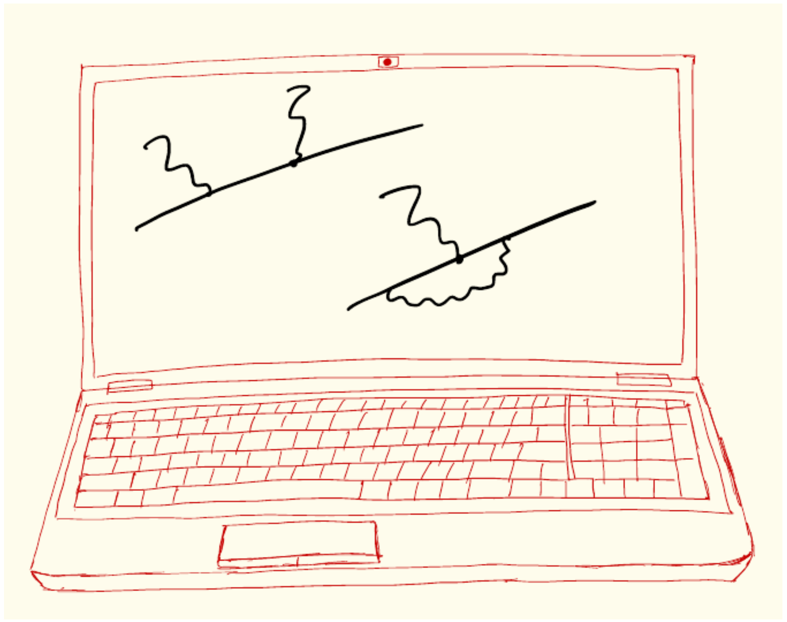
Computer Algebra for Feynman Graphs



2.

In this Lecture

- Structure of a FORM program: blocks (modules), their structure, types of module instructions
- Substitutions, pattern matching

Structure of a FORM Program

- Before we continue with pattern matching, let us briefly consider the structure of a FORM program and how FORM executes it
- A program consists of separate blocks – **modules**
- These blocks are run one by one in the following sequence:
  - the code is run through the preprocessor; it prepares the code, transforming preprocessor directives (such as preprocessor variables, loop and conditional structures, and so on – we will discuss some of them later) into compiler-recognizable statements;
  - the compiler translates the resulting code into executable instructions (machine code)
  - when the compiler encounters an end-of-module statement, it stops, and the calculations are executed
  - the resulting expressions are sorted and (if needed) are printed, stored, and used as input in the subsequent module

NB: FORM Does Local Transformations

- FORM represents the expressions that it works with as lists of terms
- All instructions are executed term-by-term: the transformations are applied to one single term in an expression at a time, so FORM works locally
- This explains why substitutions where the left-hand side is not a monomial (i.e., not a single term) are not allowed:

  ```
  id a + b = c
  ```

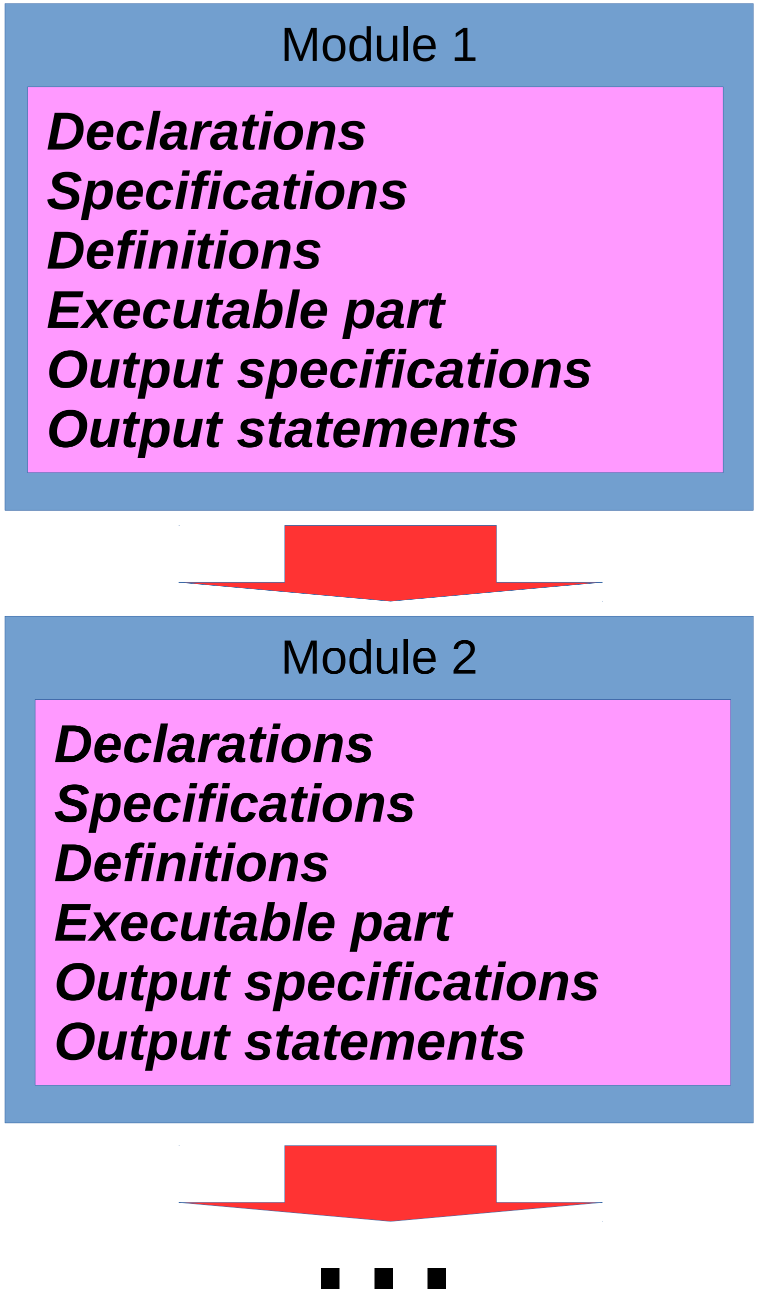
  this is just non-local!
- This also means that FORM cannot do factorization and other non-local transformations
- The representation of expressions as lists differs from what is done in, e.g., Mathematica where expressions are essentially trees (with many levels [branching points]); this is what makes FORM so fast!

Modules: Structure

- Modules generally consists of several sections, going in the following order:
  - ***Declarations*** (variables: vectors, indices, symbols, functions etc.; instructions on how to handle statistics [names, generated symbols etc.])
  - ***Specifications*** (instructions on how to handle existing expressions)
  - ***Definitions*** (introduce new expressions)
  - ***Executable part*** (operations on expressions)
  - ***Output specifications*** (formatting output)
  - ***Output statements*** (e.g., print/save the results)
- The order is strict, i.e., no statements from one of these categories can come after statements from subsequent categories

  [Example file: Module_Structure_1.frm]
- Each module ends with one of the module instructions: .sort, .end, .clear, or .store.

FORM Program



Module Instructions: Types

[Example file: Module_Instructions_1.frm]

- `.sort`
  - Processes all active expressions and prepares them (sorts) as input for the next module
- `.end`
  - Processes all active expressions and terminates FORM
- `.clear`
  - Processes all active expressions, and clears buffers except the main input (as if FORM started from this point in the program)
- `.store`
  - Processes all active expressions, stores global expressions in a storage file, removes all local expressions and all declarations (except for those that were specified before the `.global` instruction)
- `.global`
  - Saves declarations made before its occurrence from being deleted by the `.store` instruction

Notes: Usage of Module Instructions:

- `.sort` [Example file: Module_Instructions_1.frm]
  - Inserting it in the code can be very useful: since it results in a sorted (=shorter) expression, it can greatly reduce the number of generated terms; on the other hand, sorting takes time, so excessive use of it may slow the code down instead of speeding it up. Use try and error here!
- `.end`
  - Quite obvious: it terminates the program. Every FORM program should end with this statement (otherwise it is generated automatically as FORM encounters the end of the input file, along with an error message)
- `.store`
  - Global expressions that have been stored are no longer active, but can be used in assignments in the r.h.s., e.g., `Local expr = storedexpr;`
- `.global`
  - Can be useful when some of the definitions of symbols etc. need to be carried over to the next module after `.store` is executed

Substitutions

- Recall the identification statement
  `identify something = [something else];`
- It replaces the term that is in the lhs by the rhs
- It does so only once to a single term in an expression (so that it does not operate on the result of its application), term by term
- Consequently, it can be applied to monomial terms (≈products of objects) only:
  `id x * a * b * z = w; id b=b^2;` both work fine
  `id x + y = z;` results in an error
- Numerical factors are not allowed either:
  `id 2 * x = y;` won't work! [Check it!]
- It can be applied repeatedly:

```
repeat;
id something = anotherthing;
endrepeat;
```

any set of executable statements can be here!

- This cycle continues as long as terms keep being changed inside the cycle

Some Properties of Substitutions

- Simple substitutions: [Example file: Substitutions_1.frm]
  - Integer powers of symbols or their products
    `id x^2 = y + 1;`
    - zero exponent is tricky [not allowed according to the manual], so it is better to avoid it: `id x^0 = a;`
    - positive and negative powers are matched separately
      `id x = y` will change, e.g., `x^2` to `y^2`, but will not change `1/x` etc.
      `id 1/x = 1/y` will change all negative powers of `x` to respective powers of `y`
  - As many substitutions as possible are made (in these simple cases), e.g., `id x^2 = y` changes `x^5` to `y^2 x`
- Substitutions are not done inside of function arguments:
  `Local expr = f(x);`
  `id x = y^2` will not change `expr`
- There is an environment `argument` … `endargument` that allows manipulating arguments of functions; it can be nested, too

Example: Contractions of γ Matrices

- Let us calculate simple identities such as

$$\gamma_\mu \gamma^\nu \gamma^\mu = -2\gamma^\nu, \qquad \gamma_\mu \gamma^\nu \gamma^\lambda \gamma^\mu = 4g^{\lambda\nu}$$

[Example file: Gamma_Matrices_1.frm]

```
#-

ntensor g;
 tensor d(s);
index mu, nu, la, rh;

Local g2 = g(mu) * g(nu) * g(mu);
Local g4 = g(mu) * g(nu) * g(la) * g(mu);

repeat;
id g(mu) * g(nu) = 2 * d(mu,nu) - g(nu) * g(mu);
id g(mu) * g(mu) = 4;
id g(mu) * d(mu,nu) = g(nu);
id g(la) * g(mu) = 2 * d(la,mu) - g(mu) * g(la);
id g(mu) * d(mu,la) = g(la);
id g(la) * g(nu) = 2 * d(la,nu) - g(nu) * g(la);
endrepeat;

print;

.end
```

Excercise: calculate identities
with 3 and 4 sandwiched γ matrices

Example: Derivatives

- Let us calculate derivatives of expressions of the form $x^m e^x \cos^k x \sin^l x$

[Example file: Derivatives_1.frm]

```
#-

functions sin, cos, exp, D, x;
cfunctions Sin, Cos, Exp, X;

Local expr = D^4 * exp(x) * x^5 * cos(x)^2 * sin(x)^3;

repeat;
id D * exp(x) = exp(x) + exp(x) * D;
id D * x = 1 + x * D;
id D * sin(x) = cos(x) + sin(x) * D;
id D * cos(x) = - sin(x) + cos(x) * D;
endrepeat;

id x = X;
id sin(x) = Sin(X);
id cos(x) = Cos(X);
id exp(x) = Exp(X);
id D = 0;

print;

.end
```

Excercise: extend the code to expressions that have the form

$$x^m e^{ax} \cos^k(bx) \sin^l(cx)$$

and calculate derivatives of a few expressions; check with Mathematica etc. or by hand

Pattern Matching in Form: Wildcards

- Simple substitutions work well, however, it is very often the case that the programs could be written more efficiently if one could replace classes of expressions (generic objects)
- For instance, in the examples above:
  - Commutators of two γ matrices could be substituted for generic indices:
    $$\gamma^\mu\gamma^\nu = 2g^{\mu\nu} - \gamma^\nu\gamma^\mu$$
  - One could take the derivatives using the chain rule:
    $$f[g(x)]' = f'[g(x)]\,g'(x)$$
- FORM has a very powerful tool that allows one to do very efficient pattern matching: wildcards (wildcard objects)
- A wildcard is obtained when a ? symbol is added to a name of a declared object:

```
symbol a, x;
id a? = x;
```

- This wildcard matches any symbol; in general, the type of matched objects is defined by the type of the object used in the wildcard

Wildcards: Examples

- Some wildcards:

```
symbol a, b, c, x, y, z;
index mu, nu;
vector p, q;
function f, g;
```

  `a?` - matches any symbol

  `a?^2` – matches any symbol squared

  `p(mu?)` - matches any component of vector `p`

  `p?` - matches any vector

  `f(x?)` - matches function f of any scalar argument

  [`f(x), f(q.p), f(2 * x + y), f(1),` ...]

  `f(p?)` - matches function f of any vector argument

  [`f(p), f(q + p), f(2 * p + 3 * q)` …]

  `f?(x?,y)` – matches any function of two scalar arguments where `y` is the second argument

  `mu?` - matches any index

  ...

Wildcards: Usage and Properties

- Wildcards are used in the l.h.s. of the substitutions:
  `id f(g?(x),x) = x^2;`
- Symbols that define the wildcard can appear in the r.h.s. of the substitutions:
  `id f(g?(x?),x) = g(x)`
  The r.h.s. symbols will take the values that match the pattern on the l.h.s.:
  `id f(g?(x?),x) = g(x):     f(h(y),x) → h(y)`
- If a symbol appears in a wildcard in the l.h.s. more than once, all instances should be the same in order to match: `f(x?,x?)` matches all appearances of function `f` where it has two equal (scalar) arguments, but not those where `f` has two different arguments
- Sometimes wildcards can match more general patterns compared to separate wildcard objects; this is especially true when wildcards are used in function arguments, e.g., `x^n?` will match powers of `x` with both symbolic and integer numerical `n`, but `n?` will match only symbolic `n`; `x?` will match any symbol, whereas `f(x?)` will match function f with any scalar-like argument [e.g., `f(p.q)` or `f(1+2*x)`].

Wildcards in Functions: Properties

- There are some properties regarding wildcards in function arguments:
  - Function wildcards cannot match tensors and vice versa
  - Index wildcards will match indices and vectors contracted with functions: `f(mu?)` matches `f(nu)` and `f(p)=f(nu) * p(nu)` [SCHOONSCHIP]
  - Vector wildcards match vectors and vector-like expressions: `f(p?)` matches `f(q), f(p+2*q), f(x*p + y*q),` … but not `f(x + q)` etc.
  - Symbol wildcards can match any scalar arguments: `f(x?)` matches `f(y), f(x + y*z), f(2), f(p.q + 3*x),` ...

Argument Field Wildcards

- A different type of wildcard exists for functions: argument field wildcards, which refer to groups of function arguments:
  `f(?group,x)` matches all instances of `f` where the last argument is `x` and other arguments can be arbitrary
- Here, `group` is just an identifier, no need to define a variable `group`
- Argument field wildcards cannot be used in symmetric and antisymmetic functions (due to the need to make too many argument permutations in the function arguments in order to identify the possible pattern match, which slows the calculation down)
- Note: too many argument field wildcards can make the pattern matching very slow (since there are too many options)
- Also, note that there can be clashes, for instance, if `f` is a function and `g` is a tensor, the substitution `id f(?a)=g(?a)` can lead to arguments [such as scalars] that are legal for a function but illegal for a tensor passed to `g`; this will lead to a runtime error [a simple wildcard variable would generate a compilation error in a such case]

Excercise

- Rewrite the programs that calculate the contraction of γ matrices and the derivatives of a trigonometric monomial, using wildcards