chirality projector has to be taken.

A case where $\gamma_\mu\omega_\pm$ occurs is the `FFV` coupling in `Lorentz.gen`:

```
AnalyticalCoupling[ s1 F[i, mom1], s2 F[j, mom2],
    s3 V[k, mom3, {li3}] ] ==
  G[-1][s1 F[i], s2 F[j], s3 V[k]] .
    { NonCommutative[DiracMatrix[li3], ChiralityProjector[-1]],
      NonCommutative[DiracMatrix[li3], ChiralityProjector[+1]] }
```

FeynArts implements the fermion flip rules in the following way: For each fermionic coupling a list of position interchange rules is built which specifies how the given vertex is mapped onto the coupling's definition in the model file. For instance, if the given coupling is `C[F[3], -F[3], V[1]]` and the definition in the model file is `C[-F[3], F[3], V[1]]`, the rules would be `{1 -> 2, 2 -> 1, 3 -> 3}`, indicating the interchange of the first two fields.

So in particular, these position interchange rules tell whether the fermions are in the original order (`{1 -> 1, 2 -> 2, ...}`) or reversed (`{1 -> 2, 2 -> 1, ...}`) with respect to the model file. The actual rules for flipping the couplings are then obtained as values of the function `M$FermionFlipRule` whose arguments are the position interchange rules.

In `Lorentz.gen` the following flip rules are defined.

```
M$FermionFlipRule[ 1 -> 2, 2 -> 1, __ ] =
  NonCommutative[dm_DiracMatrix, ChiralityProjector[pm_]] :>
    -NonCommutative[dm, ChiralityProjector[-pm]]

M$FermionFlipRule[ 1 -> 2, 2 -> 1 ] =
  NonCommutative[ds_DiracSlash, ChiralityProjector[pm_]] :>
    -NonCommutative[ds, ChiralityProjector[-pm]]
```

Both rules do more or less the same thing. However, the second rule applies to couplings with exactly two fermions only (the fermion–fermion counter terms) whose coupling contains a `DiracSlash` and not a `DiracMatrix` like the `FFV` coupling. Note also that the rules do not apply for the `FFS` coupling since that contains only $\omega_\pm$ but not $\gamma_\mu\omega_\pm$.

For constructing the arguments of `M$FermionFlipRules` it is important to know that the couplings are sorted according to the generic field types before looking up the flip rules, hence the fermionic fields will always be the first since `F` comes alphabetically before the other field types.

Truncation Rules

Truncation rules are needed for removing external wave functions when the option `Truncated -> True` is used in `CreateFeynAmp`. The rules depend of course on the symbols that are used for the external wave functions and must hence be defined in the generic model file.

In `Lorentz.gen` there are spinors and polarization vectors to be dealt with since scalar particles have no external wave functions. The truncation rules are then

```
M$TruncationRules = {
  PolarizationVector[__] -> 1,
  DiracSpinor[__] -> 1,
  MajoranaSpinor[__] -> 1
}
```

"Final-Touch" Rules

The last operation `CreateFeynAmp` performs on a generic amplitude is to apply the model-dependent `M$LastGenericRules`. These rules are optional and have no particular purpose. In `Lorentz.gen`, for example, they complex-conjugate the outgoing polarization vectors.

```
M$LastGenericRules = {
        (* outgoing vector bosons: throw away signs of momenta *)
  PolarizationVector[p_, _. k:FourMomentum[Outgoing, _], li_] :>
    Conjugate[PolarizationVector][p, k, li]
}
```

This rule has an interesting side-aspect: note that only the head `PolarizationVector` is wrapped in `Conjugate`. This is because the `M$LastGenericRules` are applied with `//.` (replace repeatedly until the expression no longer changes). If the whole polarization vector (including arguments) were wrapped in `Conjugate`, the rules would apply anew in every pass and lead to an endless loop.

6.2 The Classes Model

All particles of a model are arranged in classes. A class is conceptually similar, but not identical, to a multiplet (the fields in one class need not belong to a representation of some group). For single-particle model definitions each particle lives in its own class.

The classes model defines the actual classes of fields in a particular model. It should not define or rely on any kinematic objects so that it can be used with different generic model files.

In the classes model file the following items are defined:

- the index range of all possible particles indices (`IndexRange`),
- the classes and their attributes (`M$ClassesDescription`),
- the classes coupling vectors (`M$CouplingMatrices`),
- some optional "final-touch" rules (`M$LastModelRules`).

Just as the generic model file, the classes model file is an ordinary *Mathematica* input file apart from the three required items and may contain additional definitions. For example, the abbreviations for the `Restrictions` option of `InsertFields` which typically depend on classes fields are usually defined in the classes model file.

Range of Particles Indices

It is necessary to declare the index range of every particles index.

`IndexRange[Index[`t`]] = {`p`}`	declares the index of type t to have the values p

For example, the `Generation` index in `SM.mod` which counts the fermion generations has the range

```
IndexRange[ Index[Generation] ] = {1, 2, 3}
```

Classes Attributes

The attributes of each class is defined by an equation in the list `M$ClassesDescription`.

`M$ClassesDescription`	name of the list of classes descriptors
class `== {`a_1 `->` s_1`, ...}`	descriptor for a class whose attribute a_1 is $s_1, \ldots$

Consider the up-type quark class in `SM.mod`.

```
F[3] == {
      SelfConjugate -> False,
      Indices -> {Index[Generation]},
      Mass -> MQU,
      MatrixTraceFactor -> 3,
      PropagatorLabel -> ComposedChar["u", Index[Generation]],
      PropagatorType -> Straight,
      PropagatorArrow -> Forward }
```

The first two fields in the descriptor are necessary: `SelfConjugate` and `Indices`. The others are optional and have default values.

option	*default value*	
`SelfConjugate`	(required)	how the field behaves under charge conjugation
`Indices`	(required)	list of indices the class carries
`MixingPartners`	{*field* , *field*}	for mixing propagators: left and right partners, e.g. `{S[3], V[3]}`
`Mass`	`Mass[`*field*`]`	symbol to denote the mass of the class members
`MatrixTraceFactor`	`1`	for fermions: factor with which closed fermion loops will be multiplied
`InsertOnly`	`All`	types of propagators on which the field may be inserted
`PropagatorLabel`	`"`*field*`"`	label for the propagator
`PropagatorType`	`Straight`	line type for the propagator
`PropagatorArrow`	`None`	`Forward`, `Backward`, or `None`

An ordinary field also has default "mixing" partners, though trivial ones; e.g. `V[2]` has mixing partners `{V[2], V[2]}`.

`Mass -> `*cmass* specifies a symbol for the mass of the class. If omitted, the expression `Mass[`*field*`]` remains in the output of `CreateFeynAmp`. The masses of particles-level fields will appear as *cmass*`[`*indices*`]` unless specific symbols are declared with the function `TheMass`. `SM.mod` for instance declares the following symbols for the individual up-type quark masses (which would otherwise be `MQU[1]`, `MQU[2]`, and `MQU[3]`):

```
TheMass[ F[3, {1}] ] = MU;
TheMass[ F[3, {2}] ] = MC;
TheMass[ F[3, {3}] ] = MT
```

`MatrixTraceFactor` is a way to compensate for traces over indices which are not explicitly specified. For example, in the electroweak Standard Model (without QCD) the interactions are colour-diagonal and hence the quarks need no colour index. Still, when computing the trace over a quark loop, colour accounts for a factor 3 which is provided by `MatrixTraceFactor -> 3` in the quark class descriptors. `CreateFeynAmp` will complain if fields with different `MatrixTraceFactors` appear in the same loop.

`InsertOnly` specifies on which propagator types the field may be inserted, e.g. a background field (which is not allowed in loops) needs `InsertOnly -> {Internal, External}`.

The line of a propagator is in general part of a circle. This includes straight lines which can be regarded as the degenerate case of infinite radius. The line type is given with `PropagtorType` which can take the values `Straight`, `Sine` (photon-like), `Cycles` (gluon-like), `ScalarDash`, `GhostDash`, or in fact any *Mathematica* graphics directive, e.g. people who fancy red propagators could use `RGBColor[1, 0, 0]`. (Just to detain the enthusiasm: keep in mind that diagrams usually cannot be published in colour.)

The `PropagatorLabel` may contain a letter (e.g. `"Z"`), a LaTeX symbol (e.g. `"\\gamma"`), or a composite character (e.g. `ComposedChar["u", "-"]`). The double backslash is really a single character for *Mathematica* since the first \ is needed to escape the second \. Inside the quotes only one character or LaTeX symbol may appear. Multiple characters must be put in a list, e.g. `{"\\gamma", "Z"}`.

`ComposedChar[`*t*`,` *sub*`,` *sup*`,` *bar*`]`	composite label *t* with subscript *sub*, superscript *sup*, and accent *bar*; the arguments *sub*, *sup*, and *bar* are optional but their position is significant, e.g. `ComposedChar[`*t*`, ,` *sup*`]` corresponds to a label with a superscript only

For example, `ComposedChar["\\phi", "i", "*", "\\bar"]` will produce the label $\bar{\phi}_i^*$ (LaTeX representation `\bar\phi_i^*`).

Indices can be used in a label in the form `Index[`*t*`]` where *t* is the type of the index, like in `ComposedChar["\\nu", Index[Generation]]`. The rendering of indices in the graphical output is subject to the function `Appearance`, if that is defined for the particular type of index. By default, the number of the index is displayed.

`Appearance[Index[`*t*`,` *n*`_Integer]] :=` *a*	definition of the graphical appearance for indices of type *t*
`Alph[`*n*`]`	the *n*th lowercase latin letter
`UCAlph[`*n*`]`	the *n*th uppercase latin letter
`Greek[`*n*`]`	the *n*th lowercase greek letter
`UCGreek[`*n*`]`	the *n*th uppercase greek letter

The functions `Alph`, `UCAlph`, `Greek`, and `UCGreek` are useful for converting the index numbers into latin or greek letters, if this is desired. For example, the following line in `SM.mod` makes the fermion generation indices appear as i, j, ...

```
Appearance[ Index[Generation, i_Integer] ] := Alph[i + 8]
```

Similarly, `Lorentz.gen` adds this definition to render Lorentz indices as μ, ν, ...

```
Appearance[ Index[Lorentz, i_Integer] ] := Greek[i + 11]
```

Particular labels for the fields at particles level can be given by assigning values to the function `TheLabel`. If no specific label is defined for a particles-level field, the label of the associated class is used. E.g. the labels of the individual up-type quarks are declared in `SM.mod` by

```
TheLabel[ F[3, {1}] ] = "u";
TheLabel[ F[3, {2}] ] = "c";
TheLabel[ F[3, {3}] ] = "t"
```

Without these definitions, the up, charm, and top quarks would be labelled according to their classes label: u_1, u_2, and u_3.

In the case of a mixing field, `PropagatorLabel` and `PropagatorType` may be set to a list of two items, one for the left half and one for the right half. For example, `SM.mod` contains the class for a γ–Z mixing field (it is commented out by default).

```
V[4] == {
      SelfConjugate -> True,
      Indices -> {},
      Mass -> MAZ,
      MixingPartners -> {V[1], V[2]},
      PropagatorLabel -> {"\\gamma", "Z"},
      PropagatorType -> Sine,
      PropagatorArrow -> None }
```

Classes Coupling Vectors

By far the most diligent task is to enter the actual coupling vectors of the model, in particular if one endeavours to enter also the counter terms.

The coupling vectors are contained in a list `M$CouplingMatrices`. Each coupling vector is defined by an equation.

`M$CouplingMatrices`	name of the list of coupling vectors
`C[`f_1`,` f_2`, ...] == {{`$c_0^{(1)}$`,` $c_1^{(1)}$`, ...}, {`$c_0^{(2)}$`,` $c_1^{(2)}$`, ...}, ...}`	definition of the coupling for the classes-level fields $f_1, f_2, \ldots$ where the lower index of the c's refers to the counter-term order and the upper index to the component of the kinematic vector

The name `M$CouplingMatrices` is justified because the equations assign to each coupling a list of lists, or matrix (a coupling constant for each counter-term order for each component of the kinematic vector).

The coupling vector must of course have as many entries as in the kinematic vector it corresponds to. Recall the example of the quartic gauge-boson coupling from the last section. It has a kinematic vector with three entries, $(g_{\mu\nu}g_{\rho\sigma}, g_{\mu\rho}g_{\nu\sigma}, g_{\mu\sigma}g_{\rho\nu})$, so accordingly the coupling vector must also have three entries. An actual classes-level quartic gauge-boson coupling then looks like

```
C[ -V[3], V[3], V[1], V[1] ] == -I EL^2 *
  { {2, 4 dZe1 + 2 dZW1 + 2 dZAA1 - 2 CW/SW dZZA1},
    {-1, -2 dZe1 - dZW1 - dZAA1 + CW/SW dZZA1},
    {-1, -2 dZe1 - dZW1 - dZAA1 + CW/SW dZZA1} }
```

This is the $W^+W^-\gamma\gamma$ coupling in `SM.mod`. On the right side there is a list with three entries corresponding to the three entries in the kinematic vector. Each component is again a list where the first element is the usual (counter-term order 0, or tree-level) coupling, the second the counter-term order 1 coupling, and so on. The overall constant `-I EL^2` has been pulled out for clarity. (*Mathematica* automatically threads multiplications over lists, e.g. `x*{a, b}` becomes `{x*a, x*b}`.)

It is possible to enter counter terms only for a subset of couplings. The components of the coupling vector must however always be filled up to the same counter-term order.

In the special case where the kinematic vector has only one element, one level of lists may be omitted, so for example

```
C[ S[2], S[2], V[3], -V[3] ] == I EL^2/(2 SW^2) *
  { {1, 2 dZe1 - 2 dSW1/SW + dZW1 + dZchi1} }
```

is equivalent to

```
C[ S[2], S[2], V[3], -V[3] ] == I EL^2/(2 SW^2) *
  {1, 2 dZe1 - 2 dSW1/SW + dZW1 + dZchi1}
```

The classes fields in `C[`f_1`,` f_2`, ...]` must be in the same generic order as the corresponding `AnalyticalCoupling` defined in the generic model file. This means that if an analytical coupling is defined in the order `FFS`, the classes coupling may not be given in the order `SFF`.

Fields with particles indices must include a pattern for every index in the coupling definition. For example, the $\overline{\nu}_{j_1} e_{j_2} W^+$ coupling is defined as

```
C[ -F[1, {j1}], F[2, {j2}], -V[3] ] ==
  I EL/(Sqrt[2] SW) IndexDelta[j1, j2] *
  { {1, dZe1 - dSW1/SW + dZW1/2 +
          Conjugate[dZfL1[1, j1, j1]]/2 + dZfL1[2, j1, j1]/2},
    {0, 0} }
```

`j1` and `j2` stand for one index each (they are transformed to the patterns `j1_` and `j2_` internally during model initialization). For each index a class possesses a separate pattern has to be specified.

Index-diagonal terms are multiplied with `IndexDelta[`j_1`,` j_2`]`. If the whole coupling is proportional to `IndexDelta` as in the last example, `InsertFields` will use this information to fix indices as far as possible already in the insertion process. In contrast, `InsertFields` cannot constrain other indices if only part of the coupling is diagonal, such as in the first-order counter term of the $\overline{\nu}_{j_1} \nu_{j_2} Z$ coupling.

```
C[ -F[1, {j1}], F[1, {j2}], V[2] ] == I EL *
  { {gL[1] IndexDelta[j1, j2],
      IndexDelta[j1, j2] (gL[1] dZZZ1/2 + dgL[1]) +
      gL[1] dZfL1cc[1, j1, j2]},
    {0, 0} }
```

The full index diagonality of the tree-level coupling in this case is of course respected as usual.

"Final-Touch" Rules

Analogous to the `M$LastGenericRules` in the generic model, the classes model file allows to define optional `M$LastModelRules`. These rules are applied as the last operation before `CreateFeynAmp` returns its result. To avoid endless loops it must be kept in mind that the `M$LastModelRules` are applied with `//.` (replace repeatedly until expression no longer changes).

`SM.mod` currently does not define any `M$LastModelRules`.

A The Lorentz Formalism

`Lorentz.gen` contains the definitions of the generic propagators and couplings for a relativistic field theory with scalar, spinor, and vector fields transforming according to the usual representation of the Poincarè group. It introduces the following symbols.

`MetricTensor[`μ`,` ν`]`	the metric tensor $g_{\mu\nu}$
`DiracSpinor[`p`,` m`,` i`]`	the spinor of a Dirac fermion with momentum p, mass m, and particles indices i
`MajoranaSpinor[`p`,` m`,` i`]`	the spinor of a Majorana fermion with momentum p, mass m, and particles indices i
`PolarizationVector[`v`,` p`,` μ`]`	the polarization vector of the vector boson v with Lorentz index μ associated with momentum p
`DiracMatrix[`μ`]`	the Dirac matrix γ_μ
`DiracSlash[`p`]`	$\gamma_\mu p^\mu$
`ChiralityProjector[±1]`	the chirality projectors $\omega_\pm = (1 \pm \gamma_5)/2$
`FourVector[`p`,` μ`]`	the four-vector p_μ

The four spinor states can actually all be represented by just one symbol depending on its position in a `FermionChain` and type of momentum:

`FermionChain[` *spinor*`[FourMomentum[Incoming,` n`],` m`,` i`],` `...` `]` $= \overline{v} \cdot \ldots$

`FermionChain[` *spinor*`[FourMomentum[Outgoing,` n`],` m`,` i`],` `...` `]` $= \overline{u} \cdot \ldots$

`FermionChain[` `...`, *spinor*`[FourMomentum[Incoming,` n`],` m`,` i`]` `]` $= \ldots \cdot u$

`FermionChain[` `...`, *spinor*`[FourMomentum[Outgoing,` n`],` m`,` i`]` `]` $= \ldots \cdot v$

where *spinor* is either `DiracSpinor` or `MajoranaSpinor`. Majorana spinors are used for fermions with attribute `SelfConjugate -> True`.

Polarization vectors associated with outgoing momenta are brought into the form `Conjugate[PolarizationVector][`v`,` p`,` μ`]` by the `M$LastGenericRules`.

The symbol `FourVector` defined by `Lorentz.gen` should not be confused with the *FeynArts* symbol `FourMomentum`. The latter represents a momentum flowing along propagator lines, and it is very likely that using it for different purposes would upset the internal routines. In contrast, `FourVector` is not modified by *FeynArts*.

B The Electroweak Standard Model

The file `SM.mod` contains the electroweak Standard Model including all counter terms of first order in the conventions of [De93]. Gluons (i.e. QCD) are not (yet) part of `SM.mod`.

coupling constants and masses:	
`EL`	electron charge (Thomson limit)
`CW, SW`	cosine and sine of the weak mixing angle
`MW, MZ, MH`	W, Z, Higgs masses
`MLE[`i`]`	mass of lepton of generation i
`ME, MM, ML`	e, μ, τ masses
`MQU[`i`]`	mass of up-type quark of generation i
`MU, MC, MT`	up, charm, top quark masses
`MQD[`i`]`	mass of down-type quark of generation i
`MD, MS, MB`	down, strange, bottom quark masses
`CKM[`i`, `j`]`	quark-mixing matrix with generation indices i and j
`GaugeXi[A, W, Z]`	photon, W, Z gauge parameters

one-loop renormalization constants:	
`dZe1`	electromagnetic charge RC
`dSW1, dCW1`	mixing angle sine/cosine RC
`dZH1, dMHsq1`	Higgs field and mass RC
`dZW1, dMWsq1`	W field and mass RC
`dMZsq1`	Z mass RC
`dZZZ1, dZZA1, dZAZ1, dZAA1`	Z and photon field RCs
`dMf1[`t`, `i`]`	fermion mass RCs for fermions of type t and generation i
`dZfL1[`t`, `i`, `j`], dZfR1[`t`, `i`, `j`]`	left- and right-handed fermion field RCs for fermions of type t with generation indices i and j
`dCKM1[`i`, `j`]`	quark-mixing matrix RCs with generation indices i and j

The type of a fermion is 1 for neutrinos, 2 for massive leptons, 3 for up-type quarks, and 4 for down-type quarks.

The particle content of `SM.mod` is summarized in the following table.

class	*self-conjugate*	*indices*	*members*		*mass*
`F[1]`	no	`Generation`	`F[1,{1}]`	ν_e	`0`
(neutrinos)			`F[1,{2}]`	ν_μ	`0`
			`F[1,{3}]`	ν_τ	`0`
`F[2]`	no	`Generation`	`F[2,{1}]`	e	`ME`
(massive leptons)			`F[2,{2}]`	μ	`MM`
			`F[2,{3}]`	τ	`ML`
`F[3]`	no	`Generation`	`F[3,{1}]`	u	`MU`
(up-type quarks)			`F[3,{2}]`	c	`MC`
			`F[3,{3}]`	t	`MT`
`F[4]`	no	`Generation`	`F[4,{1}]`	d	`MD`
(down-type quarks)			`F[4,{2}]`	s	`MS`
			`F[4,{3}]`	b	`MB`
`V[1]`	yes		`V[1]`	γ	`0`
`V[2]`	yes		`V[2]`	Z	`MZ`
`V[3]`	no		`V[3]`	W^-	`MW`
`V[4]` (mixing field)[†]	yes		`V[4]`	γ–Z	`MAZ`
`S[1]`	yes		`S[1]`	H	`MH`
`S[2]`	yes		`S[2]`	χ	`MZ`
`S[3]`	no		`S[3]`	ϕ^-	`MW`
`U[1]`	no		`U[1]`	u_γ	`0`
`U[2]`	no		`U[2]`	u_Z	`MZ`
`U[3]`	no		`U[3]`	u_-	`MW`
`U[4]`	no		`U[4]`	u_+	`MW`
`SV[2]` (mixing field)[‡]	yes		`SV[2]`	χ–Z	`MZ`
`SV[3]` (mixing field)[‡]	no		`SV[3]`	ϕ^-–W^-	`MW`

[†] Commented out by default in `SM.mod`.

[‡] Must be enabled with `$SVMixing = True`.

C Changes in Version 2.2

Although *FeynArts* 2.2 has kept the concept and much of the syntax of *FeynArts* 2.1, some incompatible changes were necessary:

FeynArts can be kept whereever you like. You don't have to set any paths. Load it simply with `<<~/whereever_I_put_it/FeynArts.m`. On the other hand, the context no longer contains `HighEnergyPhysics‘`.

For every `ExcludeTopologies` filter for loop diagrams there is now also a corresponding one for counter-term diagrams: `TadpoleCTs`, `SelfEnergyCTs`, `TriangleCTs`, `AllBoxCTs`, `BoxCTs[`*n*`]`, `WFCorrectionCTs`. Also, all filters work on diagrams with an arbitrary number of loops. The details for programming own filters have changed; see Sect. 2.4.

The model must be specified as a string or a symbol (if the model's name allows it). In particular `Model -> {"`*name*`",` *restrictions*`}` or just `Model -> {"`*name*`"}` are no longer valid. Restrictions can be specified directly in `InsertFields` in a much more flexible way; see Sect. 3.3.

The `VS` (vector–scalar mixing) field is reserved for internal use (in fact it is used only for generic level). Reversed mixing propagators are now described by a ± 2 multiplying the field, e.g. `2 SV[3] = {V[3], S[3]}` if `SV[3] = {S[3], V[3]}`. You must remove all entries involving `VS` from the `M$ClassesDescription` list of your model file. (This shouldn't be a problem because formerly two otherwise equivalent definitions for `SV[`*n*`]` and `VS[`*n*`]` were needed.)

`InsertFields` accepts only the `{`*incoming*`} -> {`*outgoing*`}` field specification. The option `FieldSpecification -> {...}` exists no more.

In `SM.mod`: the somewhat awkwardly named quark mixing matrix `Matrix[FQ][...]` is now called `CKM[...]` and its one-loop renormalization constants `dCKM1[...]`.

`ComposedChar` is used differently: e.g. `ComposedChar["\\phi", "i", "*", "\\bar"]` will produce the label $\bar{\phi}_i^*$ (LaTeX representation `\bar\phi_i^*`). Note that LaTeX symbols are used instead of `SymbolChar[...]` (Sect. 6.2 gives details).

`TopologyGraphics` and the entire `HighEnergyPhysics‘Graphics‘` context (including in particular `PropagatorGraphics`) have disappeared. The various constants to modify graphics output (label size, font, arrow length, etc.) are now defined at the beginning of `Graphics.m` but probably need not be changed. `DLabels` (for mixing propagators) have been replaced by simple labels consisting of two parts, e.g. 'W ϕ' instead of W at one position and ϕ at another.

None of the former shapes for topologies can be used. Shaping them from scratch shouldn't cost much effort with the new topology editor.

The function `FA2toFA1` to convert *FeynArts* 2 output into the old *FeynArts* 1 conventions which had to be loaded separately (`<< FA2toFA1.m`) is now called `ToFA1Conventions` and is part of *FeynArts*.

The signs of diagrams with different numbers of external fermions is now correct. E.g. previously the fermion self-energies—diagrams with two external legs—were lacking a minus with respect to diagrams with four (or more) external fermions (the ordinal numbers of the external fermions were sorted into *ascending* order; see Sect. 5.6). In a sense this is a global sign and has little significance, but is noticeable if such diagrams are used as insertions into others, e.g. in the form of renormalization constants.

`CreateFeynAmp` used to eliminate the outgoing momentum via momentum conservation in the special case of diagrams with two external legs (the only case where elimination of one external momentum does not inflate the amplitude), but did so in a way inconsistent with the *FeynArts* momentum conventions, namely `FourMomentum[Incoming, 1] = -FourMomentum[Outgoing, 1]`. External momenta are now not eliminated at all.

`MatrixTraceFactor` now works correctly so it is no longer necessary to multiply diagrams containing quark loops with 3.

`GaugeXi` now corresponds to the standard textbook notation of the gauge parameter, e.g. the gauge-boson propagator is defined as

$$-\frac{\mathrm{i}}{p^2 - M_V^2}\left(g^{\mu\nu} - (1 - \texttt{GaugeXi[}V\texttt{]})\frac{p^\mu p^\nu}{p^2 - \texttt{GaugeXi[}V\texttt{]}\,M_V^2}\right) .$$

(`GaugeXi` used to stand for $1/\xi$ in former versions.)

References

[De93] A. Denner, *Fortschr. Phys.* **41** (1993) 307.

[DeEHK92] A. Denner, H. Eck, O. Hahn, and J. Küblbeck, *Nucl. Phys.* **B387** (1992) 467.

[KüBD91] J. Küblbeck, M. Böhm, and A. Denner, *Comp. Phys. Commun.* **60** (1991) 165.