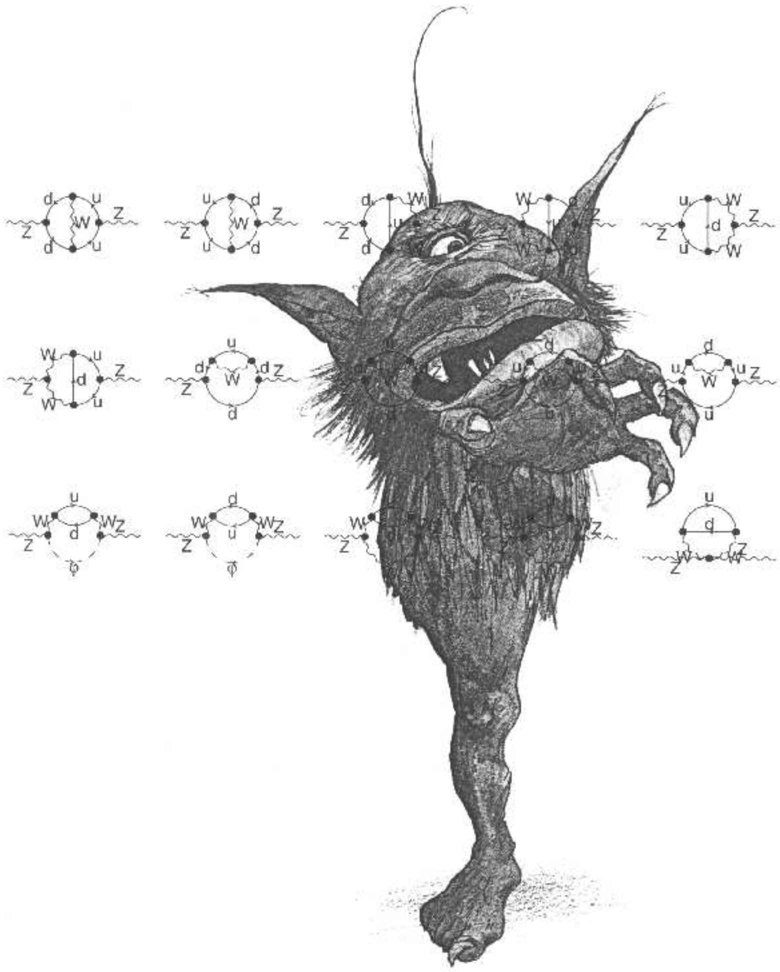


FeynArts 3 User's Guide

October 24, 2000 Thomas Hahn

FeynArts is available from `http://www.feynarts.de`. If you make this software available to others, please provide them with this manual, too. There exists a mailing list for *FeynArts* where updates will be announced. To join, send a mail (any text) to `hahn-feynarts-subscribe@particle.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@feynarts.de`

Contents

1 Getting Started **5**

2 Roadmap of *FeynArts* **7**

3 Creating the Topologies **8**

3.1 Topological Objects 8

3.2 `CreateTopologies` 9

3.3 Creating Counter-term Topologies 10

3.4 Creating Topologies with generic Vertex Functions 11

3.5 Excluding Topologies 12

3.6 Selecting Starting Topologies 14

4 Inserting Fields into Topologies **18**

4.1 The Three-Level Fields Concept 18

4.2 `InsertFields` 20

4.3 Imposing Restrictions 22

4.4 Selecting Insertions 24

4.5 Discarding Insertions 25

4.6 Structure of the Inserted Topologies 26

5 Drawing Feynman Graphs **27**

5.1 Things to do with the Paint Output 29

5.2 Shaping Topologies 30

6 Creating the Analytic Expressions **33**

6.1 Representation of Feynman Amplitudes 33

6.2 `CreateFeynAmp` 34

6.3 Interpreting the Results 38

6.4 Picking Levels 39

6.5 General Structure of the Amplitudes 40

6.6 On Fermion Chains and Signs 41
6.7 Specifying Momenta 42
6.8 Compatibility with *FeynArts* 1 42

7 Definition of a New Model **43**
7.1 The Generic Model 43
7.2 The Classes Model 51

A The Lorentz Formalism **60**

B The Electroweak Standard Model **62**
B.1 The QCD Extension 64
B.2 Background-field Formalism 65

C Changes in Version 3 **66**

1 Getting Started

FeynArts is a *Mathematica* package for the generation and visualization of Feynman diagrams and amplitudes. 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 program was taken up again by Thomas Hahn in 1998 in version 2.2 which introduced no new concepts but was nevertheless reprogrammed almost entirely. It became much faster and user-friendly and also a lot of bugs were removed. The current version 3 features a completely new rendering engine for PostScript and LaTeX, together with full support of the *Mathematica* Frontend's graphical capabilities. It is also no longer dependent on the X platform for topology editing.

Some of the features of version 3 are:

- The generation of diagrams is possible at three levels: generic fields, classes of fields, or specific particles.
- The model information is contained in two special files: The *generic model file* defines the representation of the kinematical quantities like spinors or vector fields. The *classes model file* sets up the particle content and specifies the actual couplings. 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 in several variations (`SM.mod`, `SMQCD.mod`, `SMbgf.mod`) are supplied.
- In addition to ordinary diagrams, *FeynArts* can generate counter-term diagrams and diagrams with placeholders for one-particle irreducible vertex functions (skeleton diagrams).
- *FeynArts* employs the so-called "flipping-rule" algorithm [DeEHK92] to concatenate fermion chains. This algorithm is unique in that it works also for Majorana fermions and the fermion-number-violating couplings they entail (e.g. quark–squark–gluino) and hence allows supersymmetric models to be implemented.
- Vertices of arbitrary adjacency (a requirement for effective theories) are allowed.
- Mixing propagators, such as appear in non-R_ξ-gauges, are supported.
- *FeynArts* produces publication-quality Feynman diagrams in PostScript or LaTeX in a format that allows easy customization.

Installation

FeynArts requires *Mathematica* 3.0 or above. For the topology editor, a Java VM and the J/Link package are needed. Both ingredients can be obtained free of charge from

`http://www.wolfram.com/solutions/mathlink/jlink` (J/Link),
`http://java.sun.com/j2se` (Java).

Note that many systems (e.g. Windows) have a Java VM pre-installed. Follow the instructions that come with J/Link for installation on the various platforms.

FeynArts comes in a compressed tar archive `FeynArts3.tar.gz`. To install it, create a directory for *FeynArts*—`FeynArts`, say—and unpack the archive there:

```
mkdir FeynArts
cd FeynArts
gunzip -c ../FeynArts3.tar.gz | tar xvf -
```

The contents of the *FeynArts* directory after installation should be

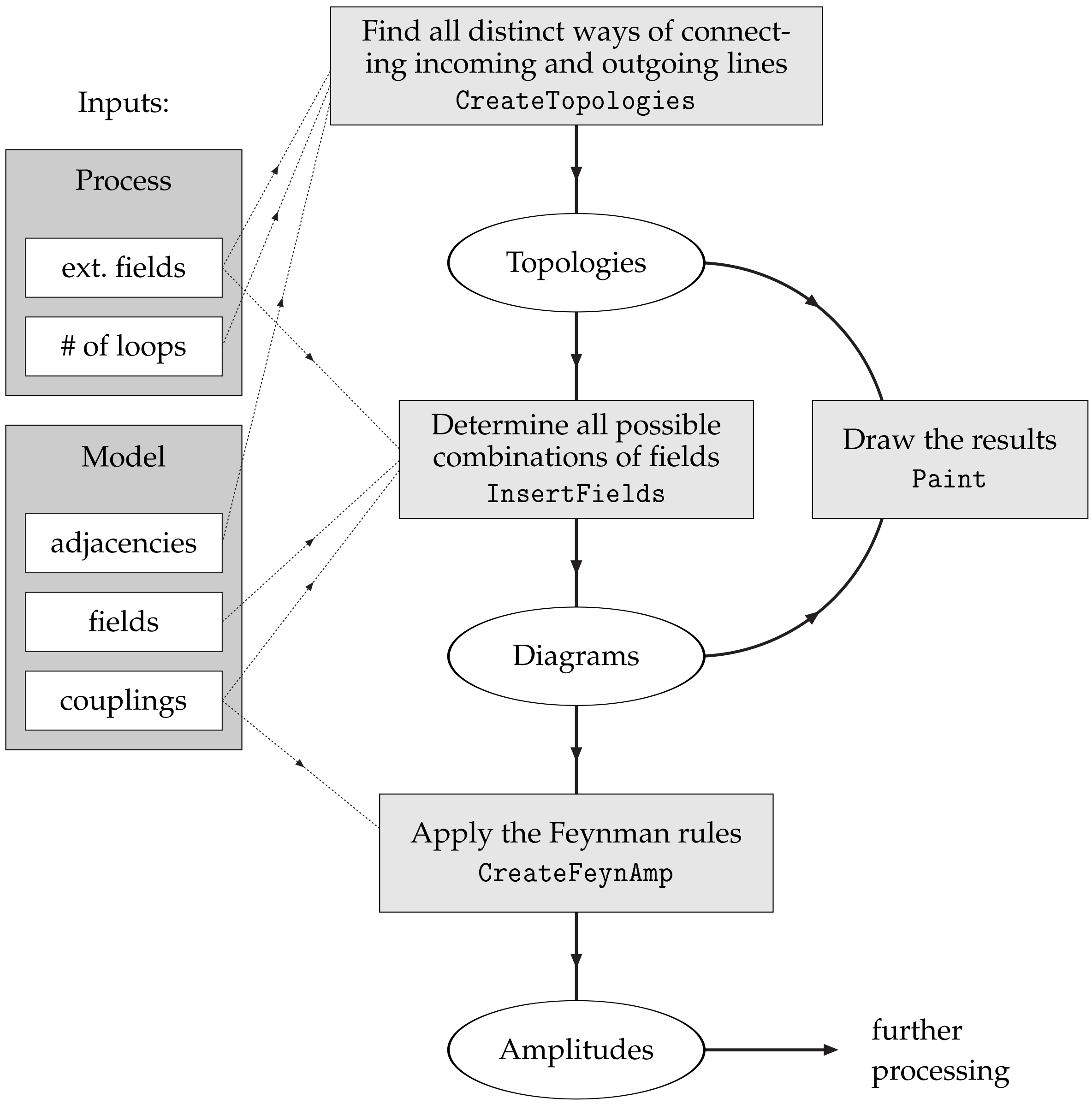
`FeynArts.m`	the main program
`Setup.m`	initialization file
`FeynArts/`	directory containing the *FeynArts* code
`Models/`	directory containing the model files
`GraphInfo/`	directory containing the shapes of topologies
`README`	additional information about this release
`HISTORY`	general blurb about the evolution of *FeynArts*
`ConvertGraphInfos.m`	conversion program for old GraphInfo files

Permanent changes of parameters, options, etc. should be placed in `Setup.m`. Patching the *FeynArts* code directly is not recommended since it is inherently unportable.

Several *FeynArts* functions have options that take a list of objects. Except in the case of level specifications (see Sect. 4.1), the list may be omitted if it contains only one element, e.g. `ExcludeTopologies -> Tadpoles` instead of `ExcludeTopologies -> {Tadpoles}`.

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).

2 Roadmap of *FeynArts*



3 Creating the Topologies

3.1 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 combinatorial 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.

3.2 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‘`

```
FeynArts 3
by Hagen Eck, Sepp Kueblbeck, and Thomas Hahn
last revised 24 Oct 00
```

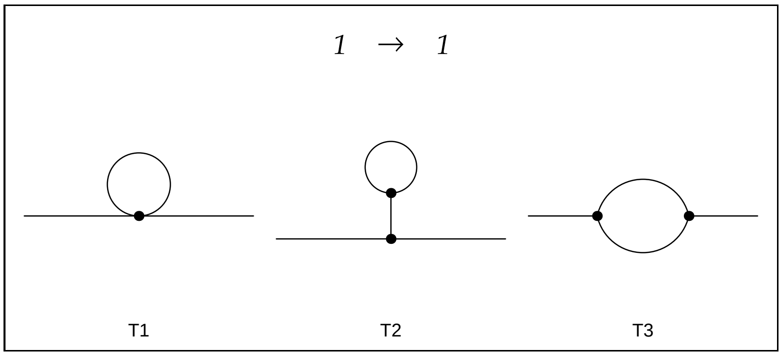
Create all topologies with one loop, one incoming, and one outgoing line. The results are collected in a `TopologyList`.

In[2]:= `CreateTopologies[1, 1 -> 1]`

Out[2]=
```
TopologyList[
  Topology[2][
    Propagator[Incoming][Vertex[1][1], Vertex[4][3]],
    Propagator[Outgoing][Vertex[1][2], Vertex[4][3]],
    Propagator[Loop[1]][Vertex[4][3], Vertex[4][3]] ]
  Topology[2][
    Propagator[Incoming][Vertex[1][1], Vertex[3][3]],
    Propagator[Outgoing][Vertex[1][2], Vertex[3][3]],
    Propagator[Internal][Vertex[3][3], Vertex[3][4]],
    Propagator[Loop[1]][Vertex[3][4], Vertex[3][4]] ],
  Topology[2][
    Propagator[Incoming][Vertex[1][1], Vertex[3][3]],
    Propagator[Outgoing][Vertex[1][2], Vertex[3][4]],
    Propagator[Loop[1]][Vertex[3][3], Vertex[3][4]],
    Propagator[Loop[1]][Vertex[3][3], Vertex[3][4]] ] ]
```

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
`CTOrder`	`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 quantum field theories, 3 and 4 are the only possible adjacencies.

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

3.3 Creating Counter-term Topologies

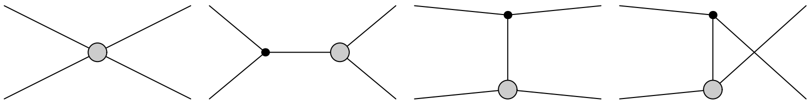
Beyond one loop, one generally needs counter-terms of more than one order, e.g. in a two-loop calculation the second-order counter-terms on tree topologies as well as the first-order counter-terms on one-loop topologies are needed. This more comprehensive task is handled by `CreateCTTopologies`. Its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

3.4 Creating Topologies with generic Vertex Functions

Topologies can also be created with "placeholders" for one-particle irreducible (1PI) vertex functions. Such topologies are sometimes also called skeleton diagrams. The vertex-function placeholders are represented graphically by a grey bubble, e.g.



The idea is to reduce the number of diagrams by calculating the vertex functions separately and inserting the final expression into tree diagrams at the proper places. For example, in a $2 \to 2$ process, the self-energy diagrams are generated once for the s-channel, once for the t-channel, and once for the u-channel. It would of course be much more economic to calculate the necessary 1PI two-point vertex functions only once, and insert them at the proper places in the s-, t-, and u-channel tree diagrams. The problem may not be particularly acute in this example at one loop, but it can easily become significant for higher loop order or more external legs.

Topologies with vertex-function placeholders are generated by `CreateVFTopologies`. Note that there is always a loop order associated with the vertex function, hence one has to specify a loop order also for these "VF" topologies.

`CreateVFTopologies[`*l*`,` *i* `->` *o*`]`	create topologies with placeholders for 1PI vertex functions of order *l* with *i* incoming and *o* outgoing legs
`CreateVFTopologies[`*l*`,` *e*`]`	create topologies containing generic 1PI vertex-function insertions of order *l* with *e* external legs

The creation of "VF" topologies and of counter-term topologies is essentially the same. In fact, the place where the vertex function must later be inserted is given precisely by the location of the cross on the counter-term diagram of the corresponding order.

`CreateFeynAmp` translates the vertex-function placeholders into generic objects of the form `VertexFunction[`*o*`][`f_1`,` f_2`, ...]` which represent the 1PI vertex function $\Gamma^{(o)}_{f_1 f_2 \ldots}$, where o is the loop order and f_1, f_2, ... are the adjoining fields (direction: all incoming), complete with their momenta and kinematic indices. The conventions are such that a `VertexFunction` consists directly of the corresponding 1PI diagrams (as generated by *FeynArts*) without further prefactors.

Internally, the tricky part is to decide which vertex functions are allowed and which are not. Unlike in the case of ordinary counter-terms, `InsertFields` cannot simply look up which vertices are present in the model since there are exceptions: for instance in the electroweak Standard Model the $\gamma\gamma H$ vertex has neither a counter-term nor even a tree-level vertex. Nevertheless, loop diagrams for this vertex exist.

Therefore, different constraints have to be used: only such vertex functions are generated for which there exists a corresponding generic vertex, and which do not violate conservation of quantum numbers. The quantum numbers of the fields are defined in the classes model file (see Sect. 7.2).

3.5 Excluding Topologies

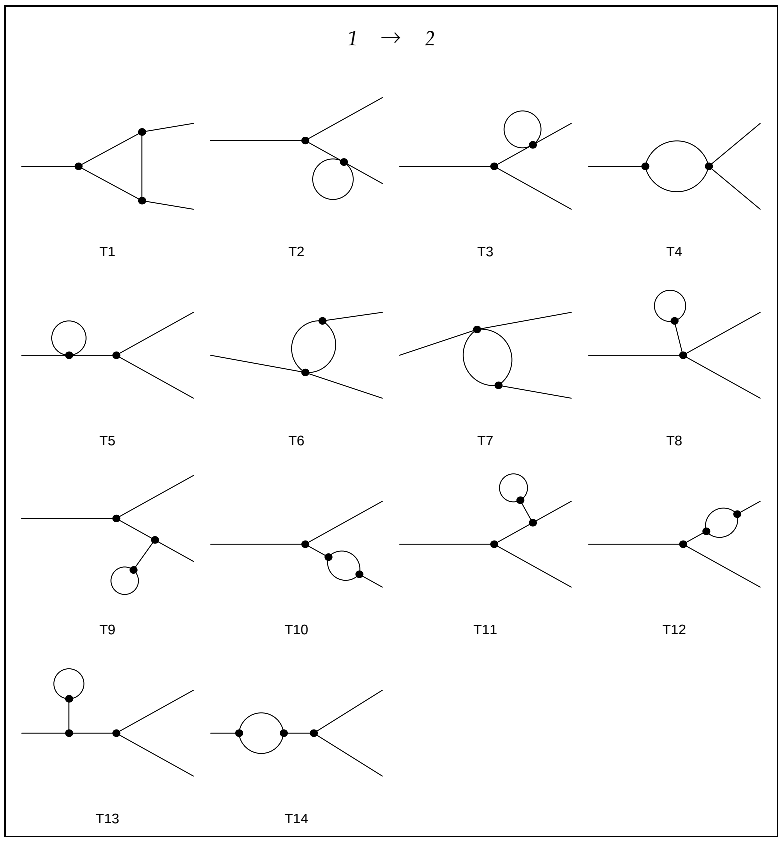
With the option `ExcludeTopologies` a list of topology exclusion filters can be specified. Such a filter is a special function whose outcome when applied to a topology—`True` or `False`—determines whether the topology is kept or discarded. Some filters are supplied with *FeynArts*, others can be defined.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

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

Because the generation of counter-term topologies and topologies with vertex-function insertions (see Sect. 3.4) is so similar, there are no special exclusion filters for the latter—one simply uses the filters for counter-terms.

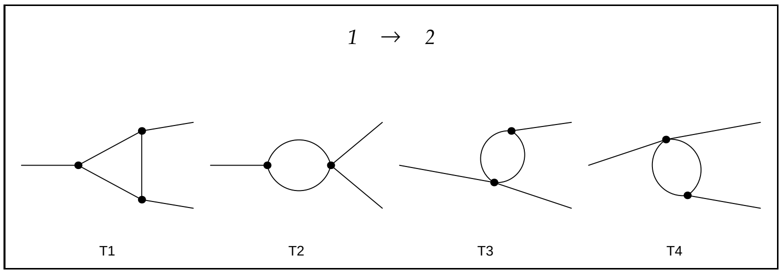
Create all $1 \to 2$ topologies.

```
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 (`foo[#]&`) 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`

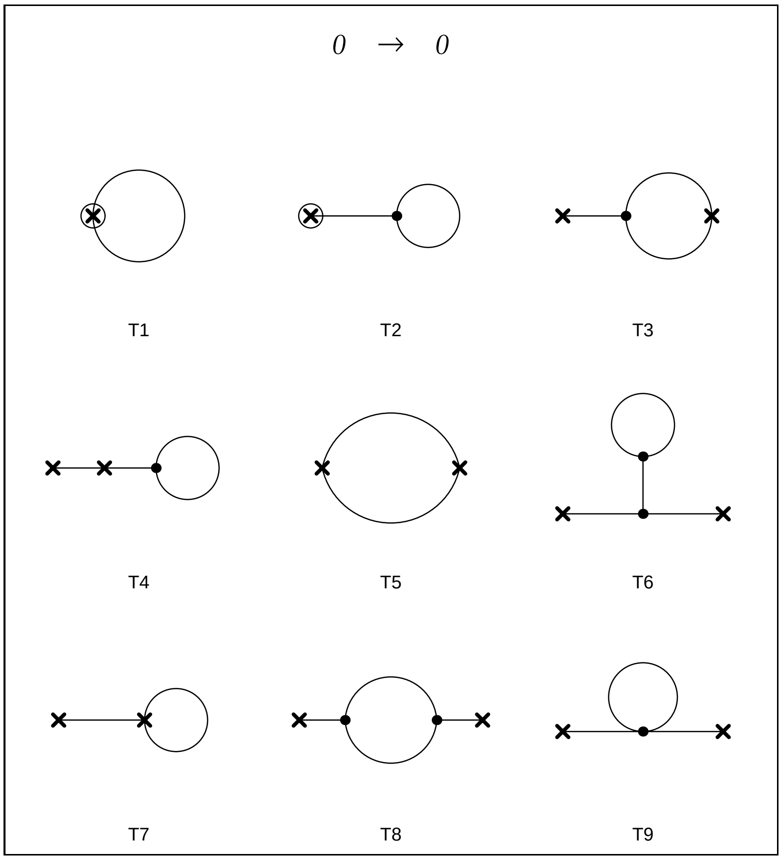
3.6 Selecting Starting Topologies

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 combinatorial 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[Internal][Vertex[3][-2], Vertex[3][-1]],
    Propagator[Loop[1]][Vertex[3][-2], Vertex[3][-2]],
    Propagator[Loop[2]][Vertex[3][-1], Vertex[3][-1]] ]
]
```

4 Inserting Fields into Topologies

4.1 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

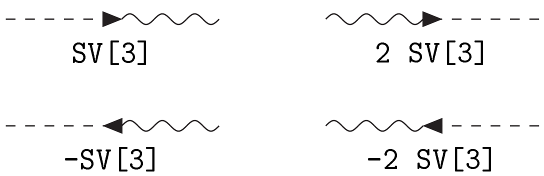
Classes fields represent sets of fields with common properties such as behaviour under charge conjugation. A class is a generic field type with a class number, e.g. `F[1]`. The class specifies which further indices (if any) the class members possess and the range of these indices.

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 $H\ell_i\ell_j$ coupling in the electroweak Standard Model (Higgs–lepton–lepton with generation indices i and j, i.e. $\ell_1 = e$, $\ell_2 = \mu$, $\ell_3 = \tau$) they happen to be identical:

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

Finally, at particles level, the generation indices i and j are resolved, so for instance the $He\mu$ 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.

4.2 `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`, ...} -> {`o_1`, `o_2`, ...}]`	
	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];
```

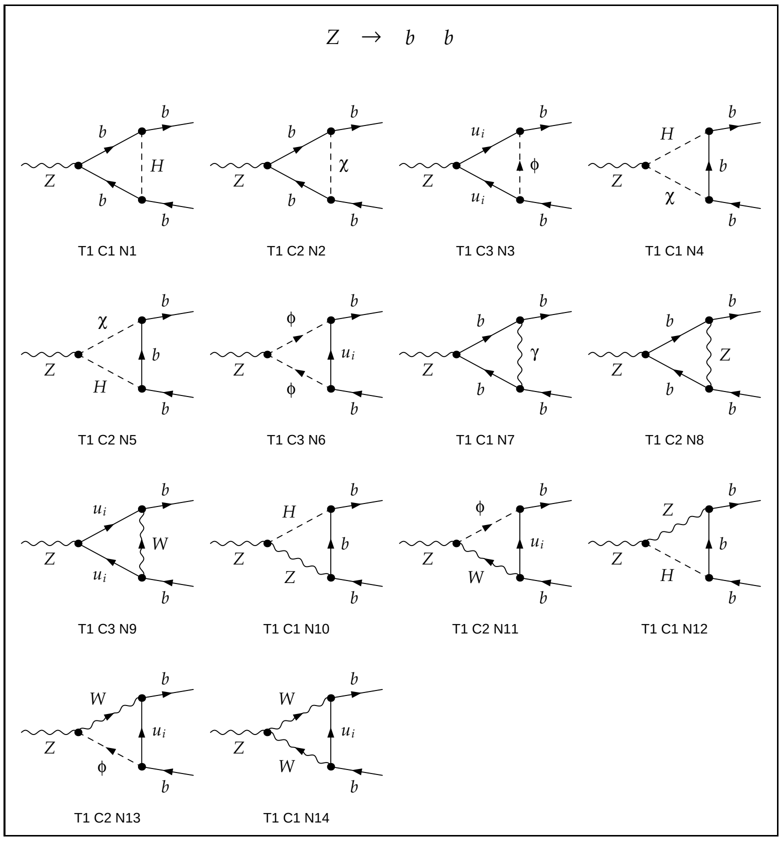
Insert $Z \to b\bar{b}$.

```
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. 4.1)
`GenericModel`	`"Lorentz"`	generic model to use
`Model`	`"SM"`	classes model to use
`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).

4.3 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.

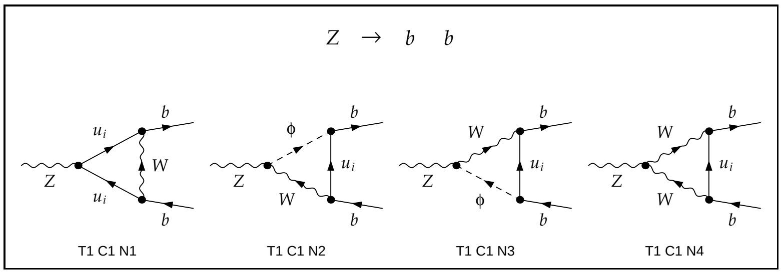
While `LastSelections` may seem more general, it works by first generating all diagrams and *afterwards* selecting those that match the given criteria (hence the name). In contrast, `ExcludeParticles` and `ExcludeFieldPoints` work by eliminating particles and couplings *before* starting the insertion 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]`



4.4 Selecting Insertions

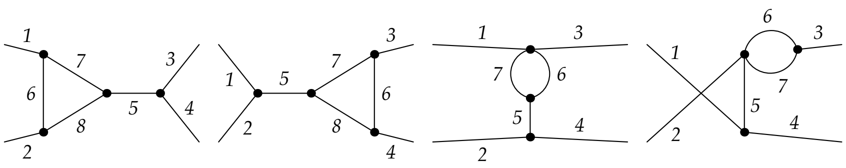
One further way to pick a selected set of diagrams is to use the function `DiagramSelect`.

`DiagramSelect[`*d*`,` *crit*`]`	select the diagrams from *d* for which *crit* gives `True`.

This function works much like the usual `Select` of *Mathematica*, i.e. it applies a test function to every diagram, and returns only those for which the result is `True`. The test function can of course be any *Mathematica* expression, but the most common usage is something like

```
DiagramSelect[ diags, FreeQ[#, Field[5] -> S]& ]
```

which eliminates all diagrams with a scalar particle on the fifth propagator. Such a statement would not be very useful were it not for the fact that *FeynArts* orders its propagators in a very systematic way. This is best exploited if the topologies are grouped into categories like self-energies, vertices, or boxes. For example, in a $2 \to 2$ vertex-correction diagram the first four propagators are those of the external particles, the fifth is the propagator of the tree part, and the rest are the loop propagators, e.g.

 etc.

The important thing to note here is that the same “parts” of a diagram (say, the loop) in general get the same propagator numbers. Nevertheless, the number of a certain propagator is not fixed a priori, so the only reliable way to find the propagator numbers is to paint the bare topologies with the option `FieldNumbers -> True` (see Sect. 5).

4.5 Discarding Insertions

If the various ways of restricting `InsertFields` are not sufficient to select the desired diagrams, the user may discard the superfluous ones by number. The function for this is `Discard`.

`Discard[`*t*`,` *sel*`]`	discard the diagrams *sel* from the list of inserted topologies *t*. *sel* can be e.g. `5, 10...12, 20` which discards diagrams 5, 10, 11, 12, and 20.

The numbers referred to are the sequential numbers of the diagrams as given by `Paint`. Note that if `Paint` is used with a `PaintLevel` different from the `InsertionLevel`, the numbering will not be useful for `Discard`. Discarding a diagram at a particular level removes derived diagrams at deeper levels. For other kinds of objects, `Discard` works like an extended version of `Drop`, e.g. `Discard[{a, b, c, d, e}, 2...4]` results in `{a, e}`.

4.6 Structure of the Inserted Topologies

If one wants to perform more advanced operations on inserted topologies it is necessary to know their structure. Essentially, a topology gets enclosed in a hierarchy of rules with particles insertions nested inside classes insertions nested inside generic insertions. It is possible to select specific levels out of this hierarchy with `PickLevel` (see Sect. 6.4).

```
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
]
```

5 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. 4.1)
`ColumnsXRows`	`3`	number of diagrams per column/row, may also be given as a list `{`n_c`,` n_r`}`
`SheetHeader`	`Automatic`	title for each sheet of graphics
`Numbering`	`Full`	what type of numbering to display underneath each diagram
`FieldNumbers`	`False`	whether to label the propagators of a bare topology with the field numbers
`AutoEdit`	`True`	whether to call the topology editor when encountering an unshaped topology
`DisplayFunction`	`$DisplayFunction`	which function to apply to the final graphics object in order to display it

`PaintLevel` specifies the level at which diagrams are painted. The default is to use the level the topologies were inserted at (for inserted topologies). For bare topologies `PaintLevel` is not relevant. Note that for the numbering of the diagrams to be appropriate for discarding insertions (see Sect. 4.5), the `PaintLevel` has to be the same as the `InsertionLevel`.

`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.

`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 -> None` omits the default numbering of the diagrams. The default setting of `Numbering -> Full` places diagram numbers of the form `T1 C8 N15` (= topology 1, classes insertion 8, running number 15) underneath the diagram. A simple numbering which is useful in publications is `Numbering -> Simple` in which case just the running number is used.

The propagators of a bare topology are usually unlabelled. With `FieldNumbers -> True` `Paint` uses the field numbers (the n in `Field[`n`]`) as labels. This is useful when selecting diagrams with `DiagramSelect` (see Sect. 4.4).

`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.

`DisplayFunction` determines how the output of `Paint` is rendered. The default is to show the graphics on screen. Popular choices are `DisplayFunction -> Identity` which does not render the graphics at all, or `DisplayFunction -> (Display["file.ps", #]&)` which saves it in a file.

When running in the standalone *Mathematica* Kernel, `Paint` automatically animates the screen graphics, with the sheets of output acting as the frames of the animation. This is to inhibit the cluttering of the screen with too many windows. The speed at which the frames are displayed by default is rather high, more appropriate for a movie than for viewing sheets of graphics. The frame rate can be adjusted in the "Animation Control" menu, which also allows to view single frames. Under Unix, the initial frame rate can be set with the X resource `Motifps*frameTime`, e.g. in `.Xdefaults`:

```
Motifps*frameTime: 1000
```

5.1 Things to do with the Paint Output

The output of `Paint` is a `FeynArtsGraphics` object whose arguments represent the sheets of the output. Each of these sheets is represented by a matrix which determines the layout of the diagrams, i.e.

```
FeynArtsGraphics[title][
    { {g11, g12, ...},
      {g21, g22, ...},          } sheet 1
      ...            },
    ...
]
```

These `FeynArtsGraphics` objects can be rendered with the usual *Mathematica* rendering functions `Export`, `Display`, and `Show`. `Export` and `Display` allow the image size to be changed with the `ImageSize` option whose default value is `72*{6, 7}` (6×7 inches) for a `FeynArtsGraphics` object.

In addition to the standard formats understood by *Mathematica*, the rendering functions accept two more output formats, `"PS"` and `"TeX"`. To run output produced with the `"TeX"` format through LaTeX, the `feynarts.sty` style file is required; it is located in the *FeynArts* directory.

One particular feature of the `"TeX"` output is that it can be touched up quite easily. This is useful for publications, e.g. when grouping together diagrams. To start with, the generated LaTeX code doesn't look too scary, for instance a single diagram might be drawn by

```
\FADiagram{T1 C1 N2}
\FAProp(0.,15.)(6.,10.)(0.,){/Sine}{0}
\FALabel(2.48771,11.7893)[tr]{$Z$}
\FAProp(0.,5.)(6.,10.)(0.,){/Sine}{0}
\FALabel(3.51229,6.78926)[tl]{$Z$}
\FAProp(20.,15.)(14.,10.)(0.,){/Sine}{0}
\FALabel(16.4877,13.2107)[br]{$Z$}
\FAProp(20.,5.)(14.,10.)(0.,){/Sine}{0}
\FALabel(17.5123,8.21074)[bl]{$Z$}
\FAProp(6.,10.)(14.,10.)(0.,){/ScalarDash}{0}
```

```
\FALabel(10.,9.18)[t]{$H$}
\FAVert(6.,10.){0}
\FAVert(14.,10.){0}
```

With this representation, it is pretty straightforward to move diagrams around as they always start with `\FADiagram`. (Incidentally, the PostScript files generated by *FeynArts* have a very similar markup.)

The `feynarts.sty` introduces the following new LaTeX commands:

`\begin{feynartspicture}`	
`\end{feynartspicture}`	delimit a sheet of diagrams
`\FADiagram{`*t*`}`	advance to the next diagram, which has title *t*
`\FAProp(`*f*`)(`*t*`)(`*c*`){`*g*`}{`*a*`}`	draw a propagator from point *f* to point *t* with curvature *c* using graphical representation *g* and arrow *a*
`\FAVert(`*p*`){`*cto*`}`	draw a vertex of counter-term order *cto* at point *p*
`\FALabel(`*p*`)[`*align*`]{`*text*`}`	write *text* at point *p* with alignment *align*

Do not be confused by the multitude of parameters: The only command likely to be edited is `\FALabel`. Suffice it to say that each diagram is drawn on a 20×20 grid with the origin in the lower left corner and the positive axes extending to the right and upward, and that the alignment is specified in the standard TeX manner, i.e. with combinations of `t`, `b`, `l`, and `r` for top, bottom, left, and right alignment, respectively.

The exact details of specifying the other parameters shall not be discussed here, as they reflect the choices made when editing the topology. That is to say that should it become necessary to edit `\FAProp` or `\FAVert` statements, it is advisable to re-edit the particular topology instead.

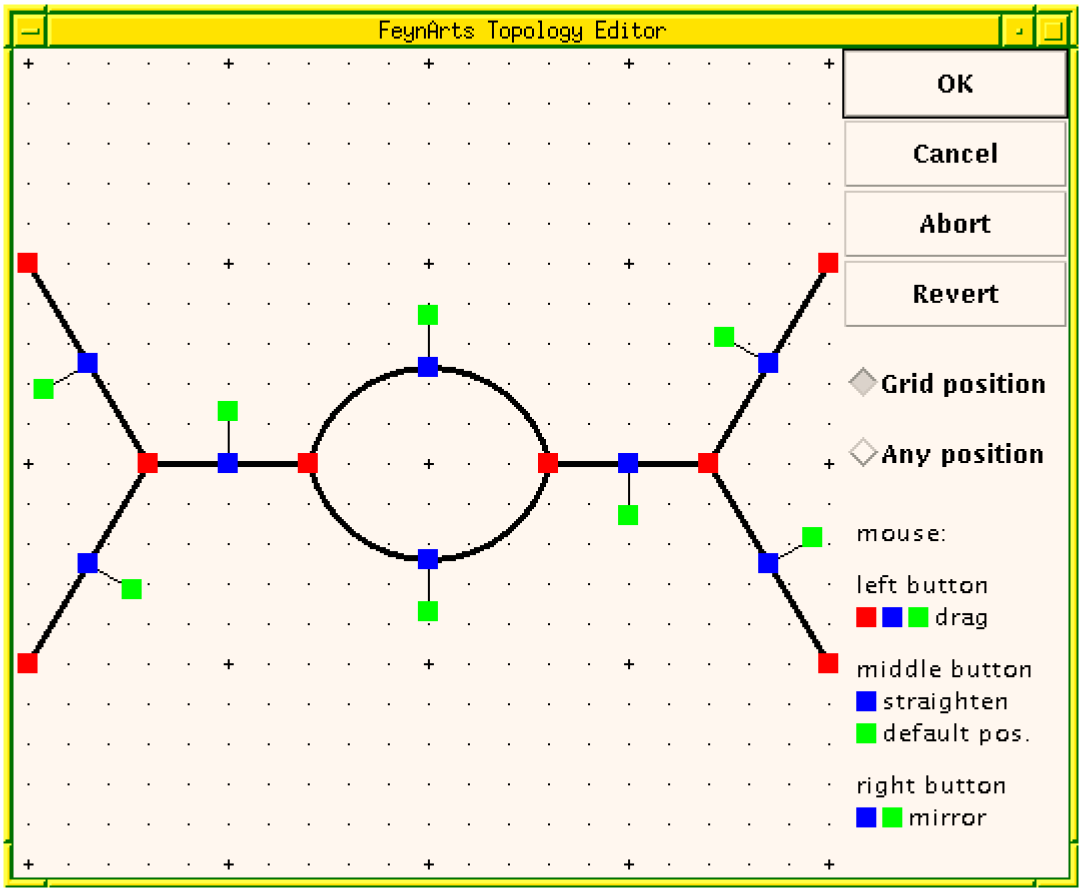
5.2 Shaping Topologies

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:

The OK button commits the changes and exits.

The CANCEL button discards the changes made to the current diagram and exits the editor but continues editing more diagrams if a list of topologies is being shaped.

The ABORT button is similar to CANCEL, but aborts the editing process altogether, i.e. it returns to the *Mathematica* prompt immediately, even if there are more topologies in line to be edited.

The REVERT button reverts to the initial layout of the topology.

Choosing GRID POSITION allows dragging of the squares only onto grid points. Conversely, ANY POSITION allows a square to be dragged to an arbitrary position.

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 Java, `TopologyEditor.java`, that communicates with *Mathematica* through the J/Link program. See Sect. 1 for setting up Java and J/Link.

6 Creating the Analytic Expressions

6.1 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`
`GraphID[`*id*`]`	identifier 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. 6.4).

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. 7 for details)
`FourMomentum[`t`, `n`]`	nth momentum of type t (`Incoming`, `Outgoing`, 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 $o_1, o_2, \ldots$
`VertexFunction[`o`][`f`]`	generic vertex function of loop order o with adjoining fields f
`SumOver[`i`, `r`, `ext`]`	indicates that the amplitude it is multiplied with is to be summed in the index i over the range r; if r is an integer, it represents the range $1 \ldots r$. ext is the symbol `External` if the summation is over an index belonging to an external particle[†]

6.2 `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

[†] It is necessary to distinguish sums over internal and external indices. For example, a squared diagram is calculated by squaring the diagram and then summing over the external indices. On the other hand, an internal index summation (say, for a quark loop) must be done for each diagram before squaring.

is CreateFeynAmp.

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

option	*default value*	
AmplitudeLevel	InsertionLevel	level specification (see end of Sect. 4.1)
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
MomentumConservation	True	whether to enforce momentum conservation at each vertex

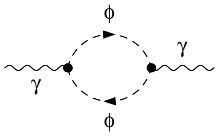
The default for AmplitudeLevel is to use the same level the topologies were inserted at.

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.

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 -> {},
    Process -> {{V[1], p1, 0}} -> {{V[1], k1, 0}}][
  FeynAmp[
    GraphID[Topology == 1, Generic == 1],
    Integral[q1],
    (I/32 RelativeCF
      FeynAmpDenominator[1/(q1^2 - Mass[S[Gen3]]^2),
        1/((-p1 + q1)^2 - Mass[S[Gen4]]^2)]
      (p1 - 2 q1)[Lor1] (-p1 + 2 q1)[Lor2]
      ep[V[1], p1, Lor1] ep*[V[1], k1, Lor2]
      G(0)SSV[(Mom[1] - Mom[2])[KI1[3]]]
      G(0)SSV[(Mom[1] - Mom[2])[KI1[3]]]) / Pi^4,
    {Mass[S[Gen3]], Mass[S[Gen4]],
      G(0)SSV[(Mom[1] - Mom[2])[KI1[3]]],
      G(0)SSV[(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 -> {},
  Process -> {{V[1], FourMomentum[Incoming, 1], 0}} ->
    {{V[1], FourMomentum[Outgoing, 1], 0}}][
FeynAmp[
  GraphID[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}] ] ]
```

6.3 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* combinatorical *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)[Lor1]` and `(-p1 + 2 q1)[Lor2]` 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. `Lor1` and `Lor2` 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, Lor1]` and `ep`*`[V[1], k1, Lor1]` 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[Gen3]] -> MW
```
full: `Mass[S[Index[Generic, 3]]]`

Mass[S[Gen4]] -> MW
full: Mass[S[Index[Generic, 4]]]

$\texttt{G}^{(0)}_{\texttt{SSV}}$[(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]]]

$\texttt{G}^{(0)}_{\texttt{SSV}}$[(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

6.4 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 GraphID. 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 differently. 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.

6.5 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, ...} ->
              Insertions[Particles][
                {MLE[1], ...}, {MLE[2], ...}, more particles insertions
              ],
          {MQU, ...} -> ...,
          more classes insertions
        ],
  ],
  more amplitudes
]
```

6.6 On Fermion Chains and Signs

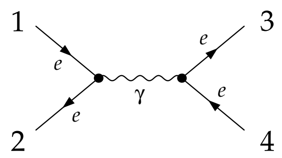
One of the most asked-for details of amplitude generation is how the signs of fermionic diagrams are determined.

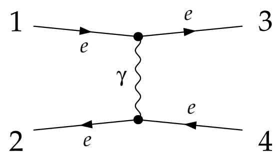
FeynArts generally uses the methods from [DeEHK92]. That is basically, the fermion flow is fixed at generic level and so-called flip rules are applied if it turns out that in the classes or particles insertion the flow is reversed. Open fermion chains—the ones terminating on external fermions—are ordered 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 $\{2–1, 3–4\}$ in the left diagram, and as $\{3–1, 2–4\}$ 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 is 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 than 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.

6.7 Specifying Momenta

`CreateFeynAmp` can be made to use a certain momentum on a certain propagator. This is done by editing the inserted topologies and appending a fourth argument to the propagators whose momenta one wants to fix. (A convenient way to do so is to assign the inserted topologies to a variable and use the *Mathematica* function `EditDefinition[`*var*`]`.)

This fourth argument will become the momentum's name. For example, the momentum assigned to the propagator

```
Propagator[Internal][Vertex[3][101], Vertex[3][100], Field[4], Q1]
```

will be `FourMomentum[Internal, Q1]`. There are two rules to observe: Do not use an integer as a name to avoid conflict with the automatically-generated momenta, and do not change the direction of the propagator! If you need the momentum flowing opposite to the direction of the propagator, give it a minus sign, e.g. `-Q1`.

While `CreateFeynAmp` takes care to eliminate automatically-generated momenta before user-specified ones when working through momentum conservation, one can prohibit the elimination of any momentum with the option `MomentumConservation -> False`.

6.8 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.

7 Definition of a New Model

7.1 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. 6.6).

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

For example, in the usual representation of the Poincaré 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]] ] ]
```

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`. By convention, all fields in `AnalyticalCoupling` are incoming.

`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

$$\begin{aligned} C\big(V^i_\mu(p_1), V^j_\nu(p_2), V^k_\rho(p_3), V^\ell_\sigma(p_4)\big) &= G^{(1)}_{V_iV_jV_kV_\ell}\, g_{\mu\nu}g_{\rho\sigma} + G^{(2)}_{V_iV_jV_kV_\ell}\, g_{\mu\rho}g_{\nu\sigma} + G^{(3)}_{V_iV_jV_kV_\ell}\, g_{\mu\sigma}g_{\rho\nu} \\ &= \vec{G}_{V_iV_jV_kV_\ell} \cdot \begin{pmatrix} g_{\mu\nu}g_{\rho\sigma} \\ g_{\mu\rho}g_{\nu\sigma} \\ g_{\mu\sigma}g_{\rho\nu} \end{pmatrix} . \end{aligned}$$

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,

$$C\big(V^i_\mu(p_1), V^j_\nu(p_2), V^k_\rho(p_3)\big) = G_{V_iV_jV_k}\left(g_{\mu\nu}(p_2 - p_1)_\rho + g_{\nu\rho}(p_3 - p_2)_\mu + g_{\rho\mu}(p_1 - p_3)_\nu\right) .$$

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.

The algorithm behind this was developed in [DeEHK92] and its main advantage is that it can deal with fermion-number violating vertices, such as appear in supersymmetric models. The idea is that instead of ordering the fermion propagators according to the given fermion flow, one *chooses* a direction for the fermion lines. (Obviously one cannot define the fermion flow properly as soon as fermion-number violating couplings are present, which is why the standard method breaks down in that case.) If later it turns out that the chosen direction is opposite to the actual fermion flow, one has to apply a so-called flipping rule. The flipping rule is nothing but a charge conjugation, i.e.

$$\Gamma \xrightarrow{\text{flip}} C\Gamma^T C^{-1}$$

where Γ is some product of Dirac matrices and C is the charge-conjugation matrix. The flipping rules for all possible objects that can appear in Γ have been worked out in [DeEHK92] and the only necessary substitution is

$$\gamma_\mu \omega_\pm \xrightarrow{\text{flip}} -\gamma_\mu \omega_\mp$$

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.

7.2 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`),
- a function that detects if a vertex violates quantum number conservation (`ViolatesQ`),
- 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}
```

If a particular index should not be "unfolded" at particles level (i.e. *FeynArts* should not generate an extra diagram for every value the index can take on), its range may be wrapped in `NoUnfold`. For instance, one usually does not want eight diagrams generated and drawn for every gluon propagating; this can be effected with

```
IndexRange[ Index[Gluon] ] = NoUnfold[ Range[8] ]
```

As a consequence, fields with specific values for such indices have no effect in places like `ExcludeParticles`. This means that it is possible to exclude either all gluons or none, but not only the gluon with index 5.

Conservation of Quantum Numbers

When *FeynArts* generates diagrams with vertex-function placeholders (see Sect. 3.4), it eliminates illegal ones by checking whether the quantum numbers of the fields joining at the vertex are conserved. To this end, it calls the function `ViolatesQ` for every possible 1PI vertex function. `ViolatesQ` receives as arguments the quantum numbers of the involved fields (times -1 for antiparticles) and must return `True` if the vertex violates the conservation of those quantum numbers. In the most common case of charge-like (additive) quantum numbers, the condition is that the sum of all quantum numbers does not vanish at the vertex, which can be coded simply as

```
ViolatesQ[ q__ ] := Plus[q] =!= 0
```

However, this is not true for all kinds of quantum numbers (R-parity for example), which is why the definition of `ViolatesQ` is given in the model file.

Note the logic: all vertices for which `ViolatesQ` does not explicitly return `True` are accepted. It is no error if there is no definition for `ViolatesQ` since that simply means that all 1PI vertex functions are allowed.

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,
      QuantumNumbers -> 2/3 Charge,
      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
`QuantumNumbers`	`{}`	quantum numbers the field carries
`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
```

`QuantumNumbers` specifies which quantum numbers the field carries. In principle, any *Mathematica* expression can be used as a quantum number, though the more common choices are multiples of a symbol, e.g. `2/3 Charge`. An important point is that the quantum number of the antiparticle is minus the quantum number of the particle. This is just what one wants in the case of additive quantum numbers like charge, since indeed if the up-quark has `2/3 Charge`, the anti-up-quark has `-2/3 Charge`. However, it is for instance not true that if a particle has R-parity 1, its antiparticle has R-parity -1. In that case, one has to choose `RParity[±1]` instead of `±RParity` and supply an appropriate `ViolatesQ` function.

`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 `PropagatorType` which can take the values `Straight`, `Sine` (photon-like), `Cycles` (gluon-like), `ScalarDash`, or `GhostDash`.

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. The convention is that all fields at a coupling are incoming, as in the generic model file.

`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
`CC[`f_1`, `f_2`, ...] == ...`	the same, except that simultaneously the charge-conjugated vertex is defined
`ConjugateCoupling[`f`][`c`]`	defines how the charge-conjugate of the expression c is derived for the coupling of the fields f
`KinematicVector[`f`]`	returns the kinematic vector for the coupling of the fields f

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.

The conjugated coupling (the part of the Lagrangian usually abbreviated as "+ h.c." in textbooks) can either be entered directly, or by using `CC` instead of `C` in the definition of the coupling; e.g. in the Standard Model one could define both the $\bar{u}dW^+$ and the $\bar{d}uW^-$ vertex, or define either one using `CC`. However, when using the `CC` method, one has to specify how the coupling vector of the conjugated coupling is derived from the original one. This is done by defining the function `ConjugateCoupling`. `ConjugateCoupling` is applied to all elements of the coupling vector and is in general different from a plain `Conjugate`, for instance the i from the exponent of the path integral (i.e. $\mathrm{i}\int \mathrm{d}^4x\mathcal{L}$) must not be conjugated. The conjugation procedure can depend on the kinematic structure of the vertex, e.g. couplings involving a ∂_μ in configuration space usually get an additional minus which comes from the Fourier transformation. The components of the kinematic vector can be obtained from the field information handed to `ConjugateCoupling` with the function `KinematicVector`.

A drawback is that `ConjugateCoupling` is a rather dumb function which has to be taught how to conjugate every symbol, so it is probably less work to enter the conjugated couplings directly if only a few are affected. Giving no definition for `ConjugateCoupling` is not an error: the amplitudes will then simply contain the symbol `ConjugateCoupling`.

"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 Poincaré 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*.

The following generic couplings are defined in `Lorentz.gen`. Antisymmetric couplings are labelled by a subscript $-$, symmetric ones by a subscript $+$.

$$C\big(V_\mu, V_\nu, V_\rho, V_\sigma\big) = \vec{G}_{VVVV} \cdot \begin{pmatrix} g_{\mu\nu} g_{\rho\sigma} \\ g_{\mu\rho} g_{\nu\sigma} \\ g_{\mu\sigma} g_{\nu\rho} \end{pmatrix}_+ \tag{VVVV}$$

$$C\big(V_\mu(k_1), V_\nu(k_2), V_\rho(k_3)\big) = \vec{G}_{VVV} \cdot \big(g_{\mu\nu}(k_2 - k_1)_\rho + g_{\nu\rho}(k_3 - k_2)_\mu + g_{\rho\mu}(k_1 - k_3)_\nu\big)_- \tag{VVV}$$

$$C\big(S, S, S, S\big) = \vec{G}_{SSSS} \cdot \big(1\big)_+ \tag{SSSS}$$

$$C\big(S, S, S\big) = \vec{G}_{SSS} \cdot \big(1\big)_+ \tag{SSS}$$

$$C\big(S, S, V_\mu, V_\nu\big) = \vec{G}_{SSVV} \cdot \big(g_{\mu\nu}\big)_+ \tag{SSVV}$$

$$C\big(S(k_1), S(k_2), V_\mu\big) = \vec{G}_{SSV} \cdot \big((k_1 - k_2)_\mu\big)_- \tag{SSV}$$

$$C\big(S, V_\mu, V_\nu\big) = \vec{G}_{SVV} \cdot \big(g_{\mu\nu}\big)_+ \tag{SVV}$$

$$C\big(F, F, V_\mu\big) = \vec{G}_{FFV} \cdot \begin{pmatrix} \gamma_\mu \omega_- \\ \gamma_\mu \omega_+ \end{pmatrix}_+ \tag{FFV}$$

$$C\big(F, F, S\big) = \vec{G}_{FFS} \cdot \begin{pmatrix} \omega_- \\ \omega_+ \end{pmatrix}_+ \tag{FFS}$$

$$C\big(U(k_1), U(k_2), V_\mu\big) = \vec{G}_{UUV} \cdot \begin{pmatrix} k_{1\mu} \\ k_{2\mu} \end{pmatrix}_+ \tag{UUV}$$

$$C\big(S, U, U\big) = \vec{G}_{SUU} \cdot \big(1\big)_+ \tag{SUU}$$

$$C\big(V_\mu(k_1), V_\nu(k_2)\big) = \vec{G}_{VV} \cdot \begin{pmatrix} g_{\mu\nu}(k_1 k_2) \\ g_{\mu\nu} \\ k_{1\mu} k_{2\nu} \end{pmatrix}_+ \tag{VV}$$

$$C\big(S(k_1), S(k_2)\big) = \vec{G}_{SS} \cdot \begin{pmatrix} (k_1 k_2) \\ 1 \end{pmatrix}_+ \tag{SS}$$

$$C\big(F(k_1), F(k_2)\big) = \vec{G}_{FF} \cdot \begin{pmatrix} \not{k}_1 \omega_- \\ \not{k}_2 \omega_+ \\ \omega_- \\ \omega_+ \end{pmatrix}_- \tag{FF}$$

$$C\big(S(k_1), V_\mu(k_2)\big) = \vec{G}_{SV} \cdot \begin{pmatrix} k_{1\mu} \\ k_{2\mu} \end{pmatrix}_+ \tag{SV}$$

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].

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`.

`SM.mod` defines the following `Restrictions`:

`NoGeneration1`	exclude generation-1 fermions (ν_e, e, u, d)
`NoGeneration2`	exclude generation-2 fermions (ν_μ, μ, c, s)
`NoGeneration3`	exclude generation-3 fermions (ν_τ, τ, t, b)
`NoElectronHCoupling`	exclude all couplings involving electrons and a Higgs ($e^- e^+ H$, $e^- e^+ \chi$, $e^- \nu_e \phi^-$).
`NoLightFHCoupling`	exclude all couplings between light fermions (all fermions except the top) and Higgs fields ($f_i \bar{f}_i H$, $f_i \bar{f}_i \chi$, $\ell_i^- \nu_i \phi^-$, $\bar{d}_i u_i \phi^-$, $f_i \neq t$, $u_i \neq t$).
`NoQuarkMixing`	exclude all couplings where off-diagonal elements of the quark mixing matrix appear ($\bar{d}_i u_j W^-$, $\bar{d}_i u_j \phi^-$, $i \neq j$). Note that the diagonal elements `CKM[`i, i`]` are nevertheless present.
`QEDOnly`	exclude all particles except the massive fermions, the photon, and the photon ghost.

B.1 The QCD Extension

The addition of QCD to the electroweak Standard Model as described in the last section requires two things:

First, it is necessary to give the quarks explicit colour indices because of the non-trivial colour structures encountered in QCD. This is done in `SMc.mod` which is identical to `SM.mod` except that the quark classes `F[3]` and `F[4]` carry two indices, `Generation` and `Colour`.

The model file `SMQCD.mod` then actually adds QCD to the electroweak sector. In fact, it just loads `SMc.mod` and appends the gluon and its ghost to the appropriate definitions.

Following is the list of additional symbols used in `SMQCD.mod`.

`GS`	the strong coupling constant
`SUNT[`a, i, j`]`	the generators of SU(N), $(T^a)_{ij}$
`SUNF[`a, b, c`]`	the SU(N) structure constants f^{abc}
`SUNF[`a, b, c, d`]`	a short-hand for the sum $\sum_i f^{abi} f^{icd}$

The additional particles are the gluon and its ghost. The gluon index is not expanded out at particles level, i.e. *FeynArts* does not generate eight diagrams for every gluon.

class	*self-conjugate*	*indices*	*members*		*mass*
`V[5]`	yes	`Gluon`	`V[5,{`i`}]`	g_i	`0`
`U[5]`	no	`Gluon`	`U[5,{`i`}]`	u_{g_i}	`0`

B.2 Background-field Formalism

The model file `SMbgf.mod` contains the electroweak Standard Model in the background-field formalism. It is based on [DeDW95], with the exception that the renormalization of the fermionic fields proceeds as in `SM.mod`, i.e. with separate renormalization constants for upper and lower components of the fermion doublet, which in turn means that fermions do not need an external wave-function renormalization. The larger part of `SMbgf.mod` is in fact derived from `SM.mod`. (Interested readers may want to look at `MakeSMbgfmod.m`.) At present, vertices containing quantum fields other than fermions do not possess counter-terms, but this is sufficient for one-loop calculations.

To work with `SMbgf.mod`, one needs to specify `Lorentzbgf.gen` as the generic model, which is a slightly generalized version of `Lorentz.gen`.

The following table lists the background fields and their quantum-field counterparts. All other fields are the same as in `SM.mod`.

background field		quantum field	
`V[10]`	$\hat{\gamma}$	`V[1]`	γ
`V[20]`	$\hat{Z}$	`V[2]`	Z
`V[30]`	$\hat{W}^-$	`V[3]`	W^-
`S[10]`	$\hat{H}$	`S[1]`	H
`S[20]`	$\hat{\chi}$	`S[2]`	χ
`S[30]`	$\hat{\phi}^-$	`S[3]`	ϕ^-
`SV[20]`§	$\hat{\chi}$–$\hat{Z}$	`SV[2]`§	χ–Z
`SV[30]`§	$\hat{\phi}^-$–$\hat{W}^-$	`SV[3]`§	ϕ^-–W^-

§Must be enabled with `$SVMixing = True`.

C Changes in Version 3

Although *FeynArts* 3 has kept the concept and much of the syntax of *FeynArts* 2.2, some incompatible changes were necessary:

The most significant changes have been made to the `Paint` function, where the options `Destination`, `FileBaseName`, and `TeXLabels` have disappeared in favour of the more *Mathematica*-like `DisplayFunction` option, together with the increased functionality of *Mathematica*'s rendering functions `Export`, `Display`, and `Show`.

Similarly, the `ProcessName` identification added to topologies by `InsertFields` has been scrapped because its only purpose was to provide a default file name for the graphics saved by `Paint` which is now no longer necessary, owing to the above-mentioned changes. The only other function which used `ProcessName`, `GraphName`, does no longer carry a process name and has been renamed to `GraphID`, so as not to confuse programs that were used to the old syntax. When converting the amplitudes back to the *FeynArts* 1 format with `ToFA1Conventions`, however, the `GraphID` is replaced again with `GraphName` for compatibility.

It is no longer admissible to choose a particle's `PropagatorType` in the definition of a model file to be an arbitrary *Mathematica* graphics primitive. Only `Straight`, `Sine`, `Cycles`, `ScalarDash`, and `GhostDash` are possible.

With the new `CreateVFTopologies` function, topologies with vertex-function placeholders are treated much more intrinsically as in version 2.2, hence the `VertexFunctions` option of `InsertFields` is gone.

`CreateTopologies` now sorts the propagators inside the topologies as well as the topologies themselves into a canonical order, which has two consequences:

- The ordering of the diagrams will in all likelihood be different than in version 2.2. This affects functions that pick diagrams by number, such as `Discard`.
- Existing shapes for topologies can no longer be used directly but must be converted to the new format. To this end, copy your old `GraphInfo` directory into the *FeynArts* 3 tree, start *Mathematica*, and run the program `ConvertGraphInfos.m`. This will first back up the old GraphInfos in `GraphInfo/backup` and then convert them to the new format.

The `HoldTimes` option of `CreateFeynAmp` which was intended for debugging purposes but was never used has been removed completely.

The model file directory, formerly specified with `$ModelDir`, has been supplanted with `$ModelPath`, which is a list of directories that *FeynArts* searches when looking for a model file.

Acknowledgements

The new graphics routines have been developed on a Visiting Scholar grant of Wolfram Research, Inc., and I am grateful in particular to Michael Malak and Lou d'Andria for sharing their knowledge of *Mathematica* with me. Thanks also to the Ph.D. students at Karlsruhe, especially Christian Schappacher, for relentless beta testing,

References

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

[DeDW95] A. Denner, S. Dittmaier, and G. Weiglein, *Nucl. Phys.* **B440** (1995) 95.

[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.

Index

.Xdefaults, 28
$DisplayFunction, 28
$ExcludeTopologies, 14
$FermionLines, 42
$ModelDir, 67
$ModelPath, 67
$ModifiedGraphInfos, 32
$SVMixing, 19, 63
$TopologyDataDir, 32
$Verbose, 6
1PI, 11, 52

ABORT, 32
accent on label, 55
Adjacencies, 10
adjacency, 8, 10
alignment, 30
All, 10
AllBoxCTs, 12
AllBoxes, 12
Alph, 55
amplitude, 33
    structure of, 40
AmplitudeLevel, 35, 39
AnalyticalCoupling, 46
AnalyticalPropagator, 44
animation, 28
antiparticle, 18
ANY POSITION, 32
Appearance, 55
attributes
    classes, 52
AutoEdit, 27, 28, 30
Automatic, 22
autoshaping, 28

background field, 65
bar on label, 55
Bicycle, 15
BoxCTs, 12
Boxes, 12
buttons, 32

C, 57
CANCEL, 32
CC, 57
Centre, 14
changes in *FeynArts* 3, 66
character conversion, 55
Charge, 54
charge conjugation, 18
chirality projector, 19, 48
ChiralityProjector, 60
CKM, 62
Classes, 18
classes model, 22, 51
closure, 47
ColumnsXRows, 27
combinatorial factor, 38
compatibility, 42, 66
ComposedChar, 55
composite label, 55
Conjugate, 50
ConjugateCoupling, 57
conjugation, 59
conservation of quantum numbers, 12, 52, 54

conventions (*FeynArts* 1), 42
`ConvertGraphInfos.m`, 66
counter-terms, 10, 57
  order of, 10
coupling
  antisymmetric, 47
  classes, 56
  closure, 47
  conjugated, 59
  constant, 46
  generic, 46
  kinematical structure, 19
  quartic gauge-boson, 46, 57
  structure, 38
  tree-level, 57
  triple gauge-boson, 47
  vector, 46, 56
`CreateCTTopologies`, 10
`CreateFeynAmp`, 34
`CreateTopologies`, 9
`CreateVFTopologies`, 11
`CT`, 15
`CTOrder`, 10
`CW`, 62
`Cycles`, 54

`dCKM1`, 63
`dCW1`, 63
`define`, 16
`Destination`, 66
`DiagramSelect`, 24
`DiracMatrix`, 49, 60
`DiracSlash`, 49, 60
`DiracSpinor`, 46, 49, 60
directory contents, 6
`Discard`, 25, 66
`Display`, 29
`DisplayFunction`, 27
`dMf1`, 63
`dMHsq1`, 63
`dMWsq1`, 63
`dMZsq1`, 63
drawing diagrams, 27
`dSW1`, 63
`dZAA1`, 63
`dZAZ1`, 63
`dZe1`, 63
`dZfL1`, 63
`dZfR1`, 63
`dZH1`, 63
`dZW1`, 63
`dZZA1`, 63
`dZZZ1`, 63

`Eight`, 15
`EL`, 62
electroweak Standard Model, 62
`ExcludeFieldPoints`, 22
`ExcludeParticles`, 22
`ExcludeTopologies`, 10, 12
`Export`, 29
`External`, 8, 34, 44, 54
external
  legs, 9
  wave functions, 35

`F`, 18
`\FADiagram`, 30
`\FALabel`, 30
`\FAProp`, 30
`\FAVert`, 30
fermion, 18
  amplitude, 41

flip rules, 47
flow, 41
Majorana, 60
self-conjugate, 60
signs, 41
type, 63
FermionChain, 34
FeynAmp, 33
FeynAmpDenominator, 34, 38
FeynAmpList, 33
FeynArts 1, 42
feynarts.sty, 29
FeynArtsGraphics, 29
feynartspicture, 30
Feynman gauge, 35
Field, 24
field, 18
background, 65
derived, 22
exclusion, 22
extended structure, 45, 46
insertion, 18
intermediate, 45
level, 18
lower level, 22
mixing, 18
FieldNumbers, 24, 27
FileBaseName, 66
filter, 12
function, 14
final-touch rules, 50, 59
flip rules, 47
Format, 43
FourMomentum, 34, 60
FourVector, 60
frame rate, 28
frameTime, 28

G, 34, 46
gauge, 35
GaugeRules, 35
GaugeXi, 34, 35, 44, 62
Generation, 51, 52, 55
Generic, 18
generic coupling, 34, 38
generic model, 22, 43
GenericModel, 22
ghost field, 18
GhostDash, 54
Global`, 16
gluon, 65
gluon line, 54
graphics primitive, 66
GraphID, 33, 66
GraphInfo, 66
GraphName, 66
Greek, 55
GRID POSITION, 32
GS, 64

h.c., 59
HoldTimes, 66

ImageSize, 29
Incoming, 8
Index, 45, 51
index
colour, 54
Dirac, 42
generation, 19, 51, 52, 55
Greek, 55
kinematic, 43
Latin, 55

- Lorentz, 43, 55
- range of, 51
- summed over, 34

IndexDelta, 58
IndexRange, 51
Indices, 53
initialization file, 6
InsertFields, 20
insertion
- discarding an, 25
- selecting, 24
- structure of, 26

InsertionLevel, 22
InsertOnly, 53
Installation, 6
Integral, 33
integration momenta, 33
Internal, 8, 12, 44, 54
irreducible, 11, 12

J/Link, 32
Java, 32

kinematic object, 34, 38
kinematic vector, 46, 57, 59
KinematicIndices, 43
KinematicVector, 57, 59

label, 55
- LaTeX, 30, 55
- composite, 55
- indices, 55
- of mixing field, 56

LastSelections, 22, 23
leptons, 63
level, 18, 40
- classes, 18
- generic, 18
- particles, 18
- picking a, 33, 39
- specification of, 6, 20

Loop, 8
LoopNr, 35
loops
- irreducible conglomerate of, 8, 16
- number of, 9
- quark, 54

Lorentz, 44, 55
Lorentz formalism, 19
Lorentz.gen, 43, 60
Lorentzbgf.gen, 65

M$ClassesDescription, 51, 52
M$CouplingMatrices, 51, 57
M$FermionFlipRule, 43, 49
M$GenericCouplings, 43, 46
M$GenericPropagators, 43, 44
M$LastGenericRules, 43, 50, 60
M$LastModelRules, 51, 59
M$TruncationRules, 35, 43, 49
MajoranaSpinor, 46, 49, 60
Mass, 44, 53
MatrixTrace, 34
MatrixTraceFactor, 53
MB, 62
MC, 62
MD, 62
MDQ, 62
ME, 62
MetricTensor, 45, 60
MH, 62
mixing field, 18, 53
- reversed, 19

`MixingPartners`, 53
`ML`, 62
`MLE`, 62
`MM`, 62
`Model`, 22
model, 22
  classes, 22, 51
  definition, 43
  generic, 22, 43
momenta, 42
`MomentumConservation`, 35
`motifps`, 28
`MQU`, 54, 62
`MS`, 62
`MT`, 62
`MU`, 62
multiplet, 51
`MW`, 62
`MZ`, 62

neutrinos, 63
`NoElectronHCoupling`, 23, 64
`NoGeneration1`, 64
`NoGeneration2`, 64
`NoGeneration3`, 64
`NoLightFHCoupling`, 64
`NonCommutative`, 45
noncommuting object, 34, 45
`NoQuarkMixing`, 64
`NoUnfold`, 51
`Numbering`, 27
numbering
  for use with `Discard`, 25, 27

OK, 32
ordering
  of topologies, 66

`Outgoing`, 8

`Paint`, 27
`PaintLevel`, 27
`Particles`, 18
patterns, 44, 58
permutable, 16
photon line, 54
`PickLevel`, 33, 39
placeholders, 11
Poincaré group, 43
polarization vector, 38
`PolarizationVector`, 45, 49, 60
position interchange rules, 48
`PreFactor`, 35
prefactor, 38
`ProcessName`, 66
`Propagator`, 8
propagator, 8
  generic, 44
  type of, 8
  vector-boson, 45
`PropagatorArrow`, 53
`PropagatorDenominator`, 34, 44
`PropagatorLabel`, 53
`PropagatorType`, 53, 66

QCD, 64
`QED.gen`, 43
`QEDOnly`, 64
quantum numbers, 12, 52, 54
`QuantumNumbers`, 53
quarks, 63

`RelativeCF`, 38
renormalization constants, 63
replacement rules, 38–40

`Restrictions`, 22, 23, 51, 64
REVERT, 32
roadmap, 7
`RParity`, 54

`S`, 18
scalar field, 18
scalar–vector mixing, 18
`ScalarDash`, 54
screen messages, 6
`SelfConjugate`, 53, 60
`SelfEnergies`, 12
`SelfEnergyCTs`, 12
`Setup.m`, 6
`Shape`, 30
shapes, 28
  saving, 32
`SheetHeader`, 27
`Show`, 29
`Sine`, 54
`SM.mod`, 62
`SMbgf.mod`, 65
`SMc.mod`, 64
`SMQCD.mod`, 64
specifying momenta, 42
Standard Model, 62
  masses, 62
starting topology, 10, 14
  definition, 16
  name, 16
`StartingTopologies`, 10, 14
`StartTop`, 15
`Straight`, 54
subscript, 55
`SumOver`, 34
`SUNF`, 64
`SUNT`, 64
superscript, 55
`SV`, 18, 44
`SW`, 62
`SymmetryFactor`, 15, 16

`TadpoleCTs`, 12
`Tadpoles`, 12
`TeXLabels`, 66
`TheLabel`, 55
`TheMass`, 54
`Theta`, 15
`Three`, 15
`ThreeRed`, 15
`ToFA1Conventions`, 42
`Topology`, 9
topology, 8
  comparison, 16
  connected, 8
  counter-term, 10
  editor, 31
  exclusion of, 12
  filter, 12
    defining own, 14
    function, 14
  inserted, 26
  irreducible, 12
  starting, 10, 14
    definition, 16
    name, 16
  with vertex function, 11
`Topology.m`, 16
`TopologyEditor.java`, 32
`TopologyList`, 9
`ToTree`, 14
`TriangleCTs`, 12

`Triangles`, 12
`Truncated`, 35

`U`, 18
`UCAlph`, 55
`UCGreek`, 55

`V`, 18
`V4onExt`, 14
vector field, 18
`Vertex`, 8
vertex, 8
    adjacency, 8, 10
    identifiers, 16
    permutable, 16
vertex functions, 11
`VertexFunction`, 11, 34
`VertexFunctions`, 66
`ViolatesQ`, 51, 52, 54
`VS`, 19

wave-function corrections, 12
`WFCorrectionCTs`, 12
`WFCorrections`, 12