Introduction to Mathematica and FORM

Thomas Hahn

Max-Planck-Institut für Physik
München

http://wwwth.mpp.mpg.de/members/hahn/corfu2016/mmaform.pdf

Computer Algebra Systems

- Commercial systems: Mathematica, Maple, Matlab/MuPAD, MathCad, Reduce, Derive . . .
- Free systems: FORM, GiNaC, Maxima, Axiom, Cadabra, Fermat, GAP, Singular, Sage . . .
- Generic systems: Mathematica, Maple, Matlab/MuPAD, Maxima, MathCad, Reduce, Axiom, Sage, GiNaC . . .
- Specialized systems: Cadabra, Singular, Magma, CoCoA, GAP . . .
- Many more . . .

Mathematica vs. FORM

Mathematica

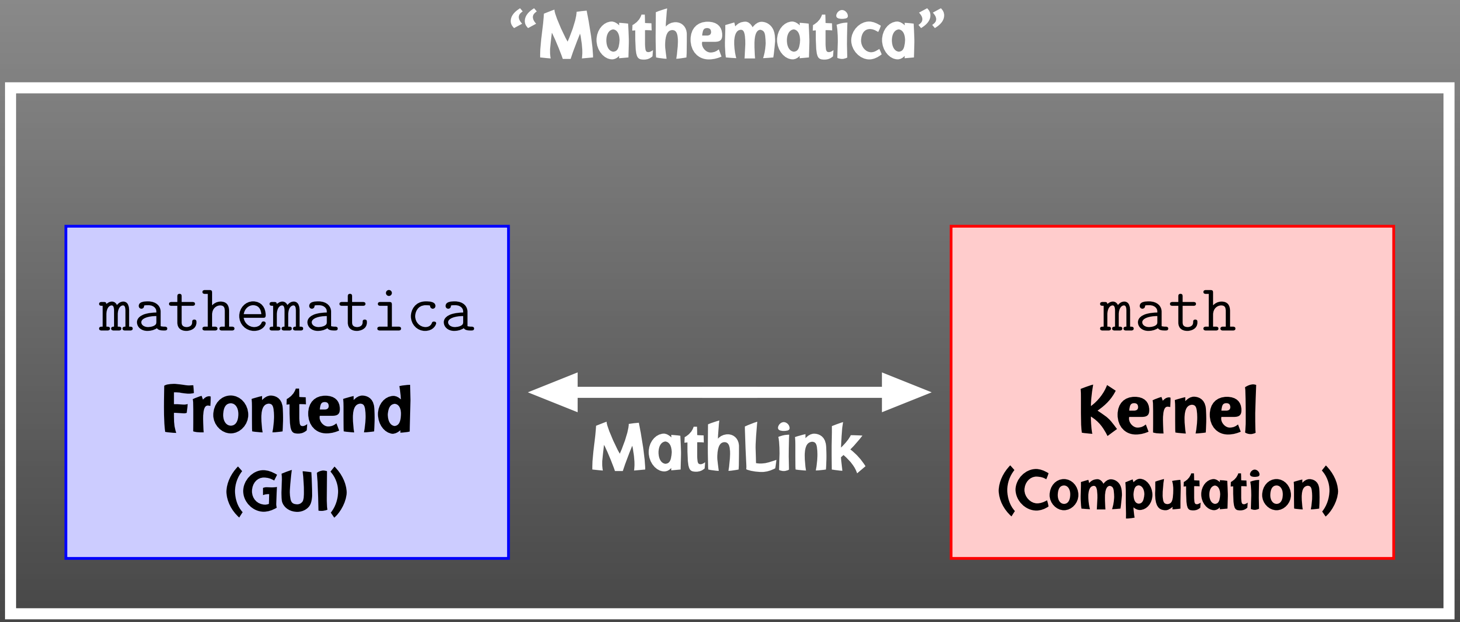
- Much built-in knowledge,
- ‘Big and slow’ (esp. on large problems),
- Very general,
- GUI, add-on packages…

FORM

- Limited mathematical knowledge,
- ‘Small and fast’ (also on large problems),
- Optimized for certain classes of problems,
- Batch program (edit-run cycle).

Mathematica

Mathematica Components


“Mathematica”

`mathematica`

Frontend

(GUI)

MathLink

`math`

Kernel

(Computation)

`http://wwwth.mpp.mpg.de/members/hahn/corfu2016/intro_math.pdf`

Expert Systems

In technical terms, Mathematica is an **Expert System.**
Knowledge is added in form of **Transformation Rules.**
An expression is transformed until no more rules apply.

Example:

```
myAbs[x_] := x /; NonNegative[x]
myAbs[x_] := -x /; Negative[x]
```

We get:

`myAbs[3]` ☞ `3`

`myAbs[-5]` ☞ `5`

`myAbs[2 + 3 I]` ☞ `myAbs[2 + 3 I]`

— no rule for complex arguments so far

`myAbs[x]` ☞ `myAbs[x]`

— no match either

Immediate and Delayed Assignment

Transformations can either be

- **added “permanently” in form of Definitions,**

```
norm[vec_] := Sqrt[vec . vec]
norm[{1, 0, 2}] ☞ Sqrt[5]
```

- **applied once using Rules:**

```
a + b + c /. a -> 2 c ☞ b + 3 c
```

Transformations can be **Immediate** or **Delayed.** Consider:

```
{r, r} /. r -> Random[] ☞ {0.823919, 0.823919}
{r, r} /. r :> Random[] ☞ {0.356028, 0.100983}
```

Mathematica is one of those programs, like TeX, where you wish you'd gotten a US keyboard for all those braces and brackets.

Almost everything is a List

All Mathematica objects are either Atomic, e.g.

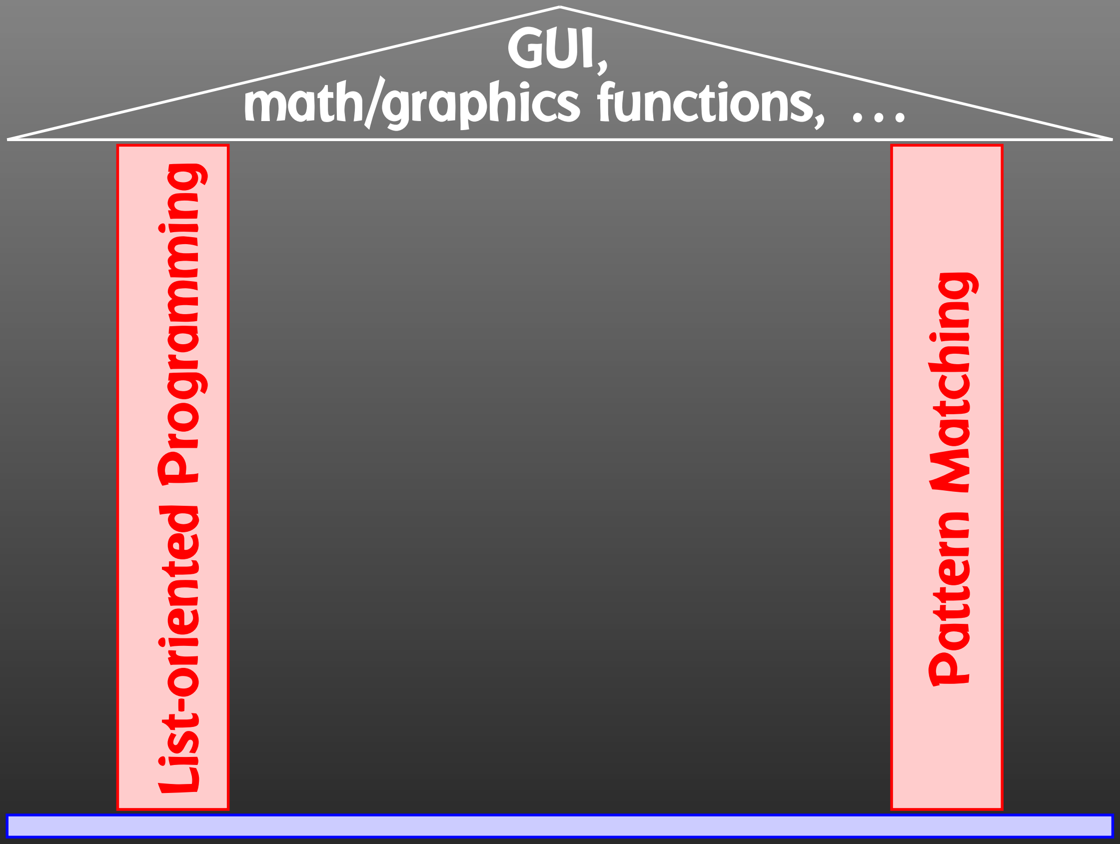
```
Head[133] ☞ Integer
Head[a] ☞ Symbol
```

or (generalized) Lists with a Head and Elements:

```
expr = a + b

FullForm[expr] ☞ Plus[a, b]
Head[expr] ☞ Plus
expr[[0]] ☞ Plus      — same as Head[expr]
expr[[1]] ☞ a
expr[[2]] ☞ b
```

The Pillars of Mathematica



GUI,
math/graphics functions, …

List-oriented Programming

Pattern Matching

List-oriented Programming

Using Mathematica's list-oriented commands is almost always of advantage in both speed and elegance.

Consider:

```
tab = Table[Random[], {10^7}];

test1 := Block[ {sum = 0},
  Do[ sum += tab[[i]], {i, Length[tab]} ];
  sum ]

test2 := Apply[Plus, tab]
```

Here are the timings:

```
Timing[test1][[1]] ☞ 8.29 Second
Timing[test2][[1]] ☞ 1.75 Second
```

Map, Apply, and Pure Functions

Map applies a function to all elements of a list:

```
Map[f, {a, b, c}] ☞ {f[a], f[b], f[c]}
f /@ {a, b, c} ☞ {f[a], f[b], f[c]}     — short form
```

Apply exchanges the head of a list:

```
Apply[Plus, {a, b, c}] ☞ a + b + c
Plus @@ {a, b, c} ☞ a + b + c     — short form
```

Pure Functions are a concept from formal logic. A pure function is defined ‘on the fly’:

```
(# + 1)& /@ {4, 8} ☞ {5, 9}
```

The `#` (same as `#1`) represents the first argument, and the `&` defines everything to its left as the pure function.

Patterns

One of the most useful features is Pattern Matching:

`_` — matches one object

`__` — matches one or more objects

`___` — matches zero or more objects

`x_` — named pattern (for use on the r.h.s.)

`x_h` — pattern with head `h`

`x_:1` — default value

`x_?NumberQ` — conditional pattern

`x_ /; x > 0` — conditional pattern

Patterns take function overloading to the limit, i.e. functions behave differently depending on *details* of their arguments:

```
Attributes[Pair] = {Orderless}
Pair[p_Plus, j_] := Pair[#, j]& /@ p
Pair[n_?NumberQ i_, j_] := n Pair[i, j]
```

Attributes

Attributes characterize a function's behavior before and while it is subjected to pattern matching. For example,

```
Attributes[f] = {Listable}
f[l_List] := g[l]
f[{1, 2}] ☞ {f[1], f[2]}     — definition is never seen
```

Important attributes: `Flat`, `Orderless`, `Listable`, `HoldAll`, `HoldFirst`, `HoldRest`.

The `Hold...` attributes are needed to pass variables by reference:

```
Attributes[listadd] = {HoldFirst}
listadd[x_, other__] := x = Flatten[{x, other}]
```

This would not work if `x` were expanded before invoking `listadd`, i.e. passed by value.

Decisions

Mathematica’s **If Statement** has three entries: for True, for False, but also for Undecidable. For example:

```
If[8 > 9, yes, no] ☞ no
If[a > b, yes, no] ☞ If[a > b, yes, no]
If[a > b, yes, no, dunno] ☞ dunno
```

Property-testing Functions end in `Q`: `EvenQ`, `PrimeQ`, `NumberQ`, `MatchQ`, `OrderedQ`, … These functions have no undecided state: in case of doubt they return `False`.

Equality

Just as with decisions, there are several types of equality, decidable and undecidable:

```
a == b ☞ a == b
a === b ☞ False
a == a ☞ True
a === a ☞ True
```

The full name of '===' is `SameQ` and works as the `Q` indicates: in case of doubt, it gives `False`. It tests for **Structural Equality.**

Of course, equations to be solved are stated with '==':

```
Solve[x^2 == 1, x] ☞ {{x -> -1}, {x -> 1}}
```

Needless to add, '=' is a definition and quite different:

`x = 3` — assign 3 to x

Selecting Elements

Select selects elements fulfilling a criterium:

`Select[{1, 2, 3, 4, 5}, # > 3 &]` ☞ `{4, 5}`

Cases selects elements matching a pattern:

`Cases[{1, a, f[x]}, _Symbol]` ☞ `{a}`

Using **Levels** is generally a very fast way to extract parts:

`list = {f[x], 4, {g[y], h}}`

`Depth[list]` ☞ `4` — `list` is 4 *levels deep* (0, 1, 2, 3)

`Level[list, {1}]` ☞ `{f[x], 4, {g[y], h}}`

`Level[list, {2}]` ☞ `{x, g[y], h}`

`Level[list, {3}]` ☞ `{y}`

`Level[list, {-1}]` ☞ `{x, 4, y, h}`

`Cases[expr, _Symbol, {-1}]//Union`

— *find all variables in* `expr`

MathLink

The **MathLink API** connects Mathematica with external C/C++ programs (and vice versa). **J/Link** does the same for Java.

```
:Begin:
:Function:      copysign
:Pattern:       CopySign[x_?NumberQ, s_?NumberQ]
:Arguments:     {N[x], N[s]}
:ArgumentTypes: {Real, Real}
:ReturnType:    Real
:End:
```

```
#include "mathlink.h"

double copysign(double x, double s) {
  return (s < 0) ? -fabs(x) : fabs(x);
}

int main(int argc, char **argv) {
  return MLMain(argc, argv);
}
```

For more details see arXiv:1107.4379.

Scripting Mathematica

Efficient batch processing with Mathematica:

Put everything into a script, using **sh's Here documents:**

```
#! /bin/sh .................. Shell Magic
math << \_EOF_ ............ start Here document (note the \)
  << FeynArts`
  << FormCalc`
  top = CreateTopologies[...];
  ...
_EOF_ ..................... end Here document
```

Everything between "<< *tag*" and "*tag*" goes to Mathematica as if it were typed from the keyboard.

Note the "\\" before *tag*, it makes the shell pass everything literally to Mathematica, without shell substitutions.

Scripting Mathematica

- Everything contained in one compact shell script, even if it involves several Mathematica sessions.
- Can combine with arbitrary shell programming, e.g. can use command-line arguments efficiently:

```
#! /bin/sh
math -run "arg1=$1" -run "arg2=$2" ... << \END
  ...
END
```

- Can easily be run in the background, or combined with utilities such as make.

Debugging hint: `-x` flag makes shell echo every statement,

```
#! /bin/sh -x
```

Code generation

- Conversion of Mathematica expression to Fortran/C painless.
- Optimized output can easily run faster than in Mathematica.
- Showstopper: Functions not available in Fortran/C, e.g. `NDSolve`, `Zeta`. Maybe 3rd-party substitute (GSL, Netlib).
- Mathematica has built-in C-code generator, e.g.

  ```
  myfunc = Compile[{{x}}, x^2 + Sin[x^2]];
  Export["myfunc.c", myfunc, "C"]
  ```

  But no standalone code: shared object for use with Mathematica (i.e. also needs license).
- FormCalc's code-generation functions produce optimized standalone code.

Mathematica ↔ Fortran

Mathematica → Fortran:

- Get **FormCalc from http://feynarts.de/formcalc**
- Write out arbitrary Mathematica expression:

```
h = OpenCode["file"]
WriteExpr[h, {var -> expr, ...}]
Close[h]
```

Fortran → Mathematica:

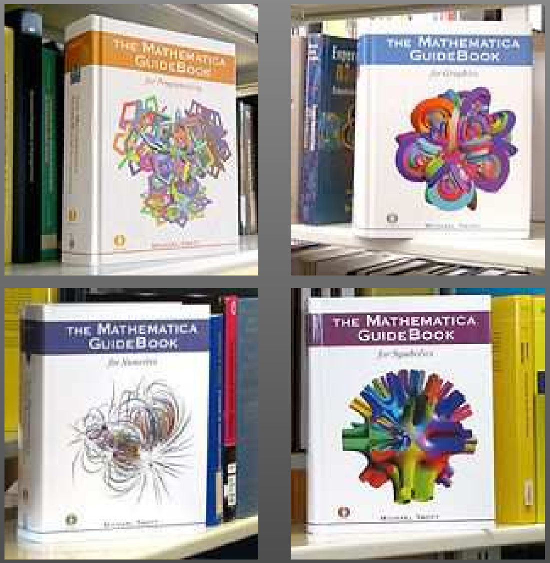
- Get **http://feynarts.de/formcalc/FortranGet.tm**
- Compile: `mcc -o FortranGet FortranGet.tm`
- Load in Mathematica: `Install["FortranGet"]`
- Read Fortran code: `FortranGet["file.F"]`

Mathematica Summary

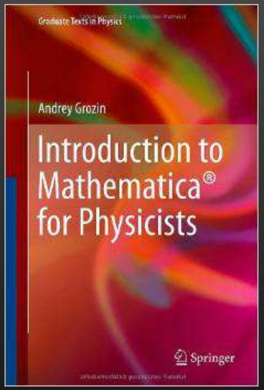
- Mathematica makes it wonderfully easy, even for fairly unskilled users, to manipulate expressions.
- Most functions you will ever need are already built in. Many third-party packages are available at MathSource, http://library.wolfram.com/infocenter/MathSource.
- When using its capabilities (in particular list-oriented programming and pattern matching) right, Mathematica can be very efficient.
  Wrong: `FullSimplify[veryLongExpression]`.
- Mathematica is a general-purpose system, i.e. convenient to use, but not ideal for everything.
  For example, in numerical functions, Mathematica usually selects the algorithm automatically, which may or may not be a good thing.

Books

- Michael Trott
  The Mathematica Guidebook
  for {Programming, Graphics,
  Numerics, Symbolics} (4 vol)
  Springer, 2004-2006.



- Andrei Grozin
  Introduction to Mathematica for
  Physicists
  Springer, 2013.



FORM

FORM Essentials

- A FORM program is divided into **Modules.** Simplification happens only at the end of a module.
- FORM is **strongly typed** – all variables have to be declared: Symbols, Vectors, Indices, (N)Tensors, (C)Functions.
- FORM works **on one term at a time:** Can do "`Expand[(a + b)^2]`" (**local** operation) but not "`Factor[a^2 + 2 a b + b^2]`" (**global** operation).
- FORM is mainly strong on **polynomial expressions.**
- FORM program + documentation + course available from **http://nikhef.nl/∼form.**

A Simple Example in FORM

```
Symbols a, b, c, d;
Local expr = (a + b)^2;
id b = c - d;
print;
.end
```

Running this program gives:

```
FORM by J.Vermaseren,version 4.0(Mar 1 2013) Run at: Tue May  8 10:14:12 2013
    Symbols a, b, c, d;
    Local expr = (a + b)^2;
    id b = c - d;
    print;
    .end

Time =       0.00 sec    Generated terms =          6
             expr        Terms in output =          6
                         Bytes used      =        104

   expr =
      d^2 - 2*c*d + c^2 - 2*a*d + 2*a*c + a^2;

  0.00 sec out of 0.00 sec
```

Module Structure

A FORM program consists of **Modules.** A Module is terminated by a “dot” statement (`.sort`, `.store`, `.end`, ...)

- **Generation Phase** (“normal” statements)
  During the execution of “normal” statements terms are only generated. This is a purely **local operation** - only one term at a time needs to be looked at.
- **Sorting Phase** (“dot” statements):
  At the end of the module all terms are inspected and similar terms collected. This is the only ‘global’ operation which requires FORM to look at all terms ‘simultaneously.’

Sorting and Generating

First module

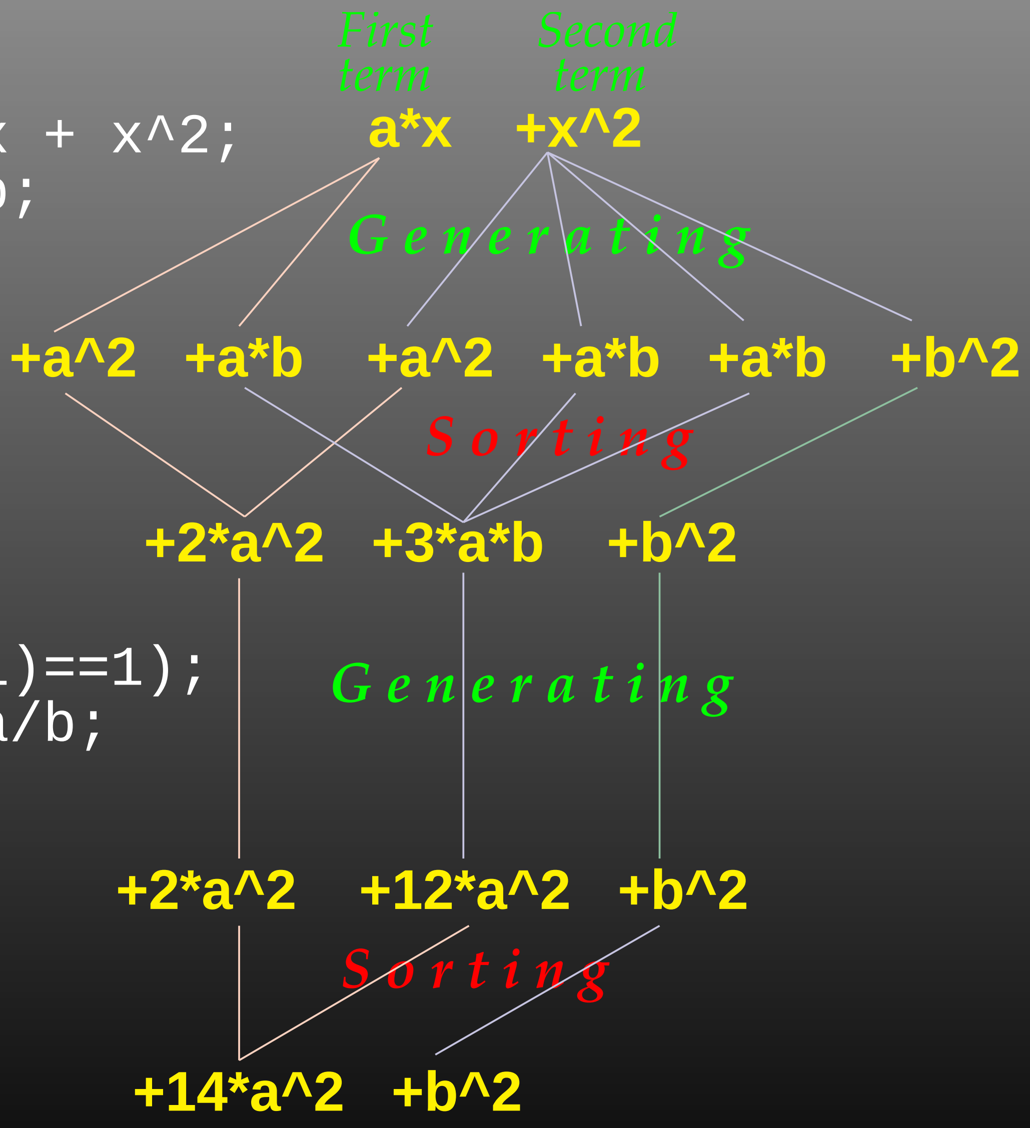
```
l expr = a*x + x^2;
id x = a + b;

.sort
```

Second module

```
if(count(b,1)==1);
multiply 4*a/b;
endif;

print;
.end
```



First term *Second term*

a*x +x^2

Generating

+a^2 +a*b +a^2 +a*b +a*b +b^2

Sorting

+2*a^2 +3*a*b +b^2

Generating

+2*a^2 +12*a^2 +b^2

Sorting

+14*a^2 +b^2

Id-Statement

The central statement in FORM is the `id`-Statement:

`a^3*b^2*c`

`id a*b = d;` ☞ `a*c*d^2` — multiple match

`once a*b = d;` ☞ `a^2*b*c*d` — single match

`only a*b = d;` ☞ `a^3*b^2*c` — no exact match possible

`id` **does not, by default, match negative powers:**

`x + 1/x`

`id x = y;` ☞ `x^-1 + y`

`id x^n? = y^n;` ☞ `y^-1 + y` — wildcard exponent

Patterns

Patterns are possible, too:

```
f(a, b, c) + f(1, 2, 3)
id f(a, b, c) = 1; ☞ 1 + f(1, 2, 3)
```

— explicit match

```
id f(a?, b?, c?) = 1; ☞ 2
```

— wildcard match

```
id f(?a) = g(?a); ☞ g(a, b, c) + g(1, 2, 3)
```

— group-wildcard match

```
id f(a?int_, ?a) = a; ☞ 1 + f(a, b, c)
```

— constrained wildcard

```
id f(a?{a,b}, ?a) = a; ☞ a + f(1, 2, 3)
```

— alternatives

Bracketing, Collecting

`bracket` **puts specified items outside the bracket.**
`antibracket` **puts specified items inside the bracket.**
`collect` **moves the bracket contents to a function.**

```
Symbols a, b, c, d;
Local expr = (a + b)*(c + d);
print;
.sort
     expr = a*c + a*d + b*c + b*d;

bracket a, b;
print;
.sort
     expr = + a * ( c + d )
            + b * ( c + d );

CFunction f;
collect f;
bracket f;
print;
.end
     expr = + f(c + d) * ( a + b );
```

Preprocessor

FORM has a Preprocessor which operates before the compiler.

Many constructs are familiar from C, but the FORM preprocessor can do more:

- `#define`, `#undefine`, `#redefine`,
- `#if{,def,ndef}` ... `#else` ... `#endif`,
- `#switch` ... `#endswitch`,
- `#procedure` ... `#endprocedure`, `#call`,
- `#do` ... `#enddo`,
- `#write`, `#message`, `#system`.

The preprocessor works across modules, e.g. a `do`-loop can contain a `.sort` statement.

Dollar Variables

- Not strongly typed, can contain ‘everything.’
- Preserved across module boundaries.
- Can be operated on during preprocessing (`#$X = ...`) and normal operation (`$X = ...`).
- Can received matched pattern: `once f(x?$var) = ...`
- No arrays.

```
s a, b;
L F = (a + b)^6;
#$n = 0;
$n = $n + 1;
print "term %$ is %t", $n;
.end
```

☞

```
term 1 is  + a^6
term 2 is  + 6*a^5*b
term 3 is  + 15*a^4*b^2
term 4 is  + 20*a^3*b^3
term 5 is  + 15*a^2*b^4
term 6 is  + 6*a*b^5
term 7 is  + b^6
```

Special Commands for High-Energy Physics

- **Gamma matrices:** `g_`, `g5_`, `g6_`, `g7_`.
- **Fermion traces:** `trace4`, `tracen`, `chisholm`.
- **Levi-Civita tensors:** `e_`, `contract`.
- **Index properties:** `{,anti,cycle}symmetrize`.
- **Dummy indices:** `sum`, `replaceloop`.
  (e.g. $\sum_i a_i b_i + \sum_j a_j b_j = 2 \sum_i a_i b_i$)

FORM Summary

- FORM is a freely available Computer Algebra System with some specialization on High Energy Physics.
- Programming in FORM takes more ‘getting used to’ than in Mathematica. Also, FORM has no GUI or other programming aids.
- FORM programs are module oriented with global (= costly) operations occurring only at the end of module. A strategical choice of these points optimizes performance.
- FORM is much faster than Mathematica on polynomial expressions and can handle in particular huge (GB) expressions.

FORM ↔ Mathematica

Mathematica → FORM:

- Get **FormCalc from http://feynarts.de/formcalc**
- After compilation the `ToForm` **utility** should be in the executables directory (e.g. Linux-x86-64):

  ```
  ToForm < file.m > file.frm
  ```

FORM → Mathematica:

- Get **http://feynarts.de/formcalc/FormGet.tm**
- Compile: `mcc -o FormGet FormGet.tm`
- Load in Mathematica: `Install["FormGet"]`
- Read a FORM output file: `FormGet["file.out"]`
  Pipe output from FORM: `FormGet["!form file.frm"]`