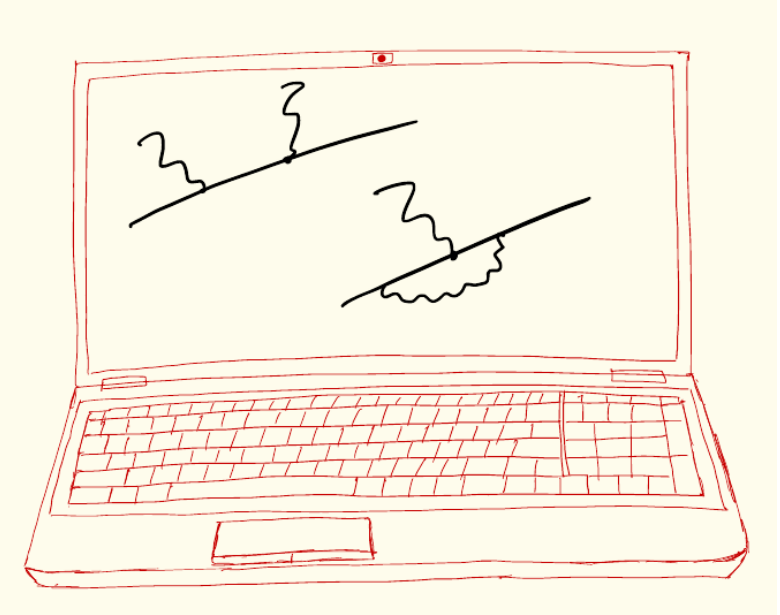
Computer Algebra for Feynman Graphs



6.

In this Lecture

- $-variables
- Flow control at the level of the compiler: if, repeat, while, do

Variables in FORM: $-variables

- As we saw previously, there are two main types of variables in FORM:
  - Algebraic variables: expressions and objects that enter there (symbols, indices, vectors, etc.), are manipulated on by the compiler during execution
  - Preprocessor variables: string variables, used by the preprocessor to prepare input for the compiler
- Dollar variables ($-variables) are something in-between the first two types, they can be assigned and used both by the compiler and the preprocessor
- A valid name for a $-variable starts with a `$` sign, followed by a letter, the rest can be alphanumeric characters:
  `$a, $b14, $aaa1, …`
- $-variables do not have to be declared, however, they should be assigned, i.e., given a value, before they are used (FORM will complain otherwise, although this will not be an error)
- They are stored in memory so should be kept small

$-variables: What Can It Be?

- The following can be stored in a $-variable:
  - An algebraic expression: `$expr = (x+y)^4;`
  - An individual object such as symbol, index, vector, function, etc.
  - Parts of a term (treated as a special case of an algebraic expression; example below)
  - Argument field of a function (example below)
- FORM will try to convert between these different types (except from argument-field-like variables), depending on how the $-variable is used
- If a $-variable is used in a way that should not be possible, e.g., a variable whose value is a vector used where a symbol is expected, this will produce an error
- $-variables cannot have arguments (as opposed to preprocessor variables)

$-variables: Assignment

- Assigning a $-variable can proceed in a few ways, some of which can be quite useful:

  [Example file: dollar_vars.frm]

  - At the preprocessor level analogously to preprocessor variables:
    `#$x = something;`
    The r.h.s. can be any algebraic expression, it also can contain $-variables and local/global expressions
  - During execution in a similar statement:
    `$x = something;`
    Here, the r.h.s. can again be any valid algebraic expression; it is evaluated by the compiler and the result is assigned to `$x`. Note that existing values are overwritten – in contrast to what happens with preprocessor variables
  - During execution a $-variable can be assigned the value of a wildcard:
    `id f(x?$y) = something;`
    In this case, the $-variable `$y` takes the value of the function argument that matches the wildcard `x?`. If there are several matches, `$y` keeps the value of the last match after the `id` statement is executed. Note the order can be implementation dependent!

$-variables: Wildcards

- The last way to assign $-variables, i.e., using them in wildcarcs, can be quite useful, for instance, one can see what exactly generated a match to the wildcard – very handy in debugging [Example file: dollar_vars.frm]
- In general, $-variables can be attached to any wildcards:
  - `id f?$x1(x?$x2) * y$x3 = something;`
    In this example `$x1`, `$x3`, and `$x3` will take the values of those `f`, `x`, and `y` that matched the wildcards
  - `id f(?a$x4) = something;`
    Here, `$x4` will become the argument field of the function that generated the match. It can be used from that point on in assignments like `Local expr = g($x4)`
  - `id f(x?set{a,b,c}$x5) = something;`
    This is an example of how a $-variable is attached to a wildcard with a restriction

$-variables in Wildcards: Examples

[Example file: dollar_vars.frm]

```
symbol a, b, c, x, y, z;
functions f, g, h;
Local expr = f(x) * y + g(x) * a;

id f?$x1(x?$x2) * y?$x3 = g(f(x)) * h(y);

print "Matched wildcard: {f,x,y} = {%$, %$, %$}", $x1, $x2, $x3;
print;
```

The output will be

```
Matched wildcard: {f,x,y} = {f, x, y}
Matched wildcard: {f,x,y} = {g, x, a}

expr =
      g(f(x))*h(y) + g(g(x))*h(a);
```

This is a special version of the `print` statement (to be discussed in detail later, see Lecture 8) that prints values term-by-term during execution

We can see what triggered the matches of each wildcard, term-by-term

The values of `$x1`, `$x2`, and `$x3` at this point are `g`, `x`, and `a`, as is easy to check by printing them out (note that one needs a `.sort` to close this module) before printing the $-variables again

$-variables in Wildcards: Examples

[Example file: dollar_vars.frm]

```
Local expr=f(x,y,z)+g(x + 1, 2 * y + 3 * z)+h(a * x^2, b, c, x);
id f?(?a$x4) = f(?a) * g(?a);
print "Matched wildcard: ?a = %$", $x4;
```

This will print

```
Matched wildcard: ?a = x,y,z
Matched wildcard: ?a = 1 + x,3*z + 2*y
Matched wildcard: ?a = a*x^2,b,c,x
```

This is also an example of a $-variable taking the value of the argument field of a function

```
Local expr = (a * x + b)^5;
id x?{x}$x1^a?$x2 = f($x1)^$x2;
print "Matched wildcard: %$^%$", $x1, $x2;
```

The printout is

```
Matched wildcard: x^5
Matched wildcard: x^4
Matched wildcard: x^3
Matched wildcard: x^2
Matched wildcard: x^1
Matched wildcard: x^1
```

And this is also an example of a $-variable taking the value of a part of a term in an expression. Note again that the printout is generated also for the terms that do not match the wildcard

$-variables: Usage

- A $-variable can be used as
  - A preprocessor variable, referred to by enclosing it in `` `': `$x1'. `` In this case, the current (at the point where it is referred to) value of the variable is turned into a string, which is then operated by the preprocessor
  - An expression in an executable statement, such as `id x = y + $z` or `Local expr = $expr0`
  - An algebraic object in an executable statement, such as `id f(x?) = g(x,$y);` here, `$y` can be anything but an expression (e.g., a symbol or an argument field)
- Note that substitutions where $-variables appear in patterns in the l.h.s., e.g., `id f($x) = something` are delayed substitutions, i.e., they are evaluated for each term at runtime when the identification takes place. One can prevent this behaviour and use the "current" value by using the $-variable as a preprocessor variable instead: `id f(`$x') = something`
- An important feature of $-variables is that they can be factorized in FORM. The only other type of object that can be factorized is expressions.

Flow Control: Compiler

- We previously considered how the programming flow control (and procedural programming) are implemented at the level of the FORM preprocessor, in particular, the #do cycle, and the conditional #if statement
- The preprocessor loops/conditionals are generally quick, which follows from the fact that no calculations are done at this level (apart from simple arithmetic). On the other hand, this also means that the preprocessor cannot take decisions based on the information about the expression (more precisely, about the current term). Such decisions have to be made at the level of the compiler
- FORM has flow control structures implemented at the level of compiler, too. We will consider the following environments: do, repeat, while, if.

Repeat

- We have already encountered the simplest cycle realised at the level of compiler

```
repeat;
statements;
endrepeat;
```

- This cycle continues while something is done on the active expressions by the statements inside (even if the expressions are left unchanged in the end)
- Note that FORM allows one to insert end-of-module instructions inside of the cycle; this can be very useful, e.g., when one calculates something recursively, so that one can drop expressions calculated at previous steps:

```
repeat;
$i = $i+1;
Local expr`$i' =expr{`$i'-1} + expr{`$i'-2};
drop expr{`$i'-3};
.sort
endrepeat;
```

[Example file: fibonacci.frm]

Do

- The do cycle has the following syntax

```
do $i = min, max, {increment};
statements;
enddo;
```

- The loop counter has to be a $-variable, and it, as well as the other parameters, is evaluated at runtime. All of them have to evaluate to integers
- The cycle runs term-by-term
- Note that the evaluation at runtime slows down the execution, which means that the preprocessor #do cycle will usually be more efficient
- Some manipulations, however, are not possible without evaluation, e.g.,

```
id f(y?$x) = g(y);
do $i = 0, count_(x,1);
   id $x = $x * g(y);
enddo;
```

[Example file: compiler_do.frm]

Here `$x` and the power of `x` in the current term are evaluated at runtime and can be different for each term, so this cannot be realized in the preprocessor

If

- The syntax of the if statement is standard and very similar to other programming languages:

```
if (condition) statement
```

or

```
if (condition);
statements;
elseif (condition);
statements;
…
else;
statements;
endif;
```

In the first variant, only one statement follows the condition operator, and there is no semicolon in the end

In the second variant, at least the endif is necessary; both elseif and else are optional

The conditions have to be included in brackets

- The conditions can be quite complex, but have to be built of simple numerical comparison blocks, such as, e.g., `($i <=5)`, where the $-variable has to evaluate to a numerical value
- There are several functions that can be used in the conditions to check information about the current term and get one of the numerical arguments in these numerical comparison blocks

If – Conditions

[Example file: compiler_if.frm]

- `count(x,n)`
  Here, `x` is a variable (allowed: symbols, dotproducts, functions, vectors), `n` is the weight (an integer number, can be positive or negative).
  The function returns the power of the variable x with the weight n.
  Several options are allowed for vectors, such as
  `count(p+vdf?set,n),`
  where the presence of each of the options (each of them can be omitted) means:
  `v`: loose vectors with an index are taken into account;
  `d`: vectors inside dotproducts are taken into account;
  `f`: vectors inside tensors are taken into account;
  `?set`: a set of functions; vectors inside those functions (assumed to be linear in a given vector) are taken into account
  The default options are `+vdf`.

If – Conditions

[Example file: compiler_if.frm]

- `match(options,pattern)`
  Here, the argument is any valid l.h.s. of the `identify` statement, with possible options such as `once, only`, etc. The function returns the number of matches that is produced by the pattern in the current term
- `expression(list of expressions)`
  Here, the argument is a list of expressions, and the function returns 1 (0) if the current term belongs (does not belong) to any of the expressions in the list
- `occurs(list of variables)`
  Here, the argument is a list of variables, and the function returns 1 (0) if any variable from the list occurs (does not occur) in the current term, including inside function arguments
- `multipleof(natural number)`
  When this object is compared with another number (e.g., the result of evaluation of something), and that number is a multiple of the argument of this function, the two numbers match
- `coefficient`
  Represents the numerical coefficient of the current term

While

[Example file: while.frm]

- The syntax of the while loop is
  ```
  while (condition) statement;
  ```
  or
  ```
  while (condition);
  statements;
  endwhile;
  ```
- This cycle is similar to the `repeat...endrepeat` environment, whereas the condition works in the same way as that of `if … endif`.
- The operation here is again term-by-term, so for each term that reaches the `while (condition)` part, the condition is checked, if it is matched, the statements inside are executed, and when the `endwhile` is reached, the control is transferred to the `while (condition)` line again. If the condition is not met by the current term, statements after the `endwhile` are executed.