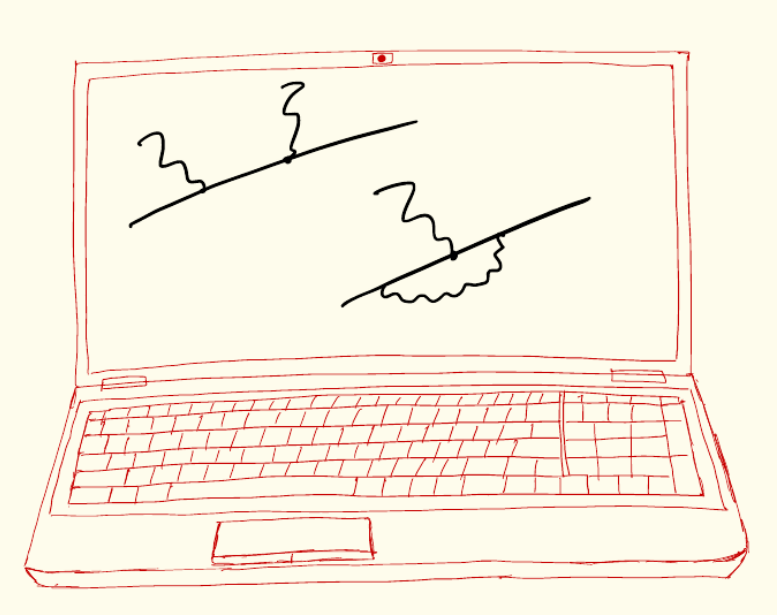
# Computer Algebra for Feynman Graphs



5.

# In this Lecture

- FORM Preprocessor – continued: #if and #switch
- Procedural programming in FORM
  - a few more notes on preprocessor variables
  - preprocessor macros
  - preprocessor procedures

# FORM Preprocessor: #if

- Conditional statements can be included at the level of the preprocessor by using the directives #if, #elseif, #else, and #endif:

```
#if (condition)
  what to do;
#elseif (another condition)
  what to do;
 …
#else
  what to do;
#endif
```

- Matching of `#if` with an `#endif` is mandatory; `#elseif` and `#else` are not
- Conditionals can be nested (also together with constructions such as `#do...#enddo`), but not interlaced as in `#do … #if … #enddo … #endif`
- The condition is a string can be a logical expression or a numerical value, where 0 corresponds to false and any other value to true. If the condition cannot be interpreted as a logical expression or a numerical value, it evaluates to false

# FORM Preprocessor: #if

- The behaviour of #if, #elseif, #else, and #endif is as usual: if one condition is matched, the corresponding block of instructions (the text between that condition and the next #elseif, #else, or #endif, whichever comes first) is executed, and then the preprocessor skips to the input after the corresponding #endif.
- Conditions can use comparison, which is numerical comparison in case the compared strings can be interpreted as numbers, and string comparison (lexicographic) otherwise
- Conditions can also be composite, with elementary conditions combined using the AND (`&&`) and OR (`||`) operators; there is no negation operator such as `!`
- Examples of valid conditions:

```
#if `i'
#if (`a' != `b')
#if (((`a' >=`b') && (`c' == `d')) || (`x'<=`y'))
```

# FORM Preprocessor: #if

- Apart from the conditions involving strings (preprocessor variables), there are functions that allow inquiries about objects:
- `exists(expr)`: returns 1 if an expression named expr exists, 0 otherwise
- `isdefined(prvar)`: returns 1 if the variable prvar been defined, 0 otherwise
- `isnumerical(expr)`: returns 1 if the expression expr contains a single term and it is a number, 0 otherwise
- `termsin(expr)`: returns the number of terms in the expression expr
- `maxpowerof(sym), minpowerof(sym)`: returns the maximum/minimum power of the symbol sym, if those are specified (see below for explanation); otherwise, it returns the general maximal/minimal power for symbols. These can only be used inside the conditions. Otherwise (e.g., in evalulations) one can use the functions `maxpowerof_(sym)` and `minpowerof_(sym)`
- Note: min/max powers of a symbol can be specified in the declaration of that symbol: `symbol x(m:n)`, where `m` specifies the min and `n` the max power. Both `m` and `n` them have to be numbers, and either `m` or `n` can be omitted.

# FORM Preprocessor: #switch

- Another multiple-choice conditional construction is the #switch directive:

```
#switch `var'
#case str1
    what to do;
#break
#case str2
    what to do;
#break
…
#default
    what to do;
#break
#endswitch
```

- `var` is a preprocessor variable whose value is compared with strings `str1`, `str2`, etc. (string comparison only, with no special actions for numbers)
- `#default` corresponds to the case when no matches are detected
- All `#break`'s and `#default` are optional

# FORM Preprocessor: #switch

- Note that `#switch` works in a fall-through manner: after the first match and the execution of the corresponding block of statements, all the subsequent statements in a switch environment are executed too, without checking the corresponding `#case` conditions, unless a #break is present:

[Example file: switch.frm]

```
#define i "2"
#switch `i'
#case 1
#message 1
#case 2
#message 2
#case 3
#message 3
#default
#message default
#endswitch
```

A match is here

And all subsequent statements are executed

# Preprocessor Variables and Macros

[Example file: macros.frm]

- Preprocessor variables can be defined
  `#define x “2”;`
  and redefined:
  `#redefine x “5”;`
  They can also be removed:
  `#undefine x;`
- Preprocessor variables are added on top of a stack: making `#define` a variable that had already been defined creates a new instance of that variable, not redefines it. Removing the current instance with `#undefine` makes the preprocessor to use the previously defined instance (if there is one)

```
#define x “2”;
Local F1 = `x’;
#define x “3”;
Local F2 = `x’;
#undefine x;
Local F3 = `x’;
=> F1 = 2, F2 = 3, F3 = 2;
```

# Preprocessor Variables and Macros

- Preprocessor variables can be defined as macros having arguments and parameters: [Example file: macros.frm]

```
#define x(a,b) "(`~a'+f`~b'-`~c')";
```

  Here, `a` and `b` are arguments, and `c` is a parameter.

- Arguments can be numbers and strings (such as preprocessor variables)

```
Local F1 = `x(1,2)';
Local F2 = `x(`i',`j')';
Local F3 = `x(x,y)';
```

- Delayed substitution: if the name of a variable used in a macro is prefixed with a tilde (~), its value is substituted at the point when the macro is used. In that, such parameters behave similarly to arguments (note the syntax):

```
#define c "0";
#define x(a,b) "`~a'+`~b'-`~c'";
#define y(a,b) "`~a'+`~b'-`c'";
Local F1 = `y(1,2)';
#redefine c "3"
Local F2 = `x(1,2)';
=> F1 = 3, F2 = 0
```

Here, `` `~c' `` is a parameter whose value is to be substituted later, when the macro is used

Here, `` `c' `` is the value of the preprocessor variable `c`, and is substituted immediately

# Preprocessor Procedures

- The FORM preprocessor allows one to define procedures that can be called later in the program:

```
#procedure procname(arg1,arg2,…,argN)
   body of the procedure
#endprocedure
```

- The arguments are optional: `procname()` is the name of a procedure that takes no arguments
- Argument field wildcards (as in functions) are allowed in the definition, such as `procname(a,?b)` – a procedure with at least one argument
- The procedure's arguments – `arg1,…,argN` – are preprocessor variables, so they have to be referred to accordingly – with backquote/quote pairs – in the body of the procedure: `` `arg1', … ``
- The scope of variables: in case variables with the same names as those of the argument variables (or local variables used inside of the procedure) are already defined in the main program, their invocation in the procedure creates new instances of those variables. After the procedure terminates, the definitions are restored to the "old" ones

# Preprocessor Procedures: Input

- Procedures can be defined in the main text of the program; in this case their definitions are stored in/read from a memory buffer
- An alternative is to put each procedure in a dedicated file, which has to be named `procname.prc` (the file name is the same as that of the procedure; the default extension for the procedure files can be changed). FORM looks for this file first in the current directory, and then searches directories indicated in the search paths
- Another alternative is to include procedures that are to be used in the current program in a header file `header.h`, which can then be included in the main input by the `#include` instruction:
  `#include header.h`
- Which way is better to use depends on the situation at hand. The first and last methods can be faster (procedures are kept in memory rather than read from files); they can also prevent issues when FORM picks the wrong procedure file (e.g., when the user forgets to create it but there is a file with the right name in another directory). The second method can work better when there are many procedures (especially when they are lengthy/created by someone else)

# Preprocessor Procedures: Calling

- To call a procedure, one has to use the following syntax:
  `#call procname(arg1,…,argN);` [Example file: procedures.frm]
- The number of arguments in a procedure call has to match that in the definition of the procedure, a mismatch causes an error
- The arguments are strings (explicit or referred to by preprocessor variables), separated by commas. If a string contains a comma, it has to be escaped: `\,`. A string can even contain a line break, which also has to be escaped with a `\` and the string has to be continued on the next line:

```
#call myproc(1,ac,a comma\,,hello\
there)
```

- Delayed substitution: prefixing a preprocessor variable with a tilde makes it to be substituted by the definition that is valid inside of the procedure. When this needs to be prevented, argument has to be prefixed with an exclamation mark:

```
#define b "a`~c'"
#call proc(b,!b)
```

# Delayed Substitution in Procedures

- An example of a procedure definition and call that illustrates these features of the delayed substitution: [Example file: procedures.frm]

```
autodeclare symbol c;
#define b "c`~a'";
#define a "2";

#procedure proc(a,?b)
   #redefine a "3"
   #message Arguments: `a',`?b'
#endprocedure

#message a=`a', b=`b'                  => a=2, b=c2
#call proc(`a',`b',`!b',`~a',`a')
                                       => Arguments: 3,c3,c2,3,2
```

- Note that the first argument is redefined in the procedure, so prefixing it with an ! does not do anything.

# Example: Recursive Calculations

- Let us calculate the Hermite polynomials recursively, using the following recursion relation

$$H_n(x) = 2xH_{n-1}(x) - 2(n-1)H_{n-2}(x),$$

togegher with

$$H_0(x) = 1,\ H_1(x) = 2x.$$

```
cfunction H;
symbol x, p;

#procedure Hermite(H,n)
  repeat;
  id `H'(0) = 1;
  id `H'(1) = 2*x;
  id `H'(`n'?) = 2*x*`H'(`n'-1) - 2*(`n' - 1)*`H'(`n'-2);
  endrepeat;
#endprocedure

Local HermiteP = H(10);
#call Hermite(H,p)
```

In the call, H and p have to be the names of a defined function and a defined symbol, since they are used in that sense in the procedure