1 A short manual of FORM

Symbolic Manipulation for the expansion of Feynman diagrams can easiest be done with FORM. In order to use FORM under linux on the NIKHEF computer systems one should make the alias

```
alias form $\sim$form/linux/bin/form
```

One prepares a file with a form program of which the extension should be .frm, eg test.frm. This program is then executed with

```
form test
```

which produces output on the screen. In case it is executed with

```
form -f test
```

it produces both output on the screen and (more completely) in the file test.log. This file can then be put in ones favorite editor. FORM can also produce other files, depending on the program.

FORM programs consist of modules. Each module again consists of declarations, definitions and statements. Each module is compiled when its turn has come and then executed. Hence at the level of modules FORM acts as an interpreter, at the level of individual statements as a compiler. This gives a high degree of flexibility as will be seen.

Modules are ended with instructions of which the name starts with a period. The most important instructions are for our purposes:

- .end Executes the current module and terminates the program afterwards.
- .sort Executes the current module, clears the buffers and continues with the next module.

Other module instructions will not play a role in this course.

1.1 General

The names of commands and built in variables are case insensitive. The built in variables all have a name that ends in an underscore. User defined variables are case sensitive and are not allowed to contain the underscore character. There is one exception to this last rule. User defined variables can have any characters in their name provided the name starts with the character valid name for a variable. It is different from $[A+B]$. The built in imaginary variable i is given by `i_` which is the same as `I_`. All statements inside the modules should be terminated with a semicolon.

1.2 Modules

The contents of each module can consist of any of the following in the given order:

- Declarations
- Settings
- Definitions
- Executable Statements
- Output control

None of these has to be present but if more than one type occurs the relative order has to be as specified above. Generically they are all called statements as well.

1.3 Declarations

The important declarations are:

- Symbol Declares the given objects as regular commuting variables.
- Vector Declares the given objects as vectors.
- Index Declares the given objects as indices (of vectors, functions or tensors)
- CFunction Declares the given objects as commuting functions.
- Function Declares the given objects as non-commuting functions.
- Tensor Declares the given object to be a (commuting) tensor. Tensors can have only indices or vectors for their arguments.

In addition one can put the word AutoDeclare in front of such a declaration. This means that all variables of which the name starts with the characters given and is not otherwise declared will become of the mentioned type. More than one variable can be declared at the same time and the notation $i1, \cdots, i12$ is understood as the sequence of the twelfe variables $i1$ till $i12$.

Vectors either have a single index between parentheses or are part of a dotproduct. Functions have zero or more arguments. The arguments can be empty. An empty argument is interpreted as zero. If a tensor has a vector for an argument it is interpreted that there was in index there that has been contracted with the index of the vector. Indices can have customized dimensions, but we will not look at that here. The default dimension for indices is four.

1.4 Settings

There are not many settings that we will look at. The most important in this course is the Format statement.

```
Format C;
```

will format potential output in such a way that a C compiler can translate the formulas.

```
Format Fortran;
```

will format potential output in such a way that a Fortran compiler can translate the formulas.

```
Format;
```

will switch back to the default FORM mode.

1.5 Definitions

A definition defines an expression as an object for symbolic manipulation. There are several types of expressions, but we will only consider the local expressions. They are defined as in

```
Local F = (a+b)^4;
```

This defines the expression F and its initial value is $(a+b)^4$. The first thing FORM will do is work out the power as $a^4 + 4a^3b + \cdots$. One is allowed to use previously defined expressions in the right hand side of a definition. For reasons of efficiency it is best that there is a .sort instruction before such a previously defined expression is being used.

Expressions are the objects we are interested in. They consist of terms like $4a^3b$. Statements act on the terms of an expression. At the end of the module the results of these manipulations are brought together and sorted into a 'normal ordered' object again.

There are two statements that have the same status as a definition and should be used together with the definitions before the regular statements. They are:

```
Drop [names of expressions];
```

and

```
Skip [names of expressions];
```

The Drop statement removes expressions from the system. The mentioned expressions can still be used in the right hand side of expressions or statements in the current module. After the current module there is no more memory of the given expression(s). If no names of expressions are given all currently defined expresseions will be dropped. This can be done as in

```
Drop;
Local M = Amp*AmpC;
```

The expressions Amp and AmpC will be dropped and a new expression that is the product of the two will be created.

The skip command just instructs FORM that the mentioned expressions are not to be treated by the statements in the current module. It is a way to selectively act on some expressions while not action on others. If no expressions are mentioned all currently active expressions will be skipped as in

```
Skip;
Local AmpC = Amp;
```

where all expressions except for AmpC will be skipped. The nskip statement does the opposite:

```
Skip;
Nskip Amp;
```

will skip all expressions except for the expression Amp.

1.6 Executable Statements

The statements come in the largest variety. We will see however only a small number of them. They act on the individual terms of all expressions The most important statement is the id or identify statement as in

```
id pattern = expression;
```

In this case the pattern will be looked for in the terms of the active expressions and replaced by the right hand side expression as in:

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

Notice that the pattern is matched and taken out as many times as possible and only after that the right hand side is inseterd as many times as the pattern fit. Hence this will cause no problems with a statement like

```
id a = a+1;
```

If repeated action is needed one should put the statement(s) inside a repeat loop:

```
repeat;
    id  a^2 = a+1;
endrepeat;
```

Generic variables or wildcards in the pattern are indicated by a trailing questionmark. In the right hand side the questionmark is not needed:

```
id a^n? = a^(n+1)/(n+1);
```

A very special type of wildcarding is done with the arguments of functions. These wildcards stand for an unspecified number of arguments of the function. Example:

```
repeat;
  id f(i1?,i2?,?a)*f(i2?,i3?,?b) = f(i1,i3,?a,?b);
endrepeat;
```

Here the 'argument field' wildcards are given by a questionmark followed by a name. They will match with any number of arguments, also zero. Note also that because i_2 occurs twice in the pattern both occurrences have to be identical. The above is a way of stringing matrices together.

Another useful statement is the trace statement which comes in two varieties. We will use only the trace in 4 dimensions which is given by the trace4 statement. The statement

```
Trace4,i;
```

will take the 4-dimensional trace of all gamma matrices that are marked as belonging to the fermion line i. I is either an index or a (short) positive number.

The multiply statement multiplies all terms with the expression given in the statement:

```
Multiply,right,replace_(u,ub,ub,u,v,vb,vb,v);
```

This multiplies on the right. It is also possible to multiply on the left and if there is no keyword the system either multiplies on the right or on the left, whatever it likes. The `replace_` function is described in the subsection on special functions.

There is also an if-statement in which questions are asked for each term and depending on the answer the statements between the if-statement and the corresponding endif statement will be executed. For more details we refer to the complete FORM manual.

1.7 Output control

Here we mention for instance whether we want the output of the current module to be printed. We can also specify whether we want a level of brackets in the output. The print statement has some varieties:

```
Print;
Print +f;
Print +s;
Print expressionnames;
```

In the first case all active expressions (that are not dropped or skipped) will be printed. In the second case all will be printed but if FORM was called with the -l flag they will be printed only in the log file. In the third case all will be printed with only a single term per line. Finally in the fourth case only the mentioned expressions will be printed. All these options can be combined. In that case the +f and/or +s come before the name(s) of the expression(s).

The bracket statement specifies whether some objects should be taken outside brackets when the output is printed. This can make the output easier to read. Each bracket is started on a new line.

```
Bracket x,y,f;
```

In this case (x and y symbols, f a function) all powers of x and y and all occurrences of f are placed outside brackets. Each different occurrence of powers of x and y and the function f defines a new bracket. Note that this has also an influence on the ordering of the terms because now powers of x which go outside the brackets are more significant than powers of a which will be inside the brackets. Without bracket statement this might be different depending on whether a is declared before x.

1.8 Special Functions

Amoung the various special functions in FORM are the Dirac gamma matrices and a number of delta functions. There is also the Levi-Civita tensor and much more. We will only need the ones mentioned here.

The Dirac gamma matrices are indicated by `g_`. They are noncommuting objects that have in principle two arguments: The first argument is either an index or a (small) positive integer that indicates to which spinline the matrix belongs. Matrices of different spinlines will commute with each other. The second argument is either an index or a vector. If it is a vector it is a shorthand notation for an index that is contracted with the index of the vector. There are several special indices for the second argument: `5_` indicates γ_5 while `6_` indicates $(1+\gamma_5)$ and `7_` stands for $(1-\gamma_5)$. If the second argument is absent we have the unit matrix. We have several shorthand notations:

```
gi_(j) = g_(j)
g5_(j) = g_(j,5_)
g6_(j) = g_(j,6_) = gi_(j)+g5_(j)
g7_(j) = g_(j,7_) = gi_(j)-g5_(j)
```

A string of gamma matrices can be written together as in

```
g_(j,nu,p1,mu,p2) = g_(j,nu)*g_(j,p1)*g_(j,mu)*g_(j,p2)
```

The Levi-Civita tensor `e_` is a totally antisymmetric tensor. In four dimensions we have $e^{0123} = 1$. It has close relations with $gamma_5$:

```
Indices mu,nu,ro,si;
Local F = g5_(j)*g_(j,mu)*g_(j,nu)*g_(j,ro,)*g_(jsi);
Trace4,j;
Print;
.end
   F = e_(mu,nu,ro,si);
```

For this course we need to know only two delta functions. The Dirac delta function $\delta^{\mu\nu}$ and the `replace_` function. The Dirac delta is the function `d_` with two index arguments. As repeated indices are automatically summed over most `d_` functions disappear quickly and have the effect of renaming indices. Something similar is acchieved for other objects with the `replace_` function. It should have an even number of arguments. The arguments come in pairs and when a term has a replace function its effect is that in the term everywhere the first element of the pair is replaced by the second element. Example

```
Multiply replace_(u,ub,ub,u,v,vb,vb,v);
```

Each term will be multiplied by this replace function after which in the term u is replaced by ub, ub by u, v by vb and vb by v. Then the replace function is removed as it has done its work. This works much faster and less complicated than trying to do this with a number of id-statements.

1.9 The Preprocessor

Just like the C compiler FORM has a rather powerful preprocessor with a whole range of preprocessor instructions and its own preprocessor variables. Preprocessor instructions start with the character `#` as in

```
#include amplitude.h
```

which would be replaced by the contents of the file amplitude.h. The preprocessor has its own variables. When they are used they are between a matching set of backquote and quote. When this is encountered a textual replacement is made and compilation continues at the beginning of the content of the variable as in

```
    #define MAX "20"
#do i = 1,‘MAX’
        id a‘i’ = a{‘i’+‘MAX’};
    #enddo
```

In the define instruction MAX is defined as the string 20. Preprocessor variables contain always character strings unless the preprocessor calculator is invoked in which case an attempt is made to interpret the string as an arithmatic expression.

In the do instruction `‘MAX’` is replaced by the string 20and the program will go through the loop twenty times, each time generating an instruction.

The first time i is the string 1 etc. The curly brackets in the right hand side invoke the preprocessor calculator and hence the first time `{1+20}` is evaluated and replaced by the string 21 and hence we generate in total twenty statements of the type

```
id a1 = a21;
id a2 = a22;
     ...
id a20 = a40;
```

Another preprocessor construction is the procedure. each procedure can be part of the regular input stream and is then read into memory and kept there, or it resides in a separate file which has the name of the procedure and the extension .prc but in our examples we have put the procedure squareamplitude in the header file amplitude.h. procedures can have arguments and FORM is very careful with the syntax of the procedure instruction and the corresponding call instruction that invokes the procedure. The procedure instruction is given by

```
#procedure name(argument(s))

    contents

#endprocedure
```

The variables in the arguments are preprocessor variables and hence when used should be enclosed in a matching pair of backquote and quote. It is not necessary to have arguments. Because the procedure is handled entirely by the preprocessor it is actually a gigantic macro. The procedures are called by the call instruction as in

```
#call squareamplitude(Amp,Mat)
```

in which case the instruction is replaced by the contents of squareamplitude in which the first argument is replaced by the string Amp and the second by the string Mat. Note that procedures can call other procedures and even recursions are possible here provided that one takes care that the recursion terminates.

1.10 Dollar variables

Apart from regular algebraic variables and the string valued preprocessor variables FORM knows a third type of variables that can be accessed both by the preprocessor and the algebraic execution unit. These variables have a name that starts with a `$` sign. These variables can contain numbers or small algebraic expressions. When referred to between a matching pair of a backquote and a quote, their value is translated into a string and used as a preprocessor variable. Without the backquote and quote their value is substituted during algebraic execution at the term level. These variables are given a value by the preprocessor if preceeded by the character `#` and during algebraic execution without this character. Example:

```
    #$count = 0;
    $count = $count + 1;
.sort
    #do i = 1,`$count'
        etc
```

During compilation the variable `$count` gets the value zero. Then during execution each time the program passes here (for each term) its value is raised by one. After the .sort it will contain the number of terms that FORM had before the sort. This number is then used by the preprocessor as a parameter in the do loop. There are more ways to give values to the dollar variables and one can do very complicated things with them. This is however outside the scope of this small manual.