TWO

In the previous chapter we saw an example in which we had to provide the same statement several times. This is not convenient. To make Form more workable, we have a completely different level of programming with its own variables: the preprocessor.

The preprocessor defines a type of control language with which we edit the input before we give it to the compiler. It has its own variables.

Preprocessor variables are distinguishable from regular variables because when we refer to them they have to be enclosed between a backquote-quote pair as in `` `i' ``. This way there can be no confusion with the regular variables.

Preprocessor variables are text string objects. They can be defined with the #define instruction as in

```
#define i "2"
#define j "3"
#define k "xx1"
Symbols x‘i’,x‘j’,x‘i’‘j’,‘k’;
Local F = x‘i’+x‘j’+x‘i’‘j’+‘k’;
Print;
.end
```

```
F =
   xx1 + x23 + x3 + x2;
```

The backquote-quote notation allows nesting as if they are brackets:

```
#define x1 "y"
#define x2 "yy"
#define x3 "yyy"
#define i "2"
Symbol ‘x‘i’’;
Local F = ‘x‘i’’;
Print;
.end
```

```
F =
   yy;
```

The first major use of these variables is the preprocessor do loop:

```
#define MAX "4"
Symbols x1,...,x‘MAX’;
#do i = 1,‘MAX’
  Local F‘i’ = (x1+...+x‘i’)^2;
#enddo
Print;
.end
```

```
F1 =
   x1^2;
F2 =
   x2^2 + 2*x1*x2 + x1^2;
F3 =
   x3^2 + 2*x2*x3 + x2^2 + 2*x1*x3 + 2*x1*x2 + x1^2;
F4 =
   x4^2 + 2*x3*x4 + x3^2 + 2*x2*x4 + 2*x2*x3 + x2^2 + 2*x1*x4 + 2*x1
   *x3 + 2*x1*x2 + x1^2;
```

We see a few new things here that all involve the preprocessor. First is the loop which runs from 1 to 4 (in this case). The increment is assumed to be +1. If we want to use a different increment we have to provide it as an extra argument as in `#do i = ‘MAX’,1,-1`. The loop is ended with the `#enddo` instruction. Loops can be nested.

We also use the `...` notation. This indicates that FORM should guess a pattern and provide the complete sequence. Hence `x1,...,x4` is translated by the preprocessor into `x1,x2,x3,x4` and `x1+...+x4` is translated into `x1+x2+x3+x4`. When the pattern is a bit harder to find for FORM (because the numbers are not at the end or there is not a single object before the number) the objects should be placed between a pair of $<$ and $>$ characters. Unfortunately this notation cannot be nested.

```
 #define MAX "4"
 Symbols <x1y5>,...,<x`MAX'y{`MAX'+4}>;
 Format 72;
 #do i = 1,`MAX'
   Local F`i' = <x1y5>+...+<x`i'y{`i'+4}>;
 #enddo
 Print;
 .end
```

```
F1 =
   x1y5;
F2 =
   x2y6 + x1y5;
F3 =
   x3y7 + x2y6 + x1y5;
F4 =
   x4y8 + x3y7 + x2y6 + x1y5;
```

And again we see a new feature: the preprocessor calculator. If the preprocessor encounters a `{}` pair it looks at what is between the parentheses and if this is an expression that can be evaluated numerically it will do so and replace the whole object by the result. There are a few restrictions: The calculator works only over the integers and the maximum size of the numbers is defined by the size of a FORM word which is 32767 for 32-bits computers and 2147483647 for 64-bits computers. There are a few postfix operators for square roots and 2-logs. This is explained in the manual. If the string cannot be interpreted as a numerical expression it is passed to the compiler, including the parentheses. The compiler will try to interpret it as a set (we have not seen those yet). To make sure that the set with just one numerical element is not 'stolen' by the preprocessor we can add an empty element as in `{2,}` or `{,2}`. The comma makes it safe.

Preprocessor instructions can occur in the middle of (multi-line) statements.

```
Local F = f(x)
 #do i = 1,10
    +f(‘i’^2+1,x)
 #enddo
    ;
```

The restriction is that a preprocessor instruction is a single line. If it is too long one can continue into a next line by terminating the line of the preprocessor instruction with a backslash character. This is the same as in shell scripts.

We are now properly equipped to deal with the last homework assignment of the previous chapter. In principle a working program would look like

```
#define MAX "8"
Symbol x, j;
*
*   Expansion of ln(1-x)
*
Local F = sum_(j,1,‘MAX’,-x^j/j);
Print;
.sort

   F =
      - x - 1/2*x^2 - 1/3*x^3 - 1/4*x^4 - 1/5*x^5 - 1/6*x^6 - 1/7*x^7
      - 1/8*x^8;
```

```
    *
    *  The expansion of 1-exp(x) is sum_(j,1,‘MAX’,-x^j/fac_(j)) but
    *  that generates many terms beyond what is accurate. We can make
    *  an automatic cutoff on the power of y with the declaration here:
    *
    Symbol y(:‘MAX’),n;
    id  x = sum_(j,1,‘MAX’,-y^j/fac_(j));
    Print;
    .end

Time =       26.71 sec    Generated terms =          255
                F         Terms in output =            1
                          Bytes used      =           36

   F =
      y;
```

Unfortunately this is a bit slow because the program is generating mostly terms that are set to zero immediately. In addition we see the new function fac_ which is the factorial function.

Using the preprocessor do loop things go much faster:

```
#define MAX "8"
Symbol x, j;
*
*   Expansion of ln(1-x)
*
Local F = sum_(j,1,`MAX',-x^j/j);
Print;
.sort
```

```
 F =
     - x - 1/2*x^2 - 1/3*x^3 - 1/4*x^4 - 1/5*x^5 - 1/6*x^6 - 1/7*x^7
     - 1/8*x^8;
```

```
    *
    *  The expansion of 1-exp(x) is sum_(j,1,‘MAX’,-x^j/fac_(j)) but
    *  that generates many terms beyond what is accurate. We can make
    *  an automatic cutoff on the power of y with the declaration here:
    *
    Symbol y(:‘MAX’),n;
    #do i = ‘MAX’,1,-1
      id  x^‘i’ = sum_(j,1,{‘MAX’-‘i’+1},-y^j/fac_(j))*x^{‘i’-1};
    #enddo
    Print;
    .end

Time =       0.00 sec    Generated terms =        255
               F         Terms in output =          1
                         Bytes used      =         36

   F =
      y;
```

This is much better already. If however we change the value of MAX to 18 we get:

```
Time =       12.62 sec    Generated terms =      262143
                F         Terms in output =           1
                          Bytes used      =          36

   F =
      y;
```

and we see that we still generate lots of terms. Next we can put a .sort inside the loop. This will make simplifications after each substitution:

```
    Symbol y(:‘MAX’),n;
    #do i = ‘MAX’,1,-1
      id  x^‘i’ = sum_(j,1,{‘MAX’-‘i’+1},-y^j/fac_(j))*x^{‘i’-1};
      .sort: i = ‘i’;

Time =       0.00 sec    Generated terms =          18
               F         Terms in output =          18
                  i = 18 Bytes used      =         460
    #enddo
                           .
                           .
                           .
Time =       0.00 sec    Generated terms =         102
               F         Terms in output =          30
                  i = 13 Bytes used      =         764
                           .
                           .
                           .
    Print;
    .end

Time =        0.00 sec    Generated terms =          1
                F         Terms in output =          1
                          Bytes used      =         36

   F =
      y;
```

Now the program is fast again because after each step we combine identical terms. With MAX equal to 50 we take now 0.15 sec. This can still be brought down by changing the critical id-statement so that it takes also the powers of y that are there already into account:

```
    #do i = ‘MAX’,1,-1
      id  x^‘i’*y^n? = sum_(j,1,{‘MAX’-‘i’+1}-n,-y^j/fac_(j))*
                          x^{‘i’-1}*y^n;
      .sort: i = ‘i’;
    #enddo
```

In that case we need only 0.05 sec. and for MAX is 100 it takes now 0.66 sec. As you can see, one can improve some programs considerably.

The next question is whether we can make something that is the equivalent of subroutines. In Form these are called procedures (they are not real subroutines but look a bit like them). They are managed by the preprocessor and are more like very big macro's.

There are two ways to introduce procedures:

1. Put the procedure in the program file before its first use.
2. Put the procedure in a file with the same name as the procedure and the extension .prc

First the syntax of a procedure.

The first line of a procedure must be like

```
    #procedure name(arguments)
```

The arguments are optional. If there are none, the first line should be

```
    #procedure name
```

If the procedure is in a file (named "name.prc") the #procedure should be the first characters in the file. The last line of a procedure must be

```
    #endprocedure
```

Between the first and the last line you can use any regular programming.

The arguments of the procedure are treated as preprocessor variables. This means that when you use them, they should be enclosed in a backquote-quote pair as in

```
#procedure derive(x)
*
*   Procedure to differentiate polynomials in the symbol `x'.
*   The calling routine must have declared the symbol n.
*
    id `x'^n? = n*`x'^(n-1);
#endprocedure
```

If we put this procedure in the file derive.prc we can use it in a program with the #call instruction:

```
 Symbols a,b,c,n;
 Local F1 = a*b^3+2*b^6+b^-10;
 Local F2 = a*b^3*c^2+2*b^6*c^5+c;
 Print;
 .sort

F1 =
   b^-10 + 2*b^6 + a*b^3;

F2 =
   c + 2*b^6*c^5 + a*b^3*c^2;

 #call derive(b)
 Print;
 .sort

F1 =
    - 10*b^-11 + 12*b^5 + 3*a*b^2;
```

```
F2 =
   12*b^5*c^5 + 3*a*b^2*c^2;

 #call derive(c)
 Print;
 .end

F1 = 0;

F2 =
   60*b^5*c^4 + 6*a*b^2*c;
```

Of course procedures can call other procedures. They can even call themselves. In that case the user should take provisions that the recursion terminates sooner or later.

The arguments of procedures are preprocessor variables. The preprocessor variables are put on a stack and when the procedure is finished the stack is popped. If you have defined other preprocessor variables inside the procedure they are popped off the stack as well:

```
    #procedure testdef(x)
      #define yy "25"
      #write "\n   The value of x is ‘x’ and the value of yy is ‘yy’\n"
    #endprocedure

    #define x "10"
    Symbol a;
    Local F = a^2;
    #call testdef(3)

   The value of x is 3 and the value of yy is 25

    #write "\n   The value of x is ‘x’\n"

   The value of x is 10

    #write "\n   The value of x is ‘x’ and the value of yy is ‘yy’
p2-12.frm Line 10 ==> Undefined preprocessor variable yy
```

```
Program terminating at p2-12.frm Line 7 -->
```

This is the way FORM deals with local variables. Of course, if you refer to a preprocessor variable that has been defined in the main program, but not in the procedure, FORM will find that definition, because it will look for the first variable by that name that it finds, starting at the top of the stack.

Things become a bit more tricky, if we would like to change the value of the variable in the main program. For this we have the #redefine instruction.

```
#procedure testdef(x)
  #write "\n   The value of x is ‘x’ and the value of yy is ‘yy’\n"
  #redefine yy "25"
  #write "\n   The value of x is ‘x’ and the value of yy is ‘yy’\n"
#endprocedure

#define x "10"
#define yy "ax"
Symbol a;
Local F = a^2;
#call testdef(3)
```

```
The value of x is 3 and the value of yy is ax


The value of x is 3 and the value of yy is 25

 #write "\n   The value of x is ‘x’\n"

The value of x is 10

 #write "\n   The value of x is ‘x’ and the value of yy is ‘yy’\n"

The value of x is 10 and the value of yy is 25

 Print;
 .end

F =
   a^2;
```

If you use a #redefine instruction when the variable has not been defined before, the effect is the same as a #define instruction. Hence you cannot push a variable from a procedure into the global space. You need to define all such variables before calling the procedure (we will see a completely different way to do such things in the third lecture).

To recapitulate:

- #define always makes a new instance of its variable.
- The variables are on a stack.
- When we use a variable, its instance that it highest on the stack is used.
- When we leave a procedure, the stack is popped to the size it had at the moment that the procedure was called.
- #redefine looks for the instance of the variable that is highest on the stack and modifies its value.
- If #redefine does not find the variable on the stack, it acts as #define.

As you see, although the concept of what is local and what is global is a bit different from what it is in FORTRAN or C, it is flexible enough. One of the reasons that it is different is because procedures are not real subroutines with compiled code. Effectively they are big macro's. All that they do is preparing statements for the compiler or interpreting preprocessor instructions.

FORM does not have real subroutines; pieces of precompiled code that can be used when called during the algebraic execution. Such a thing would add an enormous amount of complexity, and probably make things slower, rather than faster. And think of the following:

Assume that my subroutine has a local symbol x. And assume that I did some sloppy programming and the output term would still contain this symbol. The main program may not recognize this symbol and everything would crash. It should be clear that there is an essential difference between numerical and symbolic programs and hence we need different concepts. (In numerical programs you can have undefined variables in the input and in CA you can have them in the output).

The next useful feature of the preprocessor is the possibility to read input from other files. This is done, as in many other programming languages, with the #include instruction as in

```
    #include filename.h
```

It is advised to give such files the extension .h (which stands for header file). If this input is being echoed to the output, also the contents of the header files will be echoed. To avoid this with lengthy header files one can use the option

```
    #include- filename.h
```

The system will first look for the requested file in the current directory. If the file is not found there it will look in the directories that are specified in the environment variable FORMPATH which can be set in the shell from which Form is run. Other ways are explained in the reference manual.

In some cases we have directories with .h files for C programs (or other languages) and .h files for Form programs. At times this can be undesirable. In that case I use .hh for the Form header files.

Thus far we have not considered flow control. This is essential for good programming. FORM has this both at the algebraic level and at the preprocessor level. But the conditions that they use for their decisions are of a completely different nature. Here we will look at the preprocessor flow control. Of course the basic instruction is the #if instruction:

```
#if ( condition )
   code
#elseif ( othercondition )
   more code
#else
   even more code
#endif
```

The blank spaces in the above are optional. They represent my personal programming style. Note that the 'elseif' is a single word, unlike the 'else if' in the C language. Of course the #elseif and/or the #else do not have to be present.

Remember that in the previous lecture we saw that expressions exist only before and after the algebraic (execution) phase, while during the algebraic phase FORM deals only with terms. The compilation is outside the algebraic phase. This means that the conditions for the preprocessor #if may involve questions about expressions, while the conditions for the runtime if cannot. There the questions refer to the terms (and in both cases a few other things, like checks on numbers).

The condition can be numerical as in

```
#procedure addone(x,n)
#if ( ‘n’ > 0 )
  id  ‘x’ = ‘x’+1;
  #call addone(‘x’,{‘n’-1})
#endif
#endprocedure

Symbol x;
Local F = x;
#call addone(x,10)
Print;
.end

F =
   10 + x;
```

Although this example is a bit simplistic from the calculational viewpoint, it shows many essential features. First it shows the simplest use of the #if instruction. Second it shows a recursive call to the procedure addone. And then it shows how one should pass arguments to procedures. When addone is called with the parameter 10, we want to call it again with the parameter 9 and not with 10-1 which would be the case if we would not use the preprocessor calculator. Alternatively one can place the preprocessor parentheses in the #if instruction as in

```
#procedure addone(x,n)
#if ( {‘n’} > 0 )
  id  ‘x’ = ‘x’+1;
  #call addone(‘x’,‘n’-1)
#endif
#endprocedure
```

If one refuses to use the preprocessor calculator at all the program gets into an infinite loop, the reason being that `10-1-...-1` will be a string that is considered greater than the string 0.

Another condition is that we can ask for the number of terms in an expression. This refers to the number of terms that the expression has at the start of the module. We construct the program:

```
Symbol x;
Local F = x;
#do i = 1,10
  id  x = (x+1)^2;
  .sort
  #if ( termsin(F) > 13 )
    #breakdo
  #endif
#enddo
Print;
.end
```

The #breakdo instruction allows us to break out of the loop. It can have an argument which tells out of how many loops it has to break. The above instruction is equivalent to `#breakdo 1`. When we run the program we produce:

```
    Symbol x;
    Local F = x;
    #do i = 1,10
    id x = (x+1)^2;
    .sort

Time =       0.00 sec    Generated terms =          3
               F         Terms in output =          3
                         Bytes used      =         76
    #if ( termsin(F) > 13 )
      #breakdo
    #endif
    #enddo

Time =       0.00 sec    Generated terms =          9
               F         Terms in output =          5
                         Bytes used      =        124
```

```
Time =       0.00 sec    Generated terms =          25
               F         Terms in output =           9
                         Bytes used      =         220

Time =       0.00 sec    Generated terms =          81
               F         Terms in output =          17
                         Bytes used      =         412
    Print;
    .end

Time =       0.00 sec    Generated terms =          17
               F         Terms in output =          17
                         Bytes used      =         412
   F =
      676 + 4160*x + 13888*x^2 + 31776*x^3 + 54792*x^4 + 74624*x^5
       + 82432*x^6 + 74944*x^7 + 56472*x^8 + 35296*x^9 + 18208*x^10
       + 7664*x^11 + 2580*x^12 + 672*x^13 + 128*x^14 + 16*x^15 + x^16;
```

Once the number of terms in F is greater than 13 the program breaks out of the do-loop.

One can create composite conditions as in

```
#if ( ( ‘a’ > ‘b’ ) && ( ‘c’ > ‘b’ ) && ( ‘a’ >= ‘c’ ) )
```

FORM is very precise about the brackets. One cannot use

```
#if ( ‘a’ > ‘b’ && ‘c’ > ‘b’ && ‘a’ >= ‘c’ )
```

When mixing or and and operations the brackets are definitely essential:

```
#if ( ( ( ‘a’ > ‘b’ ) && ( ‘c’ > ‘b’ ) ) || ( ‘a’ >= ‘c’ ) )
```

If is not specified what FORM will do, if you do not place the brackets correctly. It may even change between versions.

Like the C language FORM also has a #ifdef as in

```
#ifdef `VAR'
#else
#endif
```

There is also the #ifndef instruction. Let us see how one can use this:

```
    #ifndef `NUM'
      #define NUM "14"
    #endif
    #write "\n   Computing the 4-dimensional trace of `NUM' \
            gamma matrices.\n"

   Computing the 4-dimensional trace of 14 gamma matrices.

    Index i1,...,i`NUM';
    Local F = g_(1,i1,...,i`NUM');
    Trace4,1;
    .end

Time =       0.03 sec    Generated terms =      31599
                F         Terms in output =      26931
                          Bytes used      =    1025268
```

The reason this is an interesting example is because we can define preprocessor variables in the command tail when we run FORM as in:

```
    form -D NUM=16 p2-17
```

which gives

```
    #ifndef `NUM'
      #define NUM "14"
    #endif
    #write "\n   Computing the 4-dimensional trace of `NUM' \
            gamma matrices.\n"

   Computing the 4-dimensional trace of 16 gamma matrices.

    Index i1,...,i`NUM';
    Local F = g_(1,i1,...,i`NUM');
    Trace4,1;
    .end

Time =       0.26 sec    Generated terms =     192357
               F         Terms in output =     163509
                         Bytes used      =    6290748
```

and we see that now the trace of 16 gamma matrices is computed. This can be particularly handy when you want to run many FORM programs from a shell-script or from a makefile.

The last major family of instructions for flow control is formed by the #switch instruction and what comes with it. The basic structure is

```
  #switch string
    #case value1
       code
    #break
    #case value2
       more code
    #break
        etc
    #default
       even more code
    #break
#endswitch
```

The string in the #switch instruction is often a preprocessor variable and the values in the #case instructions are strings as well. Form will take the initial string and compare it with the strings in the #case instructions. The first one that gives a match is taken and the code after it till the first #break that is encountered (either that or the #endswitch). After that execution continues after the #endswitch. It is best to give an example (from a real and successful program named Mincer):

```
#procedure integral(TOP,par)
*
#switch ‘TOP’
      .
      .
#case t1
 Multiply intt1;
 #call integt1
 #call integt2
 #call integt3
 #call integl1
 Multiply 1/epQ^2/int0;
#break
#case t2
 Multiply intt2;
 #call integt2
 #call integl1
 Multiply 1/epQ^2/int0;
```

```
#break
#case t3
 Multiply intt3;
 #call integt3
 #call integl1
 Multiply 1/epQ^2/int0;
#break
#case l1
 Multiply intl1;
 #call integl1
 Multiply 1/epQ/int0;
#break
#case tr
#break
#endswitch
*
```

This is a piece from the integration procedure in the minceex.h library for the integration of three loop massless propagator graphs. We do not give here all the cases, but a procedure like this allows for very transparent use in the calling routine:

```
    #include- minceex.h
    Off Statistics;
    Format nospaces;
    .global
    L   F = Q.Q^4/p1.p1^2/p2.p2^2/p3.p3/p4.p4^2/p5.p5;
    #call integral(t1,0)
    #call subvalues
~~~Answer in the Gscheme
    #call expansion(1)
~~~Answer in the Gscheme
    Print +f;
    .end

   F=
      -16+2*ep^-2-3*ep^-1+73/2*ep;
  0.01 sec out of 0.01 sec
```

We see here also the use of the library minceex.h. What this file contains is all declarations and procedures that are needed for a given class of problems. Once all procedures have been loaded this way, Form does not have to locate them when they are called. It avoids possible confusion if the FORMPATH were to contain many directories and some have files with identical names. The minceex.h library for instance contains 48 procedures and needs no other libraries or files. It is rather lengthy and contains 6173 lines (some of which is commentary).

It is allowed to combine cases and have ‘fall-throughs’.

```
#switch ‘PAR’
  #case 5
     id x^3 = x^5+5;
  #case 4
     id x^2 = x^3+15;
  #case 3
     id x^2 = x^7+1;
  #case 2
     id x = x^2-2;
  #case 1
     id x = x+1;
  #break
  #default
     #write "Parameter PAR has value ‘PAR’ which is out-of-range."
     #terminate
  #break
#endswitch
```

You see that if PAR has the value 5, there will be 5 statements to be executed. We also see the new instruction #terminate which terminates the program with an error condition. This termination is nearly immediate. Form will attempt to clean up its temporary files though.

The last instruction we will look at is the #write instruction. We have seen it already as a way to let the preprocessor write messages to the output. The instruction is however far more flexible. We can let it also write to files as in:

```
#write <filename> "string or other objects"
```

in which case the output will be in the file. If it is the first time this file is referred to it will be created (previous contents will be lost). Additional writes will add to the file. Which objects can be specified will become clear when we make actual use of the instruction.

The name of the file has to be enclosed between the < and >. When we do not specify anything the writing is to the screen and when we specify only <> the output is to the log file only.

What is the log file? When we call FORM we may specify the argument -l as in

```
form -l p2-19.frm
```

In this case FORM will create a file p2-19.log and write all its output in this file. In addition it writes on the screen. But.... Let us look at this program (p2-19.frm):

```
    Vector p1,...,p10;
    Local F = g_(1,p1,...,p10);
    Trace4,1;
    Print +f;
    .end
```

We ask here for the output to be printed, but we specify the argument +f. This tells Form that this output should go only to the log file p2-19.log. If we look what we produce on the screen, it is

```
        Vector p1,...,p10;
        Local F = g_(1,p1,...,p10);
        Trace4,1;
        Print +f;
        .end

    Time =       0.00 sec    Generated terms =        801
                   F         Terms in output =        693
                             Bytes used      =      32376
```

but in the file we find:

```
   Vector p1,...,p10;
   Local F = g_(1,p1,...,p10);
   Trace4,1;
   Print +f;
   .end

Time =       0.00 sec    Generated terms =        801
               F         Terms in output =        693
                         Bytes used      =      32376

   F =
      4*p1.p2*p3.p4*p5.p6*p7.p8*p9.p10 - 4*p1.p2*p3.p4*p5.p6*
      p7.p9*p8.p10 + 4*p1.p2*p3.p4*p5.p6*p7.p10*p8.p9 - 4*p1.p2*
      p3.p4*p5.p7*p6.p8*p9.p10 + 4*p1.p2*p3.p4*p5.p7*p6.p9*p8.p10
       - 4*p1.p2*p3.p4*p5.p7*p6.p10*p8.p9 + 4*p1.p2*p3.p4*p5.p8*
      p6.p7*p9.p10 - 4*p1.p2*p3.p4*p5.p8*p6.p9*p7.p10 + 4*p1.p2*
```

```
               etc. (411 more lines)
```

This way you can follow the statistics without having giant outputs messing up the screen, but afterwards you can still inspect those outputs.

Problem 3: The problem is to evaluate

$$\Box_P^{10} P.p_1^{10} P.p_2^{10} P.p_3^{10}$$

Of course it may be better to start with lower powers. And the hint is to look in the manual for special functions that may be helpful.

For those who have experience with Maple or Mathematica it would be interesting to see what those programs do with this and how fast.

Problem 4: FORM is not very good with composite denominators as in $1/(x+1)$. Usually it is best to emulate those yourself. This also gives you more control. Try to make a program that does the following integral:

$$\int_0^\infty dx \frac{1}{(x+1)(x+2)(x+3)(x+4)} \tag{1}$$