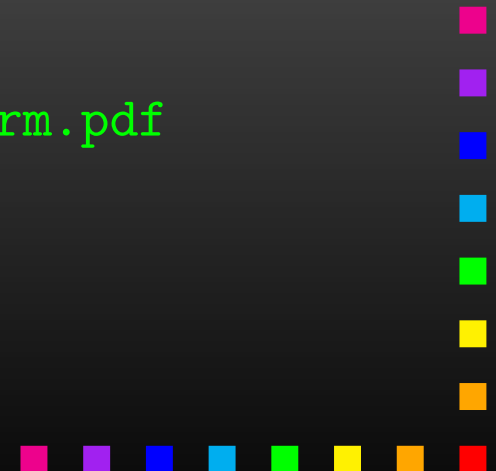


Introduction to Mathematica and FORM

Thomas Hahn

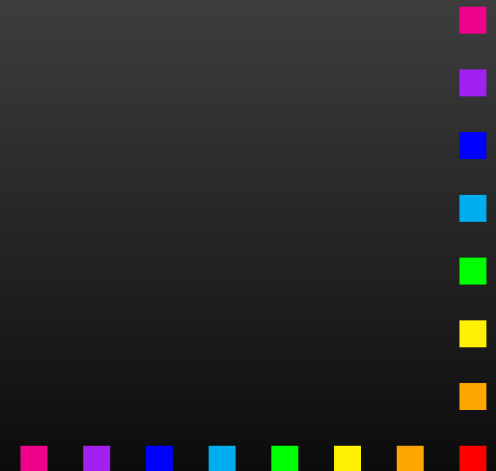
Max-Planck-Institut für Physik
München

<http://wwwth.mpp.mpg.de/members/hahn/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

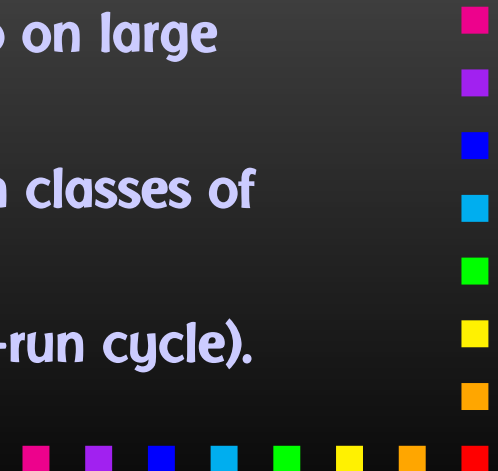


- Much built-in knowledge,
- 'Big and slow' (esp. on large problems),
- Very general,
- GUI, add-on packages...

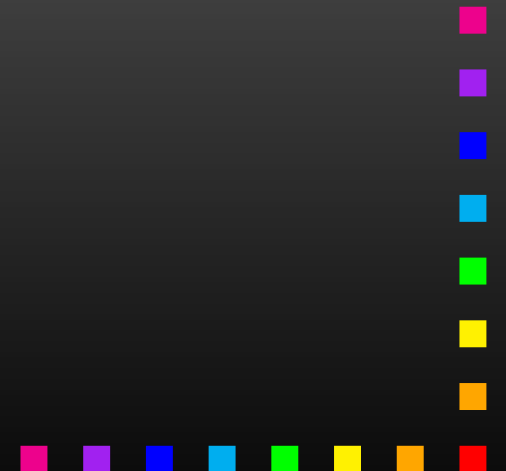
FORM



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

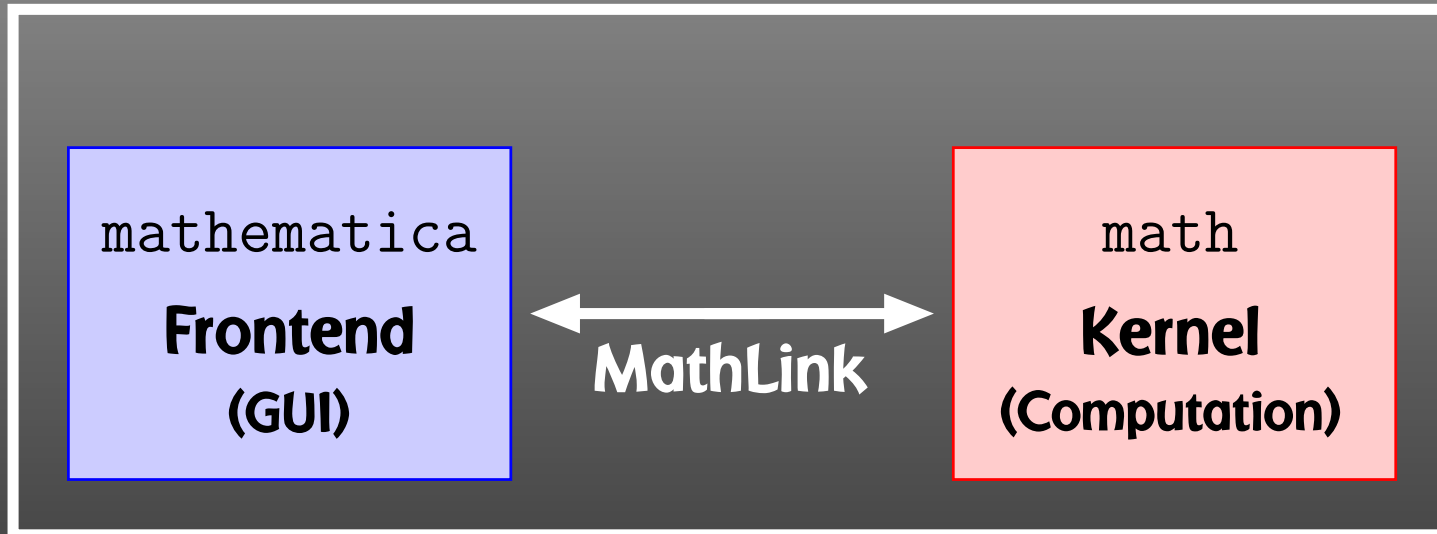


Mathematica



Mathematica Components

“Mathematica”



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



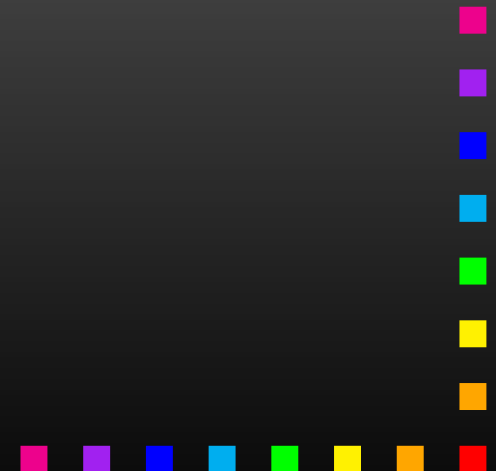
<rant>Why I hate the Frontend</rant>

FRONTEND:

- 😊 Nice formatting
- 😊 Documentation
- 😊 Ease of use
- 😞 **No obvious relation between screen and definitions**
- 😞 Always interactive
- 😞 Slow startup

KERNEL:

- 😞 Text interface
- 😞 No pretty-printing
- 😊 **1-to-1 relation to definitions**
- 😊 Interactive and non-interactive
- 😊 Scriptable
- 😊 Fast startup



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`

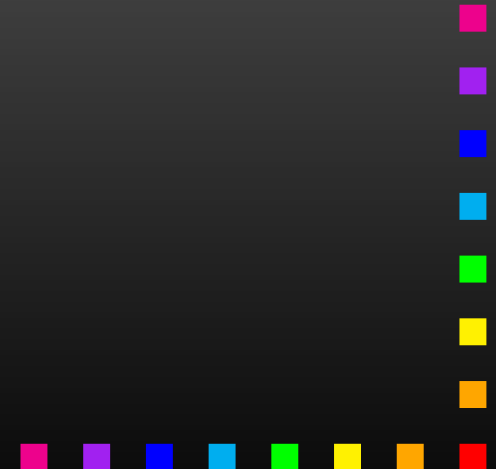
`myAbs[-5]`  `5`

`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`

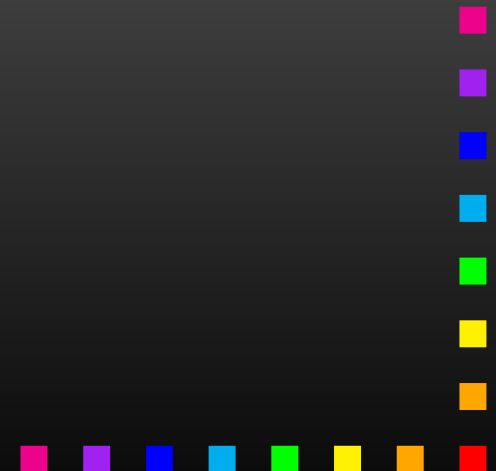
`FullForm[expr]`  `Plus[a, b]`

`Head[expr]`  `Plus`

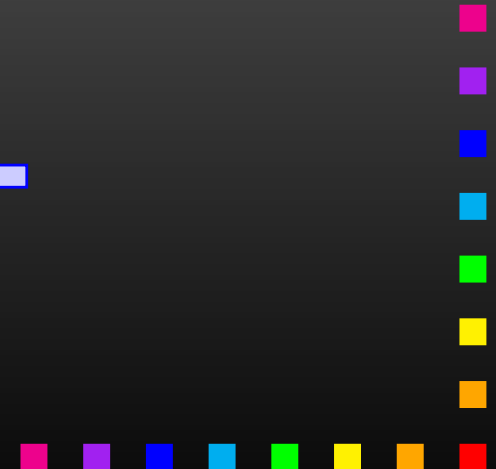
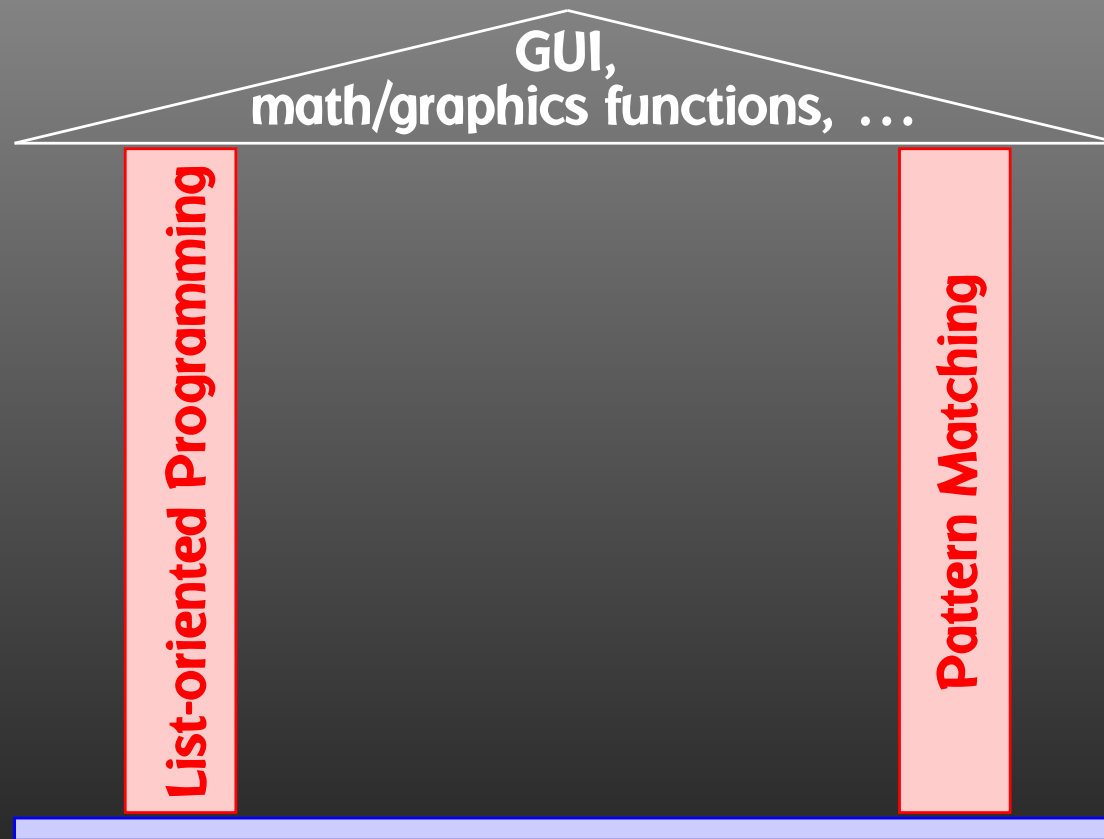
`expr[[0]]`  `Plus` — same as `Head[expr]`

`expr[[1]]`  `a`

`expr[[2]]`  `b`



The Pillars of Mathematica



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},
```

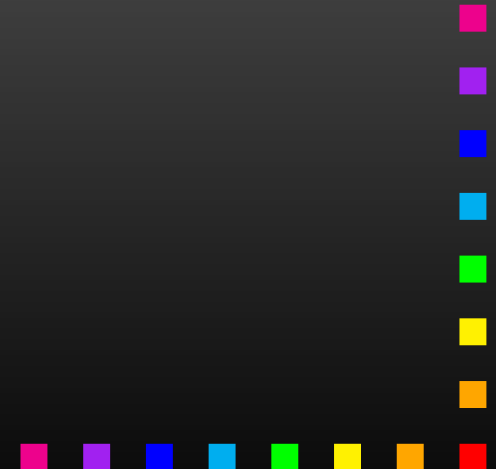
```
  Do[ sum += tab[[i]], {i, Length[tab]} ];  
  sum ]
```

```
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.



List Operations

Flatten removes all sub-lists:

`Flatten[f[x, f[y], f[f[z]]]]`  `f[x, y, z]`

Sort and **Union** sort a list. **Union** also removes duplicates:

`Sort[{3, 10, 1, 8}]`  `{1, 3, 8, 10}`

`Union[{c, c, a, b, a}]`  `{a, b, c}`

Prepend and **Append** add elements at the front or back:

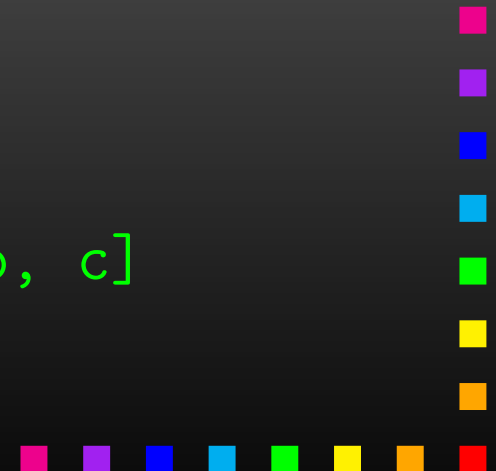
`Prepend[r[a, b], c]`  `r[c, a, b]`

`Append[r[a, b], c]`  `r[a, b, c]`

Insert and **Delete** insert and delete elements:

`Insert[h[a, b, c], x, {2}]`  `h[a, x, b, c]`

`Delete[h[a, b, c], {2}]`  `h[a, c]`



More Speed Bumps

Consider:

```
tab = Table[Random[], {10^5}];
```

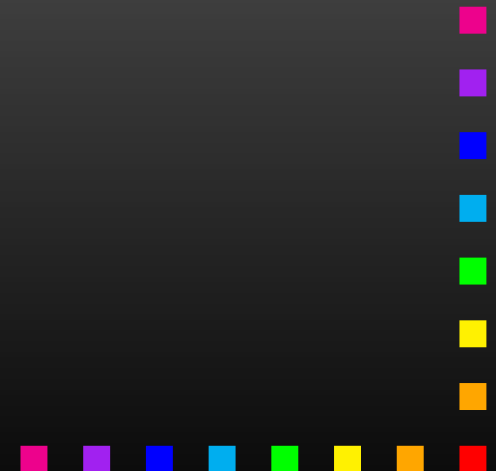
```
test1 := Block[ {res = {}},  
  Do[ AppendTo[res, tab[[i]]], {i, Length[tab]} ];  
  res ]
```

```
test2 := Block[ {res = {}},  
  Do[ res = {res, tab[[i]]}, {i, Length[tab]} ];  
  Flatten[res] ]
```

The timings:

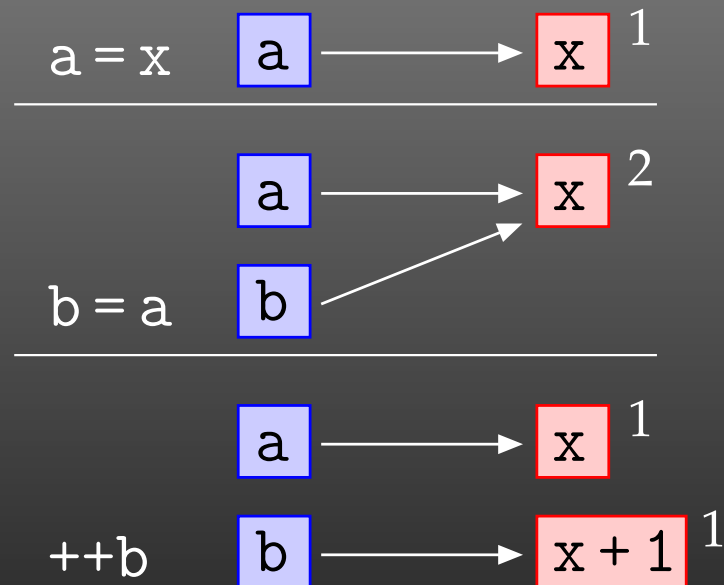
```
Timing[test1][[1]]  19.47 Second
```

```
Timing[test2][[1]]  0.11 Second
```



Reference Count

Assignments that don't change the content make no copy but just increase the **Reference Count**.



Reference Count and Speed

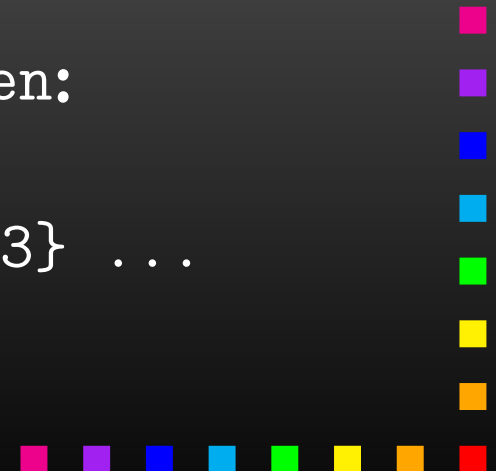
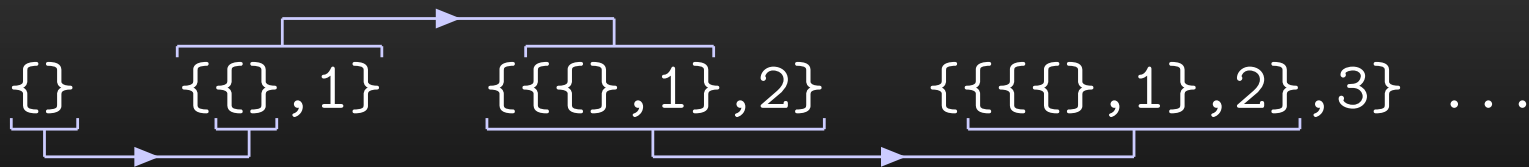
```
test1 := ...  
  ... AppendTo[res, tab[[i]]] ...  
res
```

```
test2 :=  
  ... res = {res, tab[[i]]} ...  
  Flatten[res]
```

test1 has to **re-write the list every time** an element is added:

`{}` `{1}` `{1,2}` `{1,2,3}` ...

test2 does that **only once** at the end with `Flatten`:



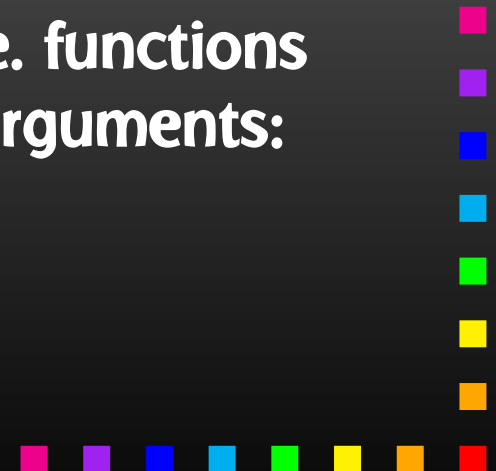
Patterns

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.

Memorizing Values

For longer computations, it may be desirable to 'remember' values once computed. For example:

```
fib[1] = fib[2] = 1
```

```
fib[i_] := fib[i] = fib[i - 2] + fib[i - 1]
```

```
fib[4]  3
```

```
?fib  Global`fib
```

```
fib[1] = 1
```

```
fib[2] = 1
```

```
fib[3] = 2
```

```
fib[4] = 3
```

```
fib[i_] := fib[i] = fib[i - 2] + fib[i - 1]
```

Note that Mathematica places more specific definitions before more generic ones.



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.

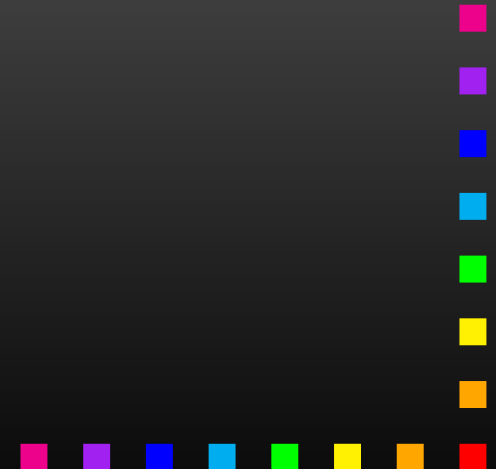
Conditional Patterns are usually faster:

```
good[a_, b_] := If[TrueQ[a > b], 1, 2]
```

– TrueQ removes ambiguity

```
better[a_, b_] := 1 /; a > b
```

```
better[a_, b_] = 2
```



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}}
```

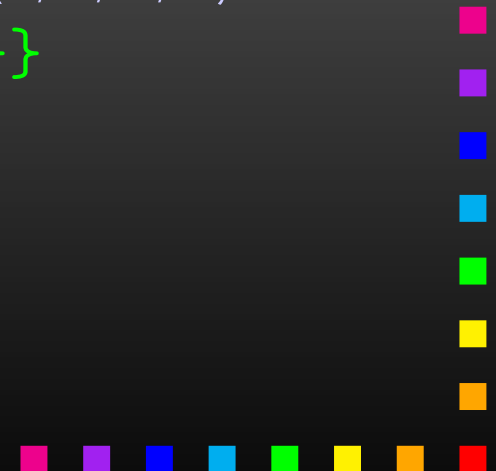
```
Level[list, {2}] → {x, g[y], h}
```

```
Level[list, {3}] → {y}
```

```
Level[list, {-1}] → {x, 4, y, h}
```

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

— find all variables in expr



Mathematical Functions

Mathematica is equipped with a large set of mathematical functions, both for symbolic and numeric operations.

Some examples:

`Integrate[x^2, {x,3,5}]`

– integral

`D[f[x], x]`

– derivative

`Sum[i, {i,50}]`

– sum

`Series[Sin[x], {x,1,5}]`

– series expansion

`Simplify[(x^2 - x y)/x]`

– simplify

`Together[1/x + 1/y]`

– put on common denominator

`Inverse[mat]`

– matrix inverse

`Eigenvalues[mat]`

– eigenvalues

`PolyLog[2, 1/3]`

– polylogarithm

`LegendreP[11, x]`

– Legendre polynomial

`Gamma[.567]`

– Gamma function



Graphics

Mathematica has formidable graphics capabilities:

```
Plot[ArcTan[x], {x, 0, 2.5}]
```

```
ParametricPlot[{Sin[x], 2 Cos[x]}, {x, 0, 2 Pi}]
```

```
Plot3D[1/(x^2 + y^2), {x, -1, 1}, {y, -1, 1}]
```

```
ContourPlot[x y, {x, 0, 10}, {y, 0, 10}]
```

Output can be saved to a file with `Export`:

```
plot = Plot[Abs[Zeta[1/2 + x I]], {x, 0, 50}]
```

```
Export["zeta.eps", plot, "EPS"]
```

[?] Hint: To get a high-quality plot with proper \LaTeX labels, don't waste your time fiddling with the `Plot` options. Use the `psfrag` \LaTeX package.




Numerics

Mathematica can express **Exact Numbers**, e.g.

```
Sqrt[2], Pi,  $\frac{27}{4}$ 
```

It can also do **Arbitrary-precision Arithmetic**, e.g.

```
N[Erf[28/33], 25]  0.7698368826185349656257148
```

But: Exact or arbitrary-precision arithmetic is fairly slow!
Mathematica uses **Machine-precision Reals** for fast arithmetic.



```
N[Erf[28/33]]  0.769836882618535
```

Arrays of machine-precision reals are internally stored as **Packed Arrays** (this is invisible to the user) and in this form attain speeds close to compiled languages on certain operations, e.g. eigenvalues of a large matrix.



Compiled Functions

Mathematica can 'compile' certain functions for efficiency. This is not compilation into assembler language, but rather a **strong typing** of an expression such that intermediate data types do not have to be determined dynamically.

```
fun[x_] := Exp[-((x - 3)^2/5)]
cfun = Compile[{x}, Exp[-((x - 3)^2/5)]]
time[f_] := Timing[Table[f[1.2], {10^5}]] [[1]]
time[fun]  2.4 Second
time[cfun]  0.43 Second
```

Compile is implicit in many numerical functions, e.g. in Plot.

In a similar manner, Dispatch hashes long lists of rules beforehand, to make the actual substitution faster.



Blocks and Modules



Block implements Dynamical Scoping

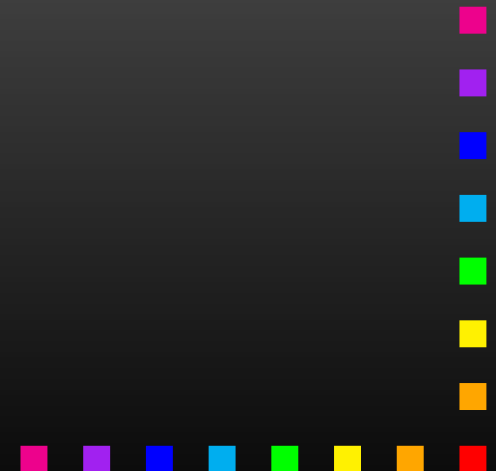
A local variable is known everywhere, but only for as long as the block executes (“temporal localization”).

Module implements Lexical Scoping

A local variable is known only in the block it is defined in (“spatial localization”). This is how scoping works in most high-level languages.

```
printa := Print[a]
a = 7
btest := Block[{a = 5}, printa]
mtest := Module[{a = 5}, printa]

btest  5
mtest  7
```



In C: Only Modules

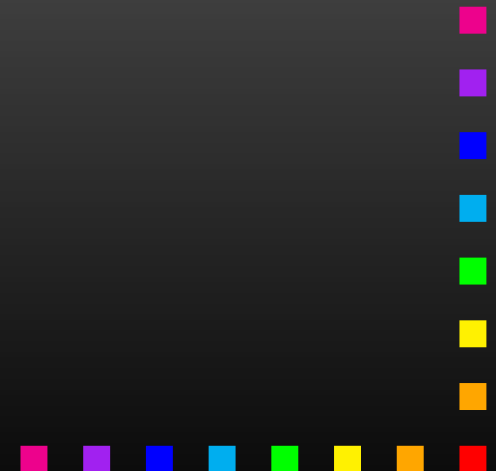
Most languages have only Lexical Scoping:

```
#include <stdio.h>
```

```
static int a = 7;
```

```
static void printa() {  
    printf("%d\n", a);  
}
```

```
int main() {  
    int a = 5;  
    printa();  
    return 0;  
}
```




DownValues and UpValues

Definitions are usually assigned to the symbol being defined: this is called **DownValue**.

For seldomly used definitions, it is better to assign the definition to the next lower level: this is an **UpValue**.

```
x/: Plus[x, y] = z
```

```
?x  Global`x  
x /: x + y = z
```

This is better than assigning to `Plus` directly, because `Plus` is a very common operation.

In other words, Mathematica “**looks**” **one level inside each object** when working off transformations.



Output Forms

Mathematica knows some functions to be **Output Forms**. These are used to format output, but don't "stick" to the result:

```
{1, 2}, {3, 4} // MatrixForm
```


$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

```
Head[%]
```



```
List
```

— not MatrixForm

Some important output forms:

InputForm, FullForm, Shallow, MatrixForm, TableForm, TeXForm, CForm, FortranForm.

```
TeXForm[alpha/(4 Pi)]
```


$$\frac{\alpha}{4\pi}$$

```
CForm[alpha/(4 Pi)]
```



```
alpha/(4.*Pi)
```

```
FullForm[alpha/(4 Pi)]
```

```
Times[Rational[1, 4], alpha, Power[Pi, -1]]
```

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](https://arxiv.org/abs/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 \)
  AppendTo[$Echo, "stdout"];
  << FeynArts'
  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
```



bash Advertisement

bash exists on 'any' system (Linux, MacOS, Cygwin + Win 10)
(but not necessarily the same as /bin/sh)

Variable assignment: `foo="1 2 3"`

Variable substitution: `ls "$foo" vs ls $foo`

Default value: `ls "${prefix:-my}"*.txt`

Alternate value: `gcc ${debug:+-Wall} -c file.c`

Value modification: `ps2pdf "$psfile" "${psfile/.ps/.pdf}"`

Brace expansion: `mv file.txt{,-old} and rm test{1..27}.o`

(Integer) Arithmetic: `$((n+5))`

Loop over all C files: `for f in *.c ; do mv {,backup/}"$f" ; done`

Arithmetic loop: `for((i=0; i<10; ++i)) ; do echo $i ; done`

Testing: `test "$time" -gt 86400 && echo too late`

`[["$email" =~ [-.%_a-z0-9]*@[-. _a-z0-9]*]] || echo Invalid`



Commercial Software?

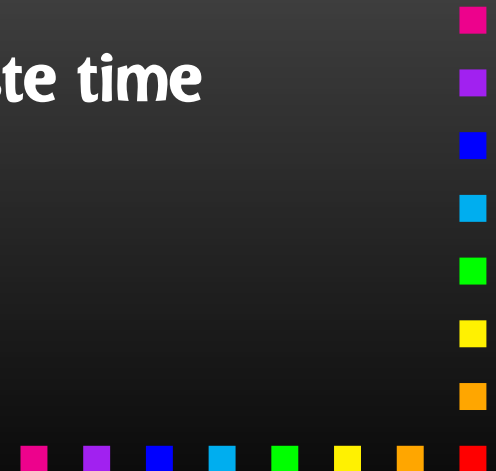
Mathematica licenses cost money ($\sim 5 \text{ k€}/\text{license}$).

While your Mathematica program runs, it blocks one license, so don't 'just' leave your Mathematica session open.

- Parallelize
- Script, Distribute, Automate
- Crunch numbers outside Mathematica

But: don't overdo it.

If your calculation takes 5 min in total, don't waste time improving.



Parallel Kernels

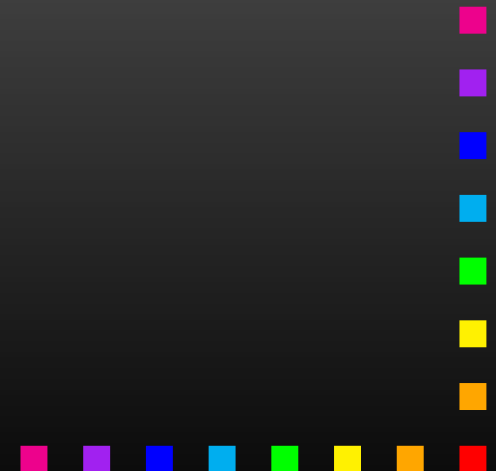
Mathematica has built-in support for parallel Kernels:

```
LaunchKernels[];  
ParallelNeeds["mypackage"];
```

```
data = << mydata;  
ParallelMap[myfunc, data];
```

Parallel Kernels count toward Sublicenses.

Sublicenses = 8 × # interactive Licenses.



Parallel Functions

- **More functions:**

```
ParallelArray   ParallelEvaluate   ParallelNeeds  
ParallelSum    ParallelCombine    ParallelTable  
ParallelDo     ParallelProduct    ParallelTry  
ParallelMap    ParallelSubmit  
DistributeDefinitions  DistributeContexts
```

- **Automatic parallelization (so-so success):**

```
Parallelize[expr]
```

- ‘Intrinsic’ functions (e.g. Simplify) **not parallelizable.**

- **Multithreaded computation partially automatic (OMP) for some numerical functions, e.g. Eigensystem.**

- Take care of **side-effects** of functions.

- Usual **concurrency stuff** (write to same file, etc).



Crunch Numbers outside Mathematica

- **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"]
```

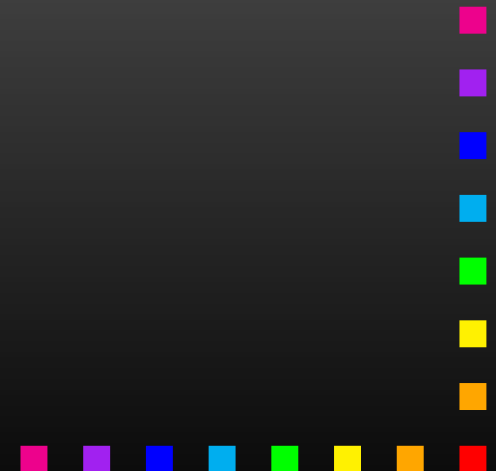
But no standalone code: shared object for use with Mathematica (i.e. also needs license).

- **FormCalc's code-generation functions produce optimized standalone code.**

Code-generation Functions

FormCalc's code-generation functions are public and disentangled from the rest of the code. They can be used to write out an arbitrary Mathematica expression as optimized Fortran or C code:

- `handle = OpenCode["file.F"]`
opens `file.F` as a Fortran file for writing,
- `WriteExpr[handle, {var -> expr, ...}]`
writes out Fortran code which calculates `expr` and stores the result in `var`,
- `Close[handle]`
closes the file again.



Code generation

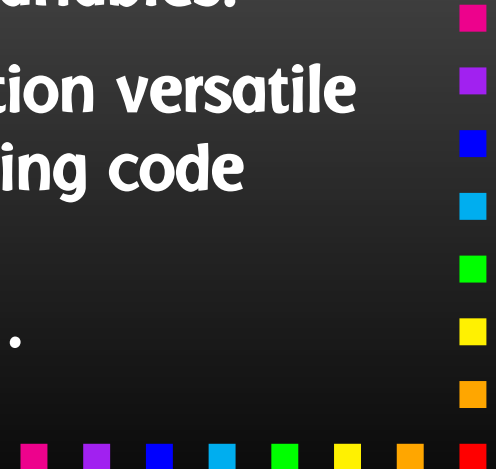
Traditionally: Output in Fortran.

Code generator is meanwhile rather sophisticated, e.g.

- **Expressions too large** for Fortran are split into parts, as in

```
var = part1  
var = var + part2  
...
```

- **High level of optimization**, e.g. common subexpressions are pulled out and computed in temporary variables.
- **Many ancillary functions** make code generation versatile and highly automatable, such that the resulting code needs few or no changes by hand:
VarDecl, ToDoLoops, IndexIf, FileSplit, ...



C Output

- **Output in C99** makes integration into C/C++ codes easier:

```
SetLanguage["C"]
```

Code structured by e.g.

- **Loops and tests handled through macros, e.g.**
LOOP(var, 1, 10, 1) ... ENDLLOOP(var)
- **Introduced data types RealType and ComplexType for better abstraction, can e.g. be changed to different precision.**



Mathematica \leftrightarrow Fortran

Mathematica \rightarrow Fortran:

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

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

Fortran \rightarrow 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]`.

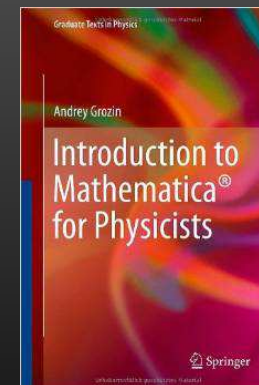
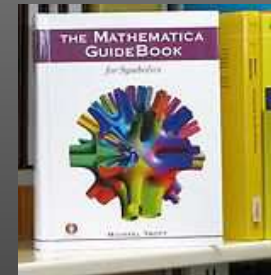
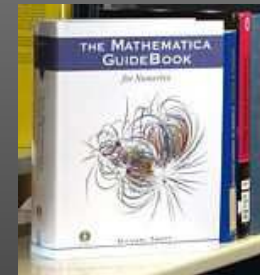
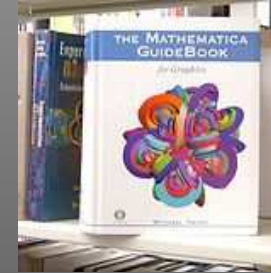
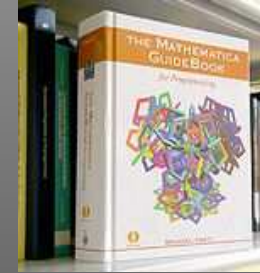
- Mathematica is a general-purpose system, i.e. convenient to use, but not ideal for everything.

For example, in numerical functions, Mathematica usually selects the algorithm automatically, which may or may not be a good thing.

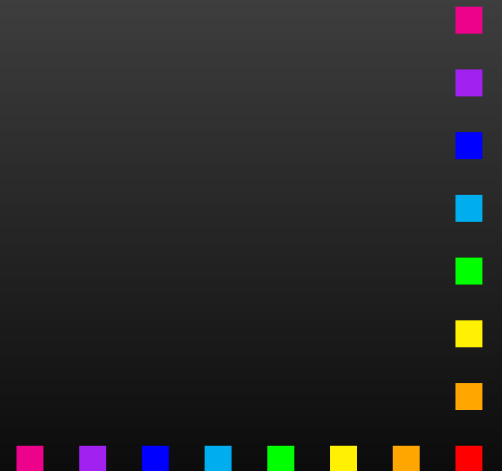


Books

- **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)

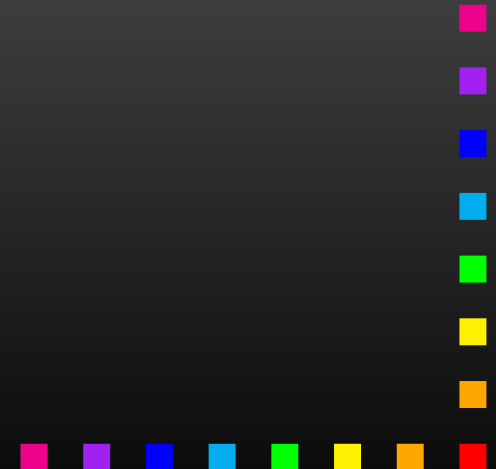
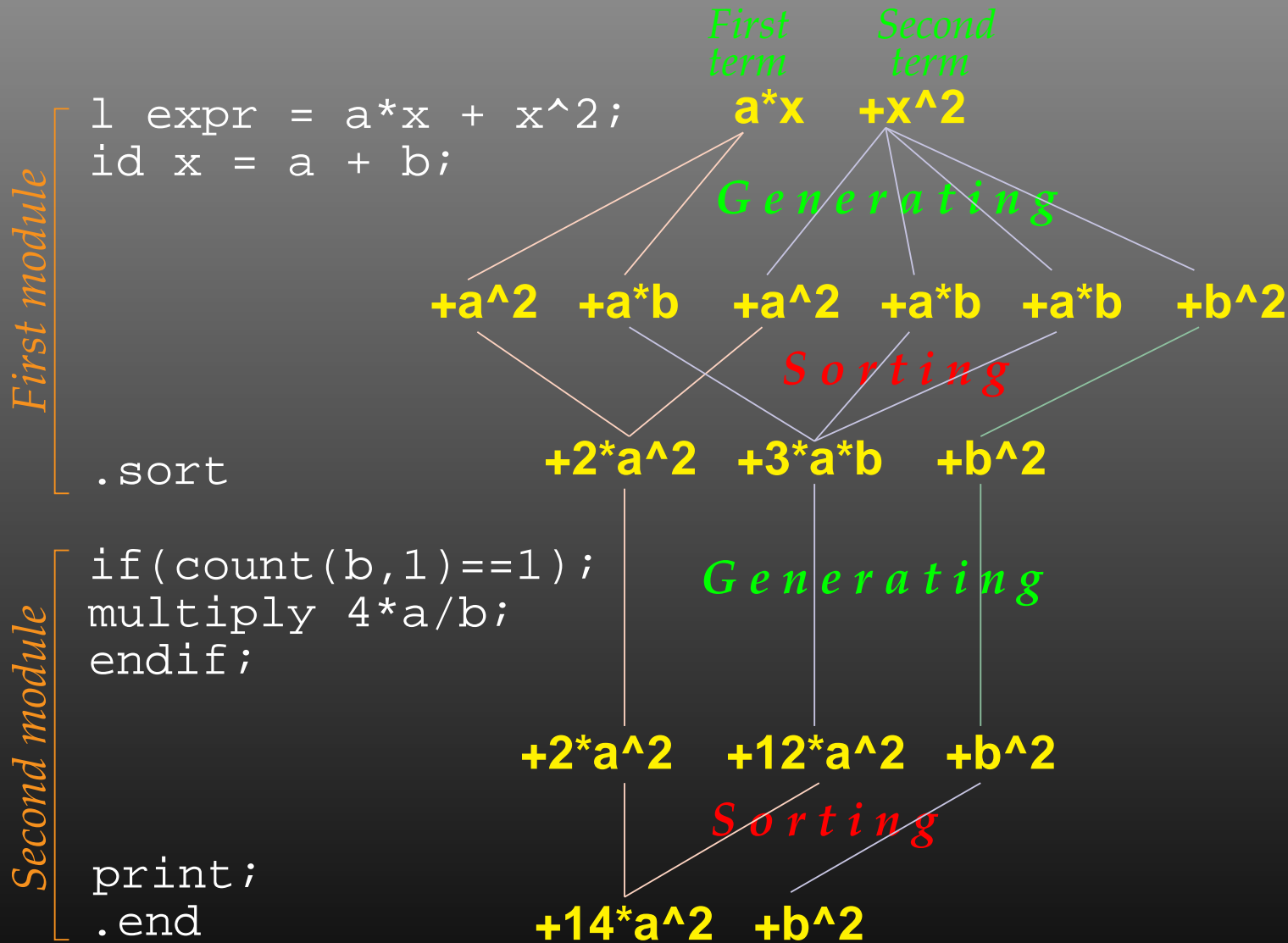
During the execution of “normal” statements terms are only generated. This is a purely **local operation** – only one term at a time needs to be looked at.

- **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.’

Closest in Mathematica: use **inert functions** instead of Mathematica’s ones, e.g. `expr /. Plus -> plus`.

Sorting and Generating



Id-Statement

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

$$a^3*b^2*c$$

$$\text{id } a*b = d; \quad \text{↔} \quad a*c*d^2$$

– multiple match

$$\text{once } a*b = d; \quad \text{↔} \quad a^2*b*c*d$$

– single match

$$\text{only } a*b = d; \quad \text{↔} \quad a^3*b^2*c$$

– no exact match possible

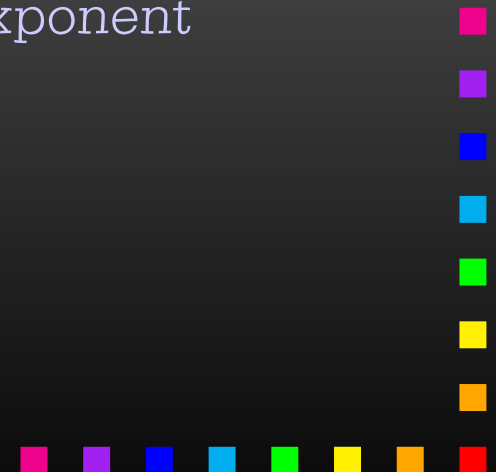
`id` does not, by default, match negative powers:

$$x + 1/x$$

$$\text{id } x = y; \quad \text{↔} \quad x^{-1} + y$$

$$\text{id } x^n? = y^n; \quad \text{↔} \quad y^{-1} + y$$

– wildcard exponent



Patterns

Patterns are possible, too:

$f(a, b, c) + f(1, 2, 3)$

$\text{id } f(a, b, c) = 1; \text{ } \rightarrow 1 + f(1, 2, 3)$

– explicit match

$\text{id } f(a?, b?, c?) = 1; \text{ } \rightarrow 2$

– wildcard match

$\text{id } f(?a) = g(?a); \text{ } \rightarrow g(a, b, c) + g(1, 2, 3)$

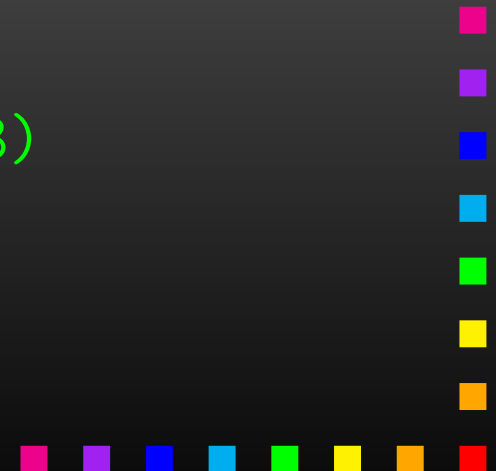
– group-wildcard match

$\text{id } f(a?\text{int}_?, ?a) = a; \text{ } \rightarrow 1 + f(a, b, c)$

– constrained wildcard

$\text{id } f(a?\{a,b\}, ?a) = a; \text{ } \rightarrow a + f(1, 2, 3)$

– alternatives



Bracketing, Collecting

`bracket` puts specified items outside the bracket.

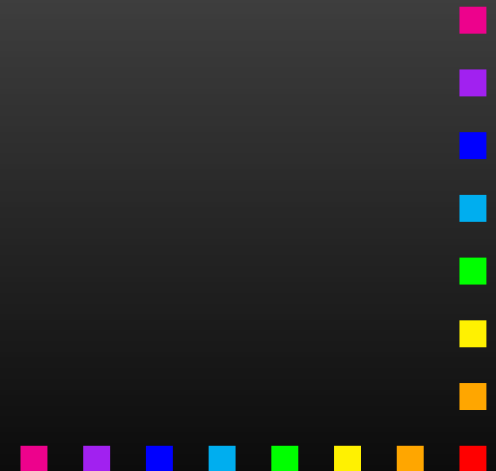
`antibracket` puts specified items inside the bracket.

`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, #undef, #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 receive matched pattern: once `f(x?$var) = ...`
- No arrays.

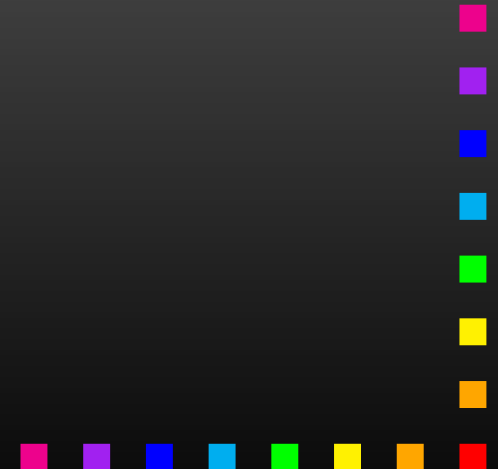
```
s a, b;  
L F = (a + b)^6;  
#$n = 0;  
$n = $n + 1;  
print "term %$ is %t", $n;  
.end
```

☞ term 1 is + a⁶
term 2 is + 6*a⁵*b
term 3 is + 15*a⁴*b²
term 4 is + 20*a³*b³
term 5 is + 15*a²*b⁴
term 6 is + 6*a*b⁵
term 7 is + b⁶



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. $\sum_i a_i b_i + \sum_j a_j b_j = 2 \sum_i a_i b_i$)
- **Very efficient combinatorics:** `dd_`, `distrib_`.



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 \leftrightarrow Mathematica

Mathematica \rightarrow 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 \rightarrow Mathematica:

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



Exercise

Write a Mathematica function that works somewhat similar to `Select`. It should, however, not just give back the list of items for which the test function is true, but

{ list of items for which test is `True`,
list of items for which test is `False` }

Try to find a version in which the test function is not evaluated more than once per item!

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

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

<http://wwwth.mpp.mpg.de/members/hahn/mmaform.tar.gz>

