12 The compiler

The compiler consists of several files: compiler.c, token.c, comexpr.c, compcomm.c, comtool.c plus some pieces of code in files that address special statements. We should distinguish two different tasks:

- Translation of generic statements.
- Translation of algebraic expressions.

The second task is by far the most complicated.

12.1 Generic statements

Because at the algebraic level Form is a compiled language, the results of each compiled statement are stored inside a compiler buffer as a string of numbers that will be executed by the virtual machine that is the execution part of Form of which the most important part resides in the file proces.c. We will see that later. A compiler buffer is a struct, which is defined in the file structs.h by

```
typedef struct CbUf {
    WORD *Buffer;          /**< [D] Size in BufferSize */
    WORD *Top;             /**< pointer to the end of the Buffer memory */
    WORD *Pointer;         /**< pointer into the Buffer memory */
    WORD **lhs;            /**< [D] Size in maxlhs. list of pointers into Buffer. */
    WORD **rhs;            /**< [D] Size in maxrhs. list of pointers into Buffer. */
    LONG *CanCommu;        /**< points into rhs memory behind WORD* area. */
    LONG *NumTerms;        /**< points into rhs memory behind CanCommu area */
    WORD *numdum;          /**< points into rhs memory behind NumTerms */
    WORD *dimension;       /**< points into rhs memory behind numdum */
    COMPTREE *boomlijst;   /**< [D] Number elements in MaxTreeSize */
    LONG BufferSize;       /**< Number of allocated WORD's in Buffer */
    int numlhs;
    int numrhs;
    int maxlhs;
    int maxrhs;
    int mnumlhs;
    int mnumrhs;
    int numtree;
    int rootnum;
    int MaxTreeSize;
    PADPOINTER(1,9,0,0);
} CBUF;
```

All statements are stored in the 'Buffer'. We distinguish left hand sides and right hand sides. Each statement has one left hand side and potentially there can be as many right hand sides as the system will allow. These right hand sides are basically for the pieces of algebraic formula's. If the space in Buffer is not enough, it can be extended by doubling it up to a maximum size that is set in the setup. With a new allocation comes then also the problem to change all the corresponding pointers. This doubling is done in the routine DoubleCbuffer in the file comtool.c. Similarly, if the number of statements becomes too large the size of the lhs buffer can be doubled. This can be done when a lhs is added by means of the routine AddLHS. The routine AddRHS is a bit more complicated, because in a very lengthy expression with very many function arguments and/or subexpressions, the number may become too large. This was in particular the case in the 32-bits version of FORM. In that case a new compiler buffer is started. It is for this reason that the subterm SUBEXPRESSION has not only the number of its subexpression (rhs) in the compiler buffer, but also the number of the corresponding buffer. The compiler buffers sit in the array cbufList in the the struct Cc (of type C_const) which again belongs to the struct A. In the program it is referred to as cbuf which is defined as

```
#define cbuf ((CBUF *)(AC.cbufList.lijst))
```

in the file variable.h. Hence the main compiler buffer, which is the element cbufnum in the struct C can be obtained by:

```
CBUF *C = cbuf+AC.cbufnum;
```

Each statement is translated into a string of (U)WORDs in the Buffer and a pointer in the lhs array that points to this string of numbers. This string starts with a number that indicates the type of the statement. These types can be found in the file ftypes.h. The list starts with TYPEEXPRESSION, TYPEIDNEW, etc. The majority of statements do not have an algebraic expression and most of them are treated in the file compcomm.c. Those routines are called in the file compiler.c which has the tables of the various statements. There are two tables: com1commands with the statements of which the name can be abbreviated, and the table com2commands of which the name of the statement has to be (case insensitive) exactly as in the table. This second table has to be in alphabetic order because it is used with a binary search. The first table is processed in a semi-sequential way: the first character has to be in alphabetic order because there is a table that points to the first statement with that character, but after that it searches sequentially. The tables tell which routine should be called, what type of statement it is – for determining whether a statement obeys proper ordering – and some flags to tell whether the statement should be checked for matching parentheses, or whether autodeclare declarations may be applied. The data struct for these tables is identical to the one we saw already in the preprocessor.

Let us start with a simple statement like

```
    ,{"dropcoefficient",(TFUN)CoDropCoefficient,  STATEMENT,    PARTEST}
```

which resides in the table com2commands. The STATEMENT value indicates what type of statement we are talking about which is relevant for having statements in the proper order. The full list (in ftypes.h) is:

```
#define DECLARATION   1
#define SPECIFICATION 2
#define DEFINITION    3
#define STATEMENT     4
#define TOOUTPUT      5
#define ATENDOFMODULE 6
#define MIXED2        8
#define MIXED         9
```

The parameter PARTEST tells that the compiler should start with testing whether the parentheses match inside the statement.

The routine CoDropCoefficient resides in the file compcomm.c and is given by

```
int CoDropCoefficient(UBYTE *s)
{
    if ( *s == 0 ) {
        Add2Com(TYPEDROPCOEFFICIENT)
        return(0);
    }
    MesPrint("&Illegal argument in DropCoefficient statement: '%s'",s);
    return(1);
}
```

This is a statement without arguments and hence we add a lhs of only two WORDs to the Buffer. The second WORD is a length indicator as we are already used to in the subterms. Add2Com is a macro, defined in declare.h, that adds these WORDs. It uses the routine AddNtoL in comtool.c to check on the array sizes, extend them if necessary and then make the addition. The return value of the routine and all other compiler routines is 0 if everything is OK, positive if there is an error, but compilation may proceed to try to catch more errors, and negative if there is a fatal error that causes an immediate abort.

Let us have a look what this did with a simple program:

```
S a,b;
L F = (a+b)^4;
.sort
On Codes;
Dropcoefficient;
Print +s;
.end
```

The "On Codes" statement makes FORM print the content of its compiler buffer and in addition the numbers of all the variables that exist during the execution of the module. Let us skip those variables for now and just look at the LHS:

```
Left Hand Sides:
   70  2
```

Checking in the file ftypes.h we see the line

```
#define TYPEDROPCOEFFICIENT 70
```

and indeed this LHS is the DropCoefficient statement.

The above binary code would cause the following action in the algebra engine (in the routine Generator in proces.c):

```
    case TYPEDROPCOEFFICIENT:
        DropCoefficient(BHEAD term);
        break;
```

and the routine DropCoefficient in the file normal.c is

```
void DropCoefficient(PHEAD WORD *term)
{
    GETBIDENTITY
    WORD *t = term + *term;
    WORD n, na;
    n = t[-1]; na = ABS(n);
    t -= na;
    if ( n == 3 && t[0] == 1 && t[1] == 1 ) return;
    *AN.RepPoint = 1;
    t[0] = 1; t[1] = 1; t[2] = 3;
    *term -= (na-3);
}
```

You should ignore the GETBIDENTITY and the PHEAD in the declaration as they are meant for TForm and anyway superfluous here. We see that `t` gets pointed to just after the term. Then n is the size times sign of

the coefficient and na its full length. The coefficient is overwritten with the coefficient 1 and the length of the term is adjusted. You should ignore the RepPoint for now. It deals with repeat loops and tells that when the coefficient is changed, we have to stay in the loop. We have ignored here a possible floating point coefficient.

The above shows an example how one adds a simple statement to FORM. One starts with defining a TYPE for it in the file ftypes.h, then one makes an entry in the array com2commands in the file compiler.c. Next one makes a routine for the compilation which one can put in the file compcomm.c and of which the name starts preferably with Co. One has to mention this in the extern declarations in declare.h. Then one makes that the routine Generator can jump to the routine that does the actual action. And after constructing the routine that takes the specific action one should not forget to make the proper entry in the manual.

Of course the above is an example of a statement without arguments. Let us see how it goes with arguments. We select the ChainIn and ChainOut statements. In their case the two statements are nearly identical in their syntax and hence we define a single routine DoChain (in compcomm.c) that is called by the routines CoChainIn and CoChainOut with an extra argument corresponding to the actual TYPE.

```
int DoChain(UBYTE *s, int option)
{
    WORD numfunc,type;
    if ( *s == '$' ) {
        if ( ( type = GetName(AC.dollarnames,s+1,&numfunc,NOAUTO) ) == CDOLLAR )
            numfunc = -numfunc;
```

```
        else {
            MesPrint("&%s is undefined",s);
            numfunc = AddDollar(s+1,DOLINDEX,&one,1);
            return(1);
        }
tests:  s = SkipAName(s);
        if ( *s != 0 ) {
            MesPrint("&ChainIn/ChainOut should have a single function
                                          or $variable for its argument");
            return(1);
        }
    }
    else if ( ( type = GetName(AC.varnames,s,&numfunc,WITHAUTO) ) == CFUNCTION ) {
        numfunc += FUNCTION;
        goto tests;
    }
    else if ( type != -1 ) {
        if ( type != CDUBIOUS ) {
            NameConflict(type,s);
            type = MakeDubious(AC.varnames,s,&numfunc);
        }
```

```
            return(1);
        }
        else {
            MesPrint("&%s is not a function",s);
            numfunc = AddFunction(s,0,0,0,0,0,-1,-1) + FUNCTION;
            return(1);
        }
        Add3Com(option,numfunc);
        return(0);
    }
```

The argument is supposed to be a function, but there is a complication: The argument can be a dollar variable that should be evaluated during execution. Hence we will indicate the number of the dollar variable with a negative integer (numfun). If the argument is not a dollar variable we have to read its name (SkipAName) and after it its type of variable and its number (GetName). If the object is not a function we need an error message and because there is a confusing use of variables (is it a function or is it the other kind of variable) we set the type of this variable now to DUBIOUS and we return a nonfatal error, allowing the program to look for more compilation errors. If the argument had not been declared before, we make a declaration with AddFunction in an attempt to cut down on more identical error messages after returning a non fatal error. If everything is fine till here, we have to test whether there is indeed only a single argument. Once that test has been passed we may enter a statement in the left hand side buffer with the macro Add3Com which places 3 words in the lhs buffer,

eg.:

```
TYPECHAININ,3,numfun
```

In the routine Generator we need the code

```
case TYPECHAININ:
  AT.WorkPointer = term + *term;
  if ( ChainIn(BHEAD term,C->lhs[level][2]) ) goto GenCall;
  AT.WorkPointer = term + *term;
  break;
```

The pointer AT.WorkPointer is a pointer to a stack in which FORM stores mainly terms at each level of the expansion tree. This means that the first statement sets the pointer to the space after the term that is to be treated. In the case of TFORM BHEAD passes the pointer to the struct of the current worker, while for FORM it is an empty macro. The object `C->lhs[level][2]` passes the number of the function (or dollar variable). The variable level tells which statement we are processing and C is the current compiler buffer. Because eventually the new term is supposed to be written over the old term, the WorkPointer is set to the location after the potentially modified term. After this Generator will continue with the next statement. If anything goes wrong inside the ChainIn routine the "goto GenCall" will cause an abort and potentially a traceback.

The internals of the routine ChainIn can be looked up in the file function.c and should become clear at a later stage. The main thing to remember here is to take into account that it may have to operate more than once when there are noncommuting functions involved because the functions to be treated may not be adjacent.

And again an example in which we skip the listing of the numbers of the variables in the output with the exception of the number of the function f1:

```
S a,b;
CF f1;
L F = f1(a,b,a,b,b);
.sort
On Codes;
ChainOut,f1;
Print +s;
.sort
#$f = f1;
ChainIn,$f;
Print +s;
.end
```

which gives in the output for the first module:

```
 Commuting Functions
     ....
   euler_(120) mzvhalf_(121) agm_(122) gamma_(123) f1(150)
     ....
 Left Hand Sides:
```

```
   63  3  150
```

and for the second module (with the dollar variable):

```
 Dollar variables
   $$(0) $f(1)
 Left Hand Sides:
   62  3  4294967295
```

Let us ignore the `$$(0)` for now. In this dump the program prints unsigned WORD's. Hence the negative value becomes basically $2^{32} - 1$. TYPECHAININ and TYPECHAINOUT are 62 and 63 respectively.

The last example of 'simple' statements we will look at is the Print statement with a format string. Here the question is how to pass the format string. We are not going to look here at the complete DoPrint routine, because it has to handle a number of cases. The important part here is the AddComString routine in compcomm.c which analyses the string and writes, after taking escape characters into account, the characters as WORDs into the compiler buffer.

Our test program is:

```
S a,b;
CF f1;
L F = f1(a,b,a,b,b);
.sort
On codes;
```

```
Print "<1> %t";
.end
```

and the output:

```
 Left Hand Sides:
   37  7  4294967295  0  2  540946748  29733
<1>  + f1(a,b,a,b,b)
```

The third number is actually -1 and it indicates STDOUT. the number two gives the length of the string in WORDs. Because there are 6 characters, they fit in two WORDs (we use WORD = 4 bytes). The zero is irrelevant for this type of print statement. We have to realise that the compilation routine does both types (with and without format string) of print statements.

When making ones own statements one is of course free to use the compiler buffer in any way suitable, subject to a few rules:

- The first WORD of a statement is its TYPE.
- The second WORD of a statement is the length in WORDs of the binary code of the statement.
- The size of the statement should not be excessively long, although the word excessive is still open to interpretation, and may also change over time.
- Try to use the AddNtoL routine in comtool.c, either directly or by means of one of the macro's Add2Com, Add3Com, Add4Com, Add5Com. This makes sure that, if a buffer needs to be extended, it is done in the

right way.

12.2 Algebraic expressions

The algebraic expressions are by far the most complicated part of the compiler. Of course nowadays there are many tools that can help in the setup of syntax etc. Much of that was not quite suitable for the task of computer algebra when FORM was designed. Hence all that is needed has been programmed from scratch. For this subsection 'the compiler' is considered the part of the compiler that translates the algebraic expressions into binary code for the virtual FORM machine.

The compiler consists of three parts:

1. The part that reads the code, looks up the variables, checks the parentheses etc. and translates it into an intermediate code. This is called the tokenizer.
2. At the level of tokens some more error checking takes place, like for instance w.r.t. the proper use of wildcards. Then there are some optimizations.
3. Finally the tokens are translated into binary code. This also does some optimizations, in particular with respect to identical subexpressions.

In the file comexpr.c one finds the routines that read the statements that contain algebraic expressions. This includes the expressions in the LHS of an id statement. Hence the complete reading of the id statements is also in this file. The routines here prepare the input for being processed by the tokenizer. All the token routines are in the file token.c. These routines need to be adapted only when fundamental changes are made, or when new types of variables are introduced as was the case with the addition of a floating point facility.

The token buffers are buffers of objects of the type SBYTE. This means they are one byte long. The type of the tokens is given as negative bytes and their names start with the character T. These types can be found in the file ftypes.h. The characters inside each token are positive. This means that if there is a number greater than 127, it has to be split into more than one byte. In some cases some numbers are base 100 to make translations easier. When the token type is TNUMBER it is base 100, while when the token type is TNUMBER1 it is base 128.

Functions will always be translated as the token for its number, followed by a token for the opening parenthesis TFUNOPEN, and eventually a token for the closing parenthesis TFUNCLOSE. With for instance $f(x+1)$ this may lead to an accumulation of parentheses because it will become effectively $f((x+1))$. Such extra parentheses are removed in one of the optimization routines, in this case simp2token.

```
S a,b;
CF f1,f2;
On Tokens;
On codes;
L F = f1(a,b,a,b,b)+f2+(a+b)^4;
.end
To tokenize: f1(a,b,a,b,b)+f2+(a+b)^4
F 1 2 (( ( S 20 ) , ( S 21 ) , ( S 20 ) , ( S 21 ) , ( S 21 ) )) + F 1 3
+ ( S 20 + S 21 ) ^ # 4
  The contents of the token buffer are:
```

```
F 1 2 (( S 20 , S 21 , S 20 , S 21 , S 21 )) + F 1 3 + ( S 20 + S 21 ) ^
# 4
S 20 + S 21
F 1 2 (( S 20 , S 21 , S 20 , S 21 , S 21 )) + F 1 3 + sub 0 ^ # 4
 Symbols
   i_#i(0) pi_(1) coeff_(2) num_(3) den_(4) xarg_(5) dimension_(6) factor_(7)
   sep_(8) a(20) b(21)
 Commuting Functions
     ....
   euler_(120) mzvhalf_(121) agm_(122) gamma_(123) f1(150) f2(151)
```

The codes like TFUNCTION, which is -4, are translated back in this output; in this to F and TSYMBOL (-1) becomes S. The `F 1 2` is a bit harder to understand. The number of the variable is written modulus 128 and hence effectively it says function 130. But functions also have an offset of 20 (FUNCTION) which is later added in the code generator. Hence `F 1 2` indicates indeed f1. The first two lines of tokens are the raw translation. One can see that each function argument obtains an extra pair of parentheses and the whole of the function has a matching pair of double parentheses. In the next two lines of tokens we see the result of some optimization in which the parentheses are stripped when possible. Next we see a line with `S 20 + S 21`. Here the optimizer has defined a subexpression which we see in the next line as `sub 0`. At this point there is no attempt yet to look for identical subexpressions. That will be done in the code generator which stores these subexpressions in a balanced tree with some indicators that make it easier to find them. hence for the expression

```
L F = f1(a,b,a,b,a+b)+f2+(a+b)^4+(a+b)^3;
```

the final result would be

```
S 20 + S 21
S 20 + S 21
S 20 + S 21
F 1 2 (( S 20 , S 21 , S 20 , S 21 , sub 0 )) + F 1 3 + sub 1 ^ # 4 + sub
2 ^ # 3
```

Also left hand sides of id statements have to pass the tokenizer. Assume we add an extra module:

```
    id  a*b = a+2*b;
To tokenize: a*b
S 20 * S 21
  The contents of the token buffer are:
S 20 * S 21
S 20 * S 21
To tokenize: a+2*b
S 20 + # 2 * S 21
  The contents of the token buffer are:
S 20 + # 2 * S 21
S 20 + # 2 * S 21
```

The repetition of the same lines is due to the printing after different stages of the optimizer.

Changes in the token.c file are only necessary when new objects are introduced that require special action during compilation. This was for instance the case with the floating point function FLOATFUN which required the type TFLOAT. Floats will be treated later. For now the only thing relevant is that if floating point capacity is activated, FORM can read floating point numbers as in:

```
    #StartFloat 90
    S   a,b,c;
    L   F= 0.12*a+34.56E+12*b+3.7/0.9*c;
    Print;
    .end
   F =
      4.11111111111111111111111111111111111111e*c + 3.456e+13*b + 1.2e-01*a;
```

12.3 The code generator

The routines that translate the tokens into code for the virtual processor reside in compiler.c, comexpr.c and comtool.c. The main problem here is the treatment of subexpressions. These can be parts of an expression that are between parentheses or general function arguments. They are treated recursively in such a way that the deepest subexpressions are translated first. Simultaneously FORM checks whether there exists already an identical subexpression in the compiler buffer. This avoids getting a pileup of subexpressions in the compiler buffer when for instance $x + 1$ is a very popular function argument. This is useful only when the contents of the (input) compiler buffer will not be changed during execution. Also TFORM would have problems with such changes. Hence one should make copies of selected parts during execution, when for instance making wildcard substitutions.

Also tables have right hand side expressions, but those should survive .sort instructions. Hence each table has its own compiler buffer. If the table has a very large number of elements it may be that more than one compiler buffer is used.

Of course searching through the buffer(s) for identical subexpressions needs to be done fast. Preferably via binary search. The compiler buffer struct is an object of type CBUF, defined in the file structs.h and the tree is an object of type COMPTREE by the name boomlijst (boom=tree and lijst=list), defined in the same file. The tree is kept balanced in the routine InsTree for inserting elements. FindTree does the binary search. Of course making an insertion in the tree would need locks if it were to be done in parallel. To avoid this problem the compiler will run only in a single core. This holds also for the addition of table elements, which have to

be handled by the compiler. One could potentially make this more dynamic by doing this at runtime, but that would require locks for each individual table. Currently such locks have not been included. It would be rather complicated to set this up properly, specially when two workers want to set the same table element to different expressions.

Wildcard information is stored in the subterms of type SUBEXPRESSION after the relevant information of its number inside the rhs of the compiler buffer, its power and the number of the compiler buffer in which the rhs is stored. The wildcard information comes in pieces of information that are each 4 WORDs long. Hence one can derive the number of them by subtracting SUBEXPSIZE from the size of the subterm and dividing by 4. The precise format will be treated in the chapter on wildcarding. Important is that this wildcard information is passed on to all deeper subexpressions, because FORM does not keep track of which ones are actually needed. That would be rather complicated.

```
    id  f(a?,b?,c?) = (a+(b+c)^2)^2;
```

In the inner subexpression the value of a is not really needed but it will be in the subexpression `(b+c)` anyway. It is not a big overhead.

Let us have a look at some examples first. We can take one of the previous examples:

```
S a,b;
CF f1,f2;
On Tokens;
On codes;
```

```
L F = f1(a,b,a,b,a+b)+f2+(a+b)^4+(a+b)^3;
id  a*b = a+2*b;
.end
S 20 + S 21
S 20 + S 21
S 20 + S 21
F 1 2 (( S 20 , S 21 , S 20 , S 21 , sub 0 )) + F 1 3 + sub 1 ^ # 4 + sub
2 ^ # 3
    id  a*b = a+2*b;
To tokenize: a*b
S 20 * S 21
To tokenize: a+2*b
S 20 + # 2 * S 21
```

Here I have edited the token outputs as they go to the code generator. The code generation results in

```
 Left Hand Sides:
   1  19  1  4294967295  0  0  6  5  3  1  0  7  1  6  20  1  21  1  0
 Right Hand Sides:
   8  1  4  20  1  1  1  3  8  1  4  21  1  1  1  3

   26  150  22  1  4294967295  20  4294967295  21  4294967295  20  4294967295
```

```
  21  11  1  9  6  5  1  1  0  1  1  3  1  1  3  7  151  3  0  1  1  3  9
6  5  1  4  0  1  1  3  9  6  5  1  3  0  1  1  3

8  1  4  20  1  1  1  3  8  1  4  21  1  2  1  3
```

The first RHS is the `a+b`, but this time only once. Remember that 4294967295 is actually -1 and for a function argument it means -SYMBOL, or a fast argument, and it is followed by the number of the symbol. The last argument of the function f1 (150) is

```
11  1  9  6  5  1  1  0  1  1  3
```

This is a general argument and it has only one term, which is a subexpression (6). It is subexpression 1 in buffer zero and it has one power. This subexpression occurs in two more terms, but now with the powers 4 and 3. The third RHS is the `a+2*b` of the id statement. The LHS of the id statement is on top. The code 1 means that it is an id statement and of the parameters `1 -1 0 0` the 1 means the suboption of the id statement, the -1 tells there is no label (as with ifmatch->label) because if there would be one it would be the number of the label and the two zeroes are for special options involving dollars or the ALL option. Next follows the subexpression 'prototype'. It points to the subexpression that is the right hand side. But it would also contain all wildcard information that has to be passed on to all subexpressions in this RHS. We will see this below. After this subexpression prototype follows the actual LHS without coefficient. In this case it has a length of 7 WORDs and stands for `a*b`.

Let us now take one example with a wildcard.

```
S a,b,n;
On codes;
L F = a+a^2*b+a*b^2;
id  a*b^n? = b+(a+b)^2+(a+2*b*(a+b))^3;
.end
```

We have intentionally created subexpressions in the RHS to see how this wildcard information is passed on. The code is

```
 Left Hand Sides:
   1  23  2  4294967295  0  0  6  9  4  1  0  2  4  22  22  7  1  6  20  1  21
     1000000022  0
 Right Hand Sides:
   8  1  4  20  1  1  1  3  12  1  4  20  2  1  4  21  1  1  1  3  12  1  4
   20  1  1  4  21  2  1  1  3

   8  1  4  20  1  1  1  3  8  1  4  21  1  1  1  3

   8  1  4  20  1  1  1  3  17  1  4  21  1  6  9  2  1  0  2  4  22  22  2  1
     3

   8  1  4  21  1  1  1  3  13  6  9  2  2  0  2  4  22  22  1  1  3  13  6  9
```

```
    3   3   0   2   4   22   22   1   1   3
```

We see two new things in the LHS. The subexpression protoype is now

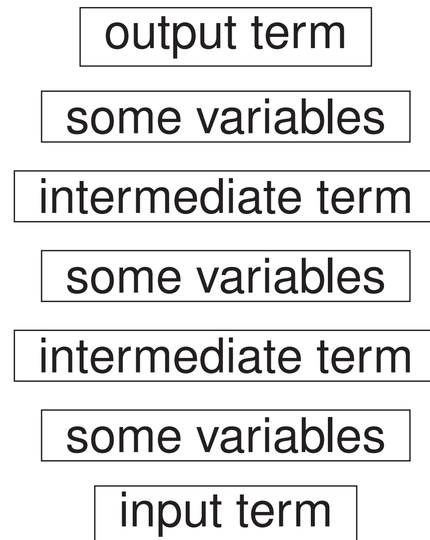
```
  6 9   4   1   0   2   4   22   22
```

and the power of b(21) is indicated by 1000000022 instead of by 22(=n). The last one is easiest to explain: to indicate that it is a wildcard we have added an offset of `2*MAXPOWER`. The first has an extra 4 WORDs in the prototype: The 2 means a wildcard that is in principle a symbol to be replaced by a symbol (SYMTOSYM as defined in ftypes.h). The symbol is 22(n) and for now we echo that in the next word as well. This will be copied during execution and then the second 22 will be replaced by its new value, which could also be SYMTONUM(1) or SYMTOSUB(3) in which case in the copy the first WORD will also be replaced. If one studies now the RHS of the id statement one can see that all subexpressions now have these 4 WORDs attached as well, and one of the challenges during execution is to make sure that the subexpressions (and their subexpressions) obtain the proper wildcard values.

Some more aspects of the compiler buffers will be treated in the parts of the code that make use of them during the execution of programs.

13 The workspace(s)

The workspace is a heap that is used to store intermediate results during the execution of the statements inside a module. It can also store other intermediate pieces of information. There are actually four workspaces: one for (U)WORDs, one for pointers, one for LONGs and one for file positions. The workspaces for pointers, LONGs and file positions are dynamic, meaning that they can be extended if possible, because they are referred to by offsets. The main workspace however is allocated during the startup of the program and cannot be modified afterwards, because there are many pointers pointing to objects inside this heap.



The variables for each of the workspaces are:

- The main Workspace:
  AM.WorkSize: All workers have the same size for their workspace. Hence a single global variable will do the job.
  AT.WorkSpace: The address of the buffer as allocated for each worker separately.
  AT.WorkTop: The top of the buffer. Used for checking whether there is a fatal buffer overflow.
  AT.WorkPointer: The pointer of where we are inside the buffer.

- The lWorkspace for LONGs:
  AR.lWorkSize: The current size of the buffer. Because this buffer is dynamic, it is a variable that may be different for each worker.
  AT.lWorkSpace: The address of the buffer as allocated for each worker separately.
  AT.lWorkPointer: The pointer of where we are inside the buffer.

- The pWorkspace for pointers:
  AR.pWorkSize: The current size of the buffer. Because this buffer is dynamic, it is a variable that may be different for each worker.
  AT.pWorkSpace: The address of the buffer as allocated for each worker separately.
  AT.pWorkPointer: The pointer of where we are inside the buffer.

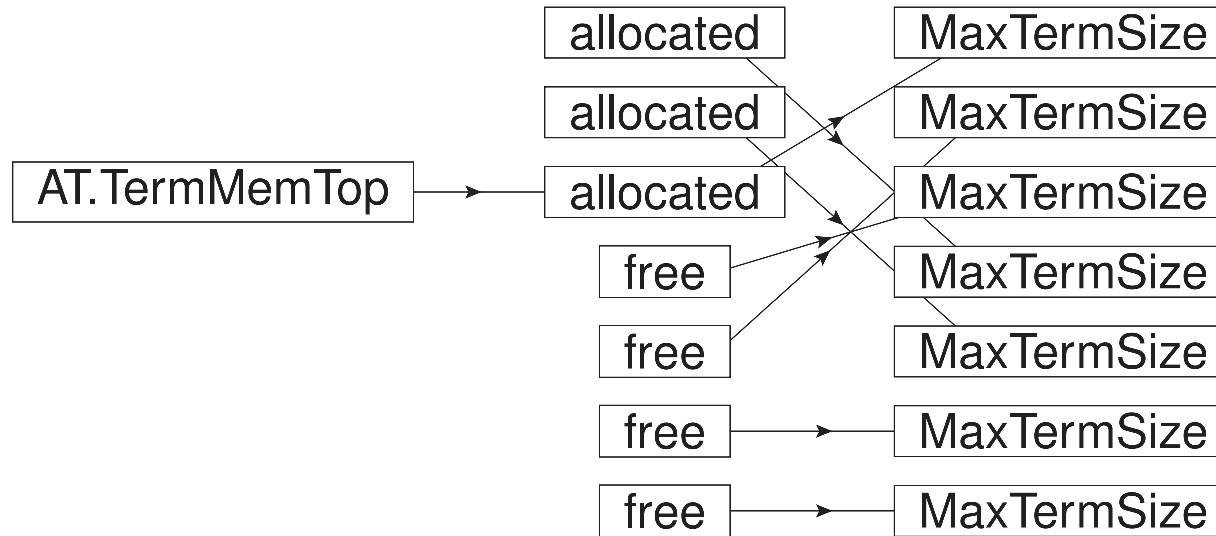
- The posWorkspace for file positions:
  AR.posWorkSize: The current size of the buffer. Because this buffer is dynamic, it is a variable that may

be different for each worker.
AT.posWorkSpace: The address of the buffer as allocated for each worker separately.
AT.posWorkPointer: The pointer of where we are inside the buffer.

The reason file positions are treated separately is because in the original 32 bits version a LONG was not enough to indicate a position in a file. In the 64 bits version this restriction is not present, but there is no reason to take it out.

Originally there was only a single workspace and bigger objects were squeezed in there in rather unelegant ways. Eventually this was cleaned up with the various dynamic spaces, but changing the main workspace in such a way that it would be dynamic would be a rather big (there are more than 1300 references to AT.WorkPointer) and error prone job. It also would have the rather nasty property that if FORM gets into an infinite loop, it may be quite a while before the computer runs out of memory space and the crash may either not give an error message, or give a confusing error message.

In addition to the above workspaces FORM needs occasionally some spaces that would ordinarily be obtained by a call to malloc and returned by a call to free. Because those would be calls with each time the same size, and because not too many of those spaces are needed, there are some arrays with the addresses of such buffers and FORM does not return these spaces to the system.



They are mainly run by macro's and they are

- TermMalloc and TermFree: For having a temporary buffer of MaxTermSize.
- NumberMalloc and NumberFree: For having a temporary buffer of a size that corresponds to the maximum number size.
- CacheNumberMalloc and CacheNumberFree

The CacheNumberMalloc is currently not used. There are also subroutine versions of these calls, but nowadays the macro's are preferred. The macro's cq subroutines need an extra argument which should be a text string which is used by the subroutine versions for the purpose of error messages. This can be helpful in finding which routine is responsible for the problematic call.

The above way of allocating was also introduced in parts of the C++ code for a number of the objects there. This made the program significantly faster. We hit here on one of the potential problems with lazy objective oriented programming: a continuous mallocing and freeing of objects.

14 Normalization

Before we go to the virtual machine, also called the algebra processor, we need to have a look at the normalization routines. These are all in the file normal.c. The folds/routines in this file are

```
## Includes : normal.c       37
#[ Normalize :
    ## CompareFunctions :    51
    ## Commute :            105
    ## Normalize :          184
    ## ExtraSymbol :       4213
    ## DoTheta :           4275
    ## DoDelta :           4372
    ## DoRevert :          4439
#] Normalize :
## DetCommu :              4515
## DoesCommu :             4583
## PolyNormPoly :          4609
## EvaluateGcd :           4646
## TreatPolyRatFun :       4971
## DropCoefficient :       5113
## DropSymbols :           5131
```

```
## SymbolNormalize :       5154
## TestFunFlag :           5254
## BracketNormalize :      5288
                           5444
```

After the colon character I have put the line number at which each fold starts to indicate how long the routine inside is. These numbers are not inside the actual file. As one can see, most routines are relatively short supporting routines, while the actual routine is of course Normalize. It is very important that this routine is fast, because it is called after each statement that changed a term, and sometimes also when it is not clear whether a term was modified. This would for instance be when a term is read from a file.

We distinguish different types of actions.

- Sorting and combining subterms. An example of combining would be when there are several subterms of the type SYMBOL.
- Substituting functions that, by design, can be evaluated into simple subterms or numbers.
- Dealing with the replace_ function.
- Combining numbers into a single coefficient.

The normalized term will be collected in the workspace.

Because the Normalize routine has currently almost 4000 lines of code, it is subdivided into various folds, each dealing with the various phases of the normalization. We show them here

```
WORD Normalize(PHEAD WORD *term)
{
/*
    ## Declarations :
    ## Setup :
    ## First scan :
    ## Easy denominators :
    ## Index Contractions :
    ## NonCommuting Functions :
    ## Commuting Functions :
    ## Track Replace_ :
    ## LeviCivita tensors :
    ## Delta :
    ## Loose Vectors/Indices :
    ## Vectors :
    ## Dotproducts :
    ## Symbols :
    ## float_ :
    ## Do Replace_ :
    ## Errors and Finish :
*/
```

```
}
```

The first problem is the allocation of temporary buffers.

```
   #[ Declarations :
*/
GETBIDENTITY
WORD *t, *m, *r, i, j, k, l, nsym, *ss, *tt, *u;
WORD shortnum, stype;
WORD *stop, *to = 0, *from = 0;
/*
The next variables could be better off in the AT.WorkSpace (?)
Now they make stackallocations rather bothersome.
*/
WORD psym[7*NORMSIZE],*ppsym;
WORD pvec[NORMSIZE],*ppvec,nvec;
WORD pdot[3*NORMSIZE],*ppdot,ndot;
WORD pdel[2*NORMSIZE],*ppdel,ndel;
WORD pind[NORMSIZE],nind;
WORD *peps[NORMSIZE/3],neps;
WORD *pden[NORMSIZE/3],nden;
WORD *pcom[NORMSIZE],ncom;
```

```
WORD *pnco[NORMSIZE],nnco;
WORD *pcon[2*NORMSIZE],ncon; /* Pointer to contractable indices */
WORD *n_coef, ncoef; /* Accumulator for the coefficient */
WORD *n_llnum, *lnum, nnum;
WORD *termout, oldtoprhs = 0, subtype;
WORD ReplaceType, ReplaceVeto = 0, didcontr, regval = 0;
WORD *ReplaceSub;
WORD *fillsetexp;
CBUF *C = cbuf+AT.ebufnum;
WORD *ANsc = 0, *ANsm = 0, *ANsr = 0, PolyFunMode;
#ifdef WITHFLOAT
WORD withfloat = 0;
WORD *firstfloat = 0;
#endif
LONG oldcpointer = 0, x;
n_coef = TermMalloc("NormCoef");
n_llnum = TermMalloc("n_llnum");
lnum = n_llnum+1;
/*
int termflag;
*/
```

```
/*
   #] Declarations :
```

There are quite a few arrays with a fixed size that is determined by the macro NORMSIZE which has currently the value 1000, defined in the file fsizes.h. This is a potential weakness, but making these arrays dynamic will add overhead in a routine that is very timing critical. In the past NORMSIZE had a much smaller value and this has caused once a nasty error that was not completely trivial to debug. In the past the stack allocations were relatively small and hence these arrays could cause problems with it. Nowadays the stacks are more flexible. Still a proper dynamic handling in which all would be sitting inside a data struct that can be reallocated would be preferable. It is something that might be considered in the future.

We will come to the meaning of the various arrays below, but first the use of a compiler buffer needs attention. During normalization it can happen that a single object gets replaced by a subexpression with more than one term by for instance a replace_ function. The substitution of subexpressions is done in the file proces.c and not in the Normalize routine. Hence we have a special compiler buffer (one for each worker) that is reserved for such a subexpression and we insert in the current term a subterm of type SUBEXPRESSION with the proper compiler buffer number. This means that after the term has been normalized, it has to go back to the expansion of subexpressions and after that come back to the normalization routine to finish the proces. Because this can also happen inside the arguments of functions it is not entirely trivial.

Let us follow what happens with a rather simple term that has only symbols and a single numerical coefficient. Let us assume that the unnormalized term looks like

```
    22 SYMBOL 6 #a 2 #b 1 SYMBOL 4 #c 5 SYMBOL 8 #a 2 #b -1 #c 3 2 7 3
```

This should of course normalize to `2/7*a^4*c^8`. In the setup fold we prepare a few things and return immediately if the term happens to be zero. The first scan fold is the major part of the code, because it is one enormous switch, recognizing all different types of subterms and many special functions. It has been programmed in such a way that in principle a good compiler can make these cases into a giant jump table, although I am not sure whether gcc actually does this.

```
        if ( t < m ) do {  /* m points at the start of the coefficient */
        r = t + t[1];
        switch ( *t ) {
            case SYMBOL :
                 .
                 .
                 .
        }
        t = r;
TryAgain:;
    } while ( t < m );
```

We have arrays to store symbols (psym), vectors (pvec), dotproducts (pdot), Kronecker delta's (pdel) and indices (pind). The pp varieties of these are pointers in these arrays (with the exception of pind) and the n variety tells how many elements there are in each array. There are also arrays of pointers to various types of functions: pcom for commuting functions, pnco for non-commuting functions, peps for Levi Civita tensors, pden for denominator

functions and finally there is the array for indices that should be contracted. The coefficient is accumulated in n_coef. The big loop in the First scan fold treats the subterms one by one. Because the first case is SYMBOL we do not have to dig very deep in our example. There are however special cases that we have to check for like:

Special symbol	action
coeff_	replace by coefficient
num_	replace by numerator
den_	replace by denominator
dimension_	replace by current dimension
wildcard to short number	replace by the short number
roots of unity	may have to adjust power

If any of these evaluate into a number it can be inserted into the coefficient. In the case of dimension_ or a root of unity it may be replaced by another symbol or the same symbol to a different power. If a symbol remains it can be inserted in the psym buffer. It looks whether there is a symbol with the same name already and if there is the powers are added. If the new power is zero, the symbol is taken from the array. Looking for a symbol with the same number is done starting at the top and then bubbling down. The reason to start at the top is that most of the time the symbols are already in order due to a previous normalization and hence a single compare suffices. And also a bubble type sort is not very slow if there are relatively few symbols which is statistically very likely. Hence a binary search was decided against.

After all subterms have been through the first scan, we can start composing the output term. In this the symbols come (almost) last. The psym array is now copied to the output term, but we do have to check for

power restrictions first. Also, when we have calculus modulus some number, the power may have to be adjusted, subject to the options we selected for the modulus calculus. The term in the example will go through the phases:

```
term  22 SYMBOL 6 #a 2 #b 1 SYMBOL 4 #c 5 SYMBOL 8 #a 2 #b -1 #c 3 2 7 3
 psym  #a 2
 psym  #a 2 #b 1
 psym  #a 2 #b 1 #c 5
 psym  #a 4 #b 1 #c 5
 psym  #a 4 #c 5
 psym  #a 4 #c 8
out   10 SYMBOL 6 #a 4 #c 8 2 7 3
```

Let us take a different example with some special functions:

```
14 MOEBIUS 5 0 -SNUMBER 102 INVERSEFACTORIAL 5 0 -SNUMBER 6 2 1 3
```

Here we have functions that can be evaluated during normalization. The routine first runs into the Moebius function. The code for this is

```
    case MOEBIUS:
/*
        Numerical function giving -1,0,1 or no evaluation
*/
        if ( t[1] == FUNHEAD+2 && t[FUNHEAD] == -SNUMBER
            && t[FUNHEAD+1] > 0 ) {
            WORD val = Moebius(BHEAD t[FUNHEAD+1]);
            if ( val == 0 ) goto NormZero;
            if ( val < 0 ) ncoef = -ncoef;
        }
        else {
            pcom[ncom++] = t;
        }
        break;
```

If the argument is a single short positive number we can evaluate the Moebius function. If not it is left as is and put in the array of pointers to commuting functions. That array is sorted at a later stage before the output

is written. The Moebius function is evaluated in the file reken.c and gives for its answer either -1, 0 or +1. If the answer is zero the whole term becones zero and we can abort rather quickly. If the number of unique prime factors was odd (as is the case here) we have to change the sign of the coefficient. The running value of the coefficient is kept in the array n_coef with the length ncoef.

The next time in the loop we run into the inverse factorial with the code:

```
    case INVERSEFACTORIAL:
        if ( t[FUNHEAD] == -SNUMBER && t[FUNHEAD+1] >= 0 ) {
            if ( Factorial(BHEAD t[FUNHEAD+1],(UWORD *)lnum,&nnum) )
                goto FromNorm;
            ncoef = REDLENG(ncoef);
            if ( Divvy(BHEAD (UWORD *)n_coef,&ncoef,(UWORD *)lnum,nnum)
            ncoef = INCLENG(ncoef);
        }
        else {
nospec:     pcom[ncom++] = t;
        }
        break;
```

Here we check that we can evaluate the factorial and if so, the Factorial routine in the file reken.c takes care of it. Note that this time the answer can be rather lengthy. It will be returned in the array lnum with the length nnum. The final step is to divide the coefficient by this factorial. The routine Divvy (also in reken.c) divides a rational by a long integer. The macro's REDLENG and INCLENG convert between two possible definitions of the length of a rational number. Normally it is the full length including the length parameter, but sometimes we need the reduced length, which is the length of each of the numerator or the denominator. In both cases however we include the sign in this length parameter.

The answer after normalization would be

```
4 1 360 -3
```

We keep two arrays with addresses of functions for the simple reason that non-commuting functions should be treated differently from commuting functions. There is the routine Commu that has been included to determine whether non-commuting functions do commute after all. This is for instance the case with gamma_ functions with different spin lines. Currently not much more use is made of this routine. Originally it was to be used for commands in which the user can specify during the declarations whether certain objects commute. This was however never implemented. Sorting the commuting functions is not entirely trivial. It is done when the output term is composed and a bubble sort is used. Again we will not discuss the floating point numbers here.

There are some extra complications when we use a polyratfun. For this function we need the additional routines for rational polynomials and we have to make sure that we do not call the GCD routine for such polynomials more than necessary, because that is a very expensive operation. This requires a very careful manipulation of the proper dirty flag in the header of the function. We should see this problem back in the file proces.c.

Potential replace_ operations take place after most of the normalization. If this operation is nontrivial the routine will return the value 1, indicating that the term has to go through the search for unsubstituted subexpressions again.

15 The algebra engine

The algebra engine, also called the algebra processor or the virtual machine, is of course the core part of FORM. Much effort has gone into making it fast. Some parts may not be very transparent. This has several causes:

- Speed optimization. This is the justification for the existence of FORM.
- Code developed over many years, becoming ever bigger and more complicated.
- Evolution in programming styles.

In the late 1990's the whole frontend of FORM was reprogrammed, but this was not the case with the algebra engine. It could probably use a similar overhaul, but this should not go at the cost of execution speed. But some clever use of more subroutines and some macro's could probably improve things. It would still be a big task though and it might take some time to get it sufficiently debugged.

The core part of the algebra engine resides in the file proces.c. This file contains the following routines:

```
## Includes : proces.c
#[ Processor :
    ## Processor :    WORD Processor()
    ## TestSub :      WORD TestSub(term,level)
    ## InFunction :   WORD InFunction(term,termout)
    ## InsertTerm :   WORD InsertTerm(term,replac,extractbuff,position,termout)
    ## PasteFile :    WORD PasteFile(num,acc,pos,accf,renum,freeze,nexpr)
    ## PasteTerm :    WORD PasteTerm(number,accum,position,times,divby)
    ## FiniTerm :     WORD FiniTerm(term,accum,termout,number)
    ## Generator :    WORD Generator(BHEAD term,level)
    ## DoOnePow :     WORD DoOnePow(term,power,nexp,accum,aa,level,freeze)
    ## Deferred :     WORD Deferred(term,level)
    ## PrepPoly :     WORD PrepPoly(term,par)
    ## PolyFunMul :   WORD PolyFunMul(term)
#] Processor :
```

There are however many routines in other files that execute specific commands.

Let us have a look at what the routines in proces.c do in order of being used:

Processor This routine deals with expressions. It locates the expressions that have to be processed. For each it starts a new sort by calling the NewSort routine after which it starts processing the expression. It reads

the terms one by one, makes sure they are properly normalized and then calls the Generator routine to apply the statements of the current module to it. After all terms have been processed it calls EndSort to do the final sorting and write the expression to the output scratch file. There is a special version of this routine in the file threads.c for the parallel processing in TForm.

Generator This routine is the core of the tree expansion of statements in a module. It is probably the most important routine in Form. The variable level indicates which statement is being treated in the current call and for the next level it has to call itself recursively until it has reached the last statement, after which the resulting term is written to the sort system by means of the StoreTerm routine. The recursive call takes only place when the current statement changes the term. After the change first the Normalize routine is called to make sure that when Generator goes to the next statement or writes the term to StoreTerm the term is always properly normalized. The order of action inside Generator is:

1. Check whether there is an unsubstituted (sub)expression by calling the TestSub routine. This routine indicates that there is such a (sub)expression (or a dollar expression), or whether there is a function that should be expanded into potentially more than a single term,
2. If there is no such action to be taken, the term is normalized and the level is raised which means that we go to the next statement. If there is no such statement, the term is written to the sort system and Generator returns.
3. If the level is raised, Form checks what type of statement has to be applied. First it checks whether it is one of the special statements that need individual treatment. If so the corresponding action is taken, either by a call to a corresponding routine, or by direct action on the spot. This is all in one very large

case switch. This switch is in the (sub)fold "Special action". If it is not one of these special statements it checks whether it is a substitution statement and whether the pattern matcher has taken out a pattern and replaced by a subexpression subterm. This is done by a call to the routine TestMatch which resides in the file pattern.c.

4. If the pattern matcher did indeed generate one or more subexpression subterms the routine TestSub is called. If this indicates a function that can generate more than one term the appropriate routine is called. It could also be that a subexpression is located inside a function argument. This causes a call to the subroutine InFunction.
5. If a dollar expression was encountered it should be expanded by a call to DolToTerms in the file dollar.c.
6. Finally subterms of the type subexpression are expanded. There are various cases here, depending on whether there is just a single power or whether in the case of multiple powers it is permitted to use binomial expansions.
7. Expressions may have to be copied from disk. Hence they are treated separately. Here we have to distinguish between active expressions and stored expressions.
8. If the statement did not 'bite', the level is raised and Generator goes back to point 3.

The way subexpressions are treated may not be ideal when the powers are very large. The way it is done it to create a heap of subterms in the workspace. This heap starts with the part of the term before the subexpression. After that the terms of the power of the subexpression are added in a 'stacked' way in the workspace. This is done in the routine PasteFile. When the term is complete the routine FiniTerm creates

the proper output term which then is fed to Generator for further treatment. The space where the term is built up is called the accumulator and it goes by the name accum. It is show a bit better below.

TestSub This routine checks whether there is unfinished work inside a term. Unfinished work may mean that there are still subexpressions, expressions or dollar expressions in the term. Those would have to be expanded. The expansion will take place in the Generator routine that calls TestSub. It also checks whether functions have arguments that still have their dirty flag set. And in addition it checks whether there are functions that can be expanded. An example of this would be a sum_ function that has arguments such that it should be expanded. Some expansions are done already inside TestSub. When function arguments need to be expanded very special care should be given to the positioning of the information in the workspace. This expansion calls Generator which in turn calls the routine InFunction and because we can have functions inside functions the recursion makes it very tricky to not overwrite parts of the original term.

DoOnePow The system we described in the Generator routine for inserting powers of subexpressions needs a bit more care when we have powers of expressions that may be on disk. In that case we need a very careful caching system to avoid having to reposition the disk for each term. The DoOnePow routine, called by Generator, takes care of this.

Deferred When we have brackets and we use the "Keep Brackets" statement, the contents of the brackets are stripped when the terms are read from the input. This is done automatically in the routine GetTerm (in the file store.c). This means that GetTerm reads a term and if it encounters a new 'outside' of brackets it sends this outside back as the new term, but if this outside is the same as the outside of the previous term we skip this term. We do however remember where this new bracket was, because the routine Deferred will

multiply all terms that are ready to go to StoreTerm by the contents of this bracket. Deferred is called from Generator and there are a few complications concerning dummy and contracted indices.

PrepPoly Checks whether the numerical coefficient should be pulled inside the polyfun (if there is one). If so, it does this.

PolyFunMul Multiplies the contents of more than a single polyfun into a single polyfun.

Of course the above is a rather rough description of what happens inside the algebra engine. There are many special cases. Unfortunately we cannot deal with all of them.

Let us have a better look at the accumulator. We do this with the following simple substitution:

```
Symbols y,a,b,x;
L = y^7
id y = a*x+b*x;
```

Because we can use binomials this gives in the accumulator after each pass:

```
a*x,a*x,a*x,a*x,a*x,a*x,a*x 1/1
a*x,a*x,a*x,a*x,a*x,a*x,b*x 7/1
a*x,a*x,a*x,a*x,a*x,b*x,b*x 7/1*6/2 = 21/1
a*x,a*x,a*x,a*x,b*x,b*x,b*x 21/1*5/3 = 35/1
a*x,a*x,a*x,b*x,b*x,b*x,b*x 35/1*4/4 = 35/1
a*x,a*x,b*x,b*x,b*x,b*x,b*x 35/1*3/5 = 21/1
a*x,b*x,b*x,b*x,b*x,b*x,b*x 21/1*2/6 = 7/1
b*x,b*x,b*x,b*x,b*x,b*x,b*x 7/1*1/7 = 1/1
```

Each call to PasteTerm adds one power to the accumulator and normalization only takes place after the pieces of terms in the accumulator are complete. The binomial coefficient is constructed accumulatively as well. More in general, when for instance we cannot use binomials:

```
Functions U,X,Y,Z;
Symbols a,b;
L F = U*Z^2*a;
id Z = X*b+Y;
```

In this case the various fillings of the accumulator

U,X*b,X*b,a

U,X*b,Y,a

U,Y,X*b,a

U,Y,Y,a

This shows that complete terms are put in the accumulator. Because the accumulator is part of the workspace one should be careful with substitutions like

```
L F = a^100000;
id a = b;
```

This will create 100000 pieces in the accumulator. There are at least two ways around this:

```
L F = a^100000;
id a^n? = b^n;
```

or

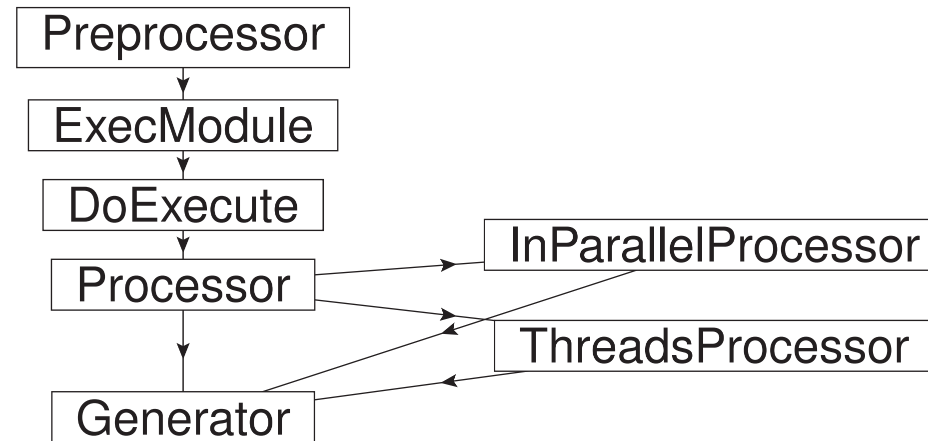
```
L F = a^100000;
Multiply replace_(a,b);
```

It might be possible to build in some intelligence by having FORM analyse the RHS of the id-statement and noticing that it is only a single symbol and hence create the first work-around, but that would slow down

performance and in addition it would violate the FORM philosophy that it does things exactly as the user feeds it in.

One might argue that this accumulator is a weak point in FORM, but one should not forget that FORM has been created for doing realistic calculations very fast, not for being good at artificial benchmarks. (*I could make a lot of remarks about benchmarks here.*)

In the above we studied the dealing with individual terms. But also important is the dealing with expressions. Let us have a look at what happens once the preprocessor decides that it is time to execute a module.



The routine ExecModule, in the file module.c, is to take actions that are module specific. At the moment that is not very much. Only a number of things with respect to floating point. The routine DoExecute on the other hand has much more work to do. First it checks a number of conditions that have to be right before execution can start. An example is that for each goto statement the corresponding label must exist inside the module. It also checks the proper matching of if/endif, switch/case/default/endswitch, do/enddo, term/endterm, inside/endinside and inexpression/endinexpression. It sets the module options on the dollarvariables right, and if there are dollars to be modified without the proper moduleoptions, execution will be switched to just the master processor. It also switches the input and output scratch buffers by calling the RevertScratch routine in the file store.c. Finally, if no error conditions have been set, the routine Processor is called. Once it returns expressions may be written. After this there is a whole cleanup phase. The thoroughness of this cleanup depends on the type of the module

(.sort, .store, .clear, .end).

The task of the routine Processor (in file proces.c) is to locate the expressions that have to be treated and make sure they recieve the proper attention. Expressions may be executed, skipped, hidden or dropped. In addition the processor decides whether complete expressions should be executed in Parallel (call to InParallelProcessor), whether the terms of an expressions should be executed in parallel (call to ThreadsProcessor) or whether they should be executed the regular sequential way. The code for the parallel processing resides in the file threads.c and the sequential code is part of the Processor routine. This sequential code starts for each expression that should be treated with a call the the NewSort routine and terminates with a call to the EndSort routine. When the terms are read it is made sure that they are in the proper notation by removing brackets and the number of dummy indices inside each term is determined. This may be needed during further execution. Also the keep brackets statement takes its effect here. The routine will only process the outside of brackets for which it constructs for each bracket an artificial term that is just the outside. The inside of the bracket is multiplied in just before the result should be sent to StoreTerm in the Generator routine.

When expressions are stored (after a .store instruction) FORM makes a list of all variables in the expression. Their names and numbers are written in an index that is written in front of the expression. This happens for each expression individually. When the stored expression is going to be used again, it may well be that some of its variables do no longer exist in the program. Hence first the index is read, after which the variables that do not exist are declared properly and a renumbering list is made by which the stored numbers can be translated into the numbers as they should be in the program. It is of course possible that there may be a conflict when for instance in the stored expression f is a function, while after the last .store f has been declared a symbol. In

that case there will be an error message and the program cannot continue. Because of the renumbering it is also possible to have arguments for global expressions. These arguments can only be used once the expressions have been stored and retrieval of the terms require renumbering anyway.

At the moment the save statement makes a direct copy from the .str file for stored expressions to the .sav file. It does this expression by expression. The result is that in principle a .str file can be used as a .sav file. This can be useful when there is a crash after a long running time. In the future the .sav files may be treated with gzip (or bzip) to save storage space. This would not upset the use of a .str file as a .sav file. Compressing the .str file with gzip is not a good idea as shown in the follwing program:

```
    S   a,b,x;
    G   F = (a+b+x)^3;
    B   x;
    Print;
    .store
   F =
       + x * ( 3*b^2 + 6*a*b + 3*a^2 )

       + x^2 * ( 3*b + 3*a )

       + x^3 * ( 1 )
```

```
      + b^3 + 3*a*b^2 + 3*a^2*b + a^3;

   S   x;
   L   G = F[x];
   Print;
   .end
  G =
     3*b^2 + 6*a*b + 3*a^2;
```

In this example FORM could pick up the bracket, which would be impossible if the .str file would be gzip compressed.

How to deal properly with the polyratfun when it contains a rational (multivariate) polynomial is not entirely trivial. Because some operations on it can be very expensive, it is important to have to normalize it as few times as possible. This requires a careful handling of the corresponding dirty-flag. I am not 100% sure that it is optimal at the moment.

16 The pattern matcher

Probably the most complicated part of the algebra engine is the pattern matcher. It also counts as one of the better features of FORM. Unfortunately there are also some inefficiencies in it that should be improved in the future, but because it is possible to work around them, this has never been given top priority. The pattern matcher is spread out over various files: pattern.c findpat.c function.c smart.c symmetr.c and wildcard.c. We will deal with the wildcards in a separate section though. The routines in the file pattern.c are more generic. The top level routine is TestMatch which figures out what type of matching has to be done and which routines have to be called for that. It also checks on 'repeat' statements. In addition there are 'operations'. The operations are leftovers from early versions of FORM when a different approach to special statements was taken. The idea there was to create an array of information with the addresses of the relevant routines more or less like in the compiler. But because different statements required different argument fields in later versions the current approach, as explained in the section on the algebra engine (Generator routine), was adopted for later defined statements. If the operations method could be implemented, it would probably make FORM a little bit faster, depending on how the C compilers treat the big case switch in the Generator routine.

Because the LHS of an id statement can contain dollar variables, the first thing is to check whether such variables are indeed present and if so, expand them. After such an expansion one does have to check that the LHS is still a legal LHS. Such expansions have to take place in many other types of statements as well.

The pattern matching is split in two parts: one for symbols, dotproducts and vectors and another for functions and their arguments. This split is still an inheritance of the first version of FORM. In Schoonschip and the very first version of FORM functions were still very primitive and most symbolic manipulations were at the level of symbols and dotproducts. Hence it was important to make their substitutions as efficient as possible. The routines that do this are in the file findpat.c. For the functions a different approach had to be selected. Although originally this was not the case, after a while functions could have complicated arguments which again might contain functions. This suggested a recursive approach which was made even more complicated when the wildcarding became more and more flexible. Specially for the functions there is the need for a proper backtracking mechanism in case the matching would fail halfway a given function but might still match in another occurrence of the same function. And restrictions on the wildcards make this even more necessary. This is where currently the problems are: the functions have a decent backtracking mechanism, the other part does also, but the two cannot talk to each other. Hence:

```
CF  f;
V   p1,p2,p3,p4,q1,q2;
L   F = f(p1,p2)*f(p3,p4)*p1.p2+f(p1,p2)*f(p3,p4)*p3.p4;
id  f(q1?,q2?)*q1?.q2? = 2;
Print;
.end
   F =
      f(p1,p2)*f(p3,p4)*p3.p4 + 2*f(p3,p4);
```

will match in the first term, but not in the second. Of course one can make it work properly by using a function dot as in

```
CF  f,dot;
V   p1,p2,p3,p4,q1,q2;
L   F = f(p1,p2)*f(p3,p4)*dot(p1,p2)+f(p1,p2)*f(p3,p4)*dot(p3,p4);
id  f(q1?,q2?)*dot(q1?,q2?) = 2;
Print;
.end
   F =
      2*f(p1,p2) + 2*f(p3,p4);
```

but that is far from elegant and counterintuitive. This is a problem that should still be solved.

With the functions we need a more recursive approach. Without wildcards it is still relatively simple. A function matches or it does not, and because the terms have been normalized one can work ones way through it from left to right. With wildcards it becomes much more complicated. We do this one function at a time. At each step we have to remember the wildcard assignments up to that point to allow us to fall back. The start is in the routine ScanFunctions which is called from FindRest in the file findpat.c. This routine starts with storing the current wildcard assignments. Next it determines which function should be tried. Complications are noncommuting functions and symmetric argument fields of functions. Once a function should be tried, it calls the routine MatchFunction (or special routines for functions with (anti)symmetric properties). The MatchFunction routine also starts with storing the current wildcard assignments, after which it goes through all possible cases that might

give a match. Each time there is a wildcard involved, it checks with the routine CheckWild whether this wildcard has already an assignment and, if so, whether it is identical. If there was no assignment, one will be made and the routine AddWild will be called. The most complicated wildcard is of course the argument field wildcard. If the function does match eventually it returns with 'success' and ScanFunctions will look whether there are more functions. If so it will call itself to do the next function. If there was no success, the old wildcard settings will be restored. In the case of success the position of the function will be stored in the array AN.RepFunList as an offset to the start of the term.

In the case of functions with (anti)symmetric properties the pattern matching becomes much more problematic. A very simple example is:

```
id  fsym(i1?,i2?)*f1(i2?)*f2(i1?) =
```

Here the first attempt will fail, and only when the symmetric property allows the exchange of the i1 and i2 in fsym we get a match. This can become worse fast:

```
id  fsym(i1?,i2?,i3?,i4?)*f1(i4?)*f2(i3?)*f3(i2?)*f4(i1?) =
```

Most likely FORM will have to try 4! times for the match if it starts from fsym.

The final stage of the pattern matcher is to actually take out what has matched and replace it by a subexpression subterm. At this point it is known what the wildcard values are. Hence the looking for what has to be taken out w.r.t. symbols, dotproducts or vector is rather simple. For functions we have the list of subterms in AN.RepFunList that has been constructed during the matching which tells us which functions should be taken

out. The subexpression (AN.FullProto) is inserted at the place of the first function taken out if there are functions, and otherwise as the last subterm. This is all done in the routine Substitute in the file pattern.c.

17 Exercises part 2

1. permute_(f,?a) gives problems when f is a tensor. Solve this.
2. replace_(function,tensor) gives problems. Try to see how to forbid this on the compiler level, and during runtime because one could have replace_(function,$-variable) in which the dollar becomes a tensor.
3. Look for commands that forget to react properly for repeat action by not setting *A.RepPoint.
4. Look for commands that forget to expand dollar variables in their parameter field.
5. Check whether we can get around the limitation of MAXSUBEXPRESSIONS.
6. Save -gzip filename; + of course the corresponding reading in the Load statement.
7. Figure out whether the tokenizer can intercept identical subexpressions inside the same rhs.
8. Create a function select_(n1,n2,arguments) which can only be used inside a function. It returns the n1 to n2 arguments (n2 inclusive). You can have a look at reverse_ how one can enforce its use inside functions.
9. Create something like ModuleOption statistics; to print statistics only in the given module if they were off.
10. Create an error, or at least a warning for replace_(eps,e_); when eps is a function, rather than a tensor. This can also occur during execution when either is the result of the expansion of a $-variable.