1 Where?

These notes can be found at website "nikhef.nl/∼form/maindir/documentation/workshop" In the documentation directory one can also find a course on the use of FORM. This course is updated every one or two years.

The sources that are described in these notes are the sources of version 5.0.0.beta. They can be found at github.com/vermaseren/form. To install this version you will need a few libraries: pthread, zlib, gmp and mpfr. Most likely the first two are already available in your system and maybe the third as well. You probably will have to install the mpfr library. Of course you will also need a C compiler.

Suggested way to pick up the sources and setup compilation:

```
git fetch origin
git pull origin master --rebase
autoreconf -i
```

During the installation of version 5.0.0.beta, some regression has occurred. My excuses for this. It has our attention. If your work depends on the fixes that have been affected by them, and you do not need the new features of ver5.0.0.beta, you can still use version 4.3 which is up to date as of April first.

2 Introduction

Already more than 30 years FORM is used in particle theory for calculations that without it would either be impossible or very expensive in terms of resources. Over the past 15 years it has also been used for a number of mathematical problems with some unexpected results: Multiple Zeta Values and Euler characteristics are two of them. It would be a great pity if the maintenance and further development of FORM would be hampered by the fact that I, as main developer, would not be able any longer to provide the same service as I managed to give it over the past 38 years. There are two possible solutions for this: the first would be a complete rewrite by other much younger people and the second would be that a number of the people whose research depends on FORM learn to do this maintenance and further development of the sources. This second approach is the purpose of this workshop. I intend to give as much information as possible in the limited time given about the inner workings of FORM. This may not get the attendants to understand every line of code, but at least the main algorithms, methods and problems can be discussed. The practise sessions in the afternoons will be intended to have you look through the code in the same way you would look through it when you want to either add new features or repair some bugs. Because this is the first time I do this, I hope things will not become too confusing. Please ask questions when you do not understand what I am trying to explain.

For those of you who think that the use of the C language is old fashioned, I agree, but only up to a point. It is very clear that algorithms are more important than the choice of language, but when you make an application that many people will be using for extended periods of time an extra gain of 20-50% in speed is something that should not be ignored. Many modern languages tend to concentrate on development speed, rather than execution speed. The best example is the method for setting up a computer algebra system as proposed in the book by Geddes, Czapor and Labahn: "Algorithms for Computer Algebra", (Springer Verlag). The way they propose an object oriented approach makes their method already inherently slower than FORM. They never looked at FORM. The flat data structures of terms and expressions in FORM allow very fast processing even though they look old fashioned and maybe a bit harder to program. But the users of FORM benefit from it.

3 A first overview

FORM is mostly written in the C language. Some parts have been written in C++. Those parts have been written by other people (Jan Kuipers, Ben Ruijl, Toshiaki Kaneko) and I have only provided connecting code. Although most of the C code was written by me, some parts have been written by other people (Misha Tentyukov, Takahiro Ueda, Jens Vollinga, Albert Retey, Denny Fliegner). This is mainly the ParFORM code, the code to communicate with external programs and the code to save intermediate stages of a calculation. In addition Takahiro and Ben have set up the git and github and Takahiro does nearly all of its maintenance. It is possible that I have forgotten a few other contributions. And there are also the criticisms that helped making the language a bit more transparent, the suggestions for improvements and of course the helpful bug reports.

Things that are also very helpful are good problems that can be solved with FORM. The challenge to solve them as efficiently as possible is one of the reasons that FORM has been improved regularly and became what it is today. A few examples are the Mincer program, the color program, the summer/harmpol programs, the three loop splitting and coefficient functions in DIS, the MZV datamine, the FORM version of the GRACE system, the Forcer program, and the Francis Brown method for MZV's. For the last one, see the section on floating point numbers.

The driving force in the development of a program like FORM should always be top level research and attempts to make them either feasible or more efficient by inventing new features that are as generic as possible in such a way that they may be used for other problems as well. Because of this FORM has many original features.

For these lectures we will use the sources of version 5.0.0.beta. Even though this version may not have been released officially, they will be in the not too distant future and they contain some features that better be discussed here.

Let us have a first look at what files make up the Form system.

`comtool.h`	`fwin.h`	`minos.h`	`polyfact.h`	`variable.h`
`declare.h`	`grcc.h`	`mpidbg.h`	`polygcd.h`	`vector.h`
`form3.h`	`grccparam.h`	`mytime.h`	`portsignals.h`	`version.h`
`fsizes.h`	`inivar.h`	`parallel.h`	`structs.h`	
`ftypes.h`	`mallocprotect.h`	`poly.h`	`unix.h`	

`argument.c`	`dict.c`	`if.c`	`namespace.c`	`reken.c`	`symmetr.c`
`bugtool.c`	`dollar.c`	`index.c`	`normal.c`	`reshuf.c`	`tables.c`
`checkpoint.c`	`evaluate.c`	`lus.c`	`notation.c`	`sch.c`	`threads.c`
`comexpr.c`	`execute.c`	`message.c`	`opera.c`	`setfile.c`	`token.c`
`compcomm.c`	`extcmd.c`	`minos.c`	`parallel.c`	`smart.c`	`tools.c`
`compiler.c`	`factor.c`	`model.c`	`pattern.c`	`sort.c`	`transform.c`
`compress.c`	`findpat.c`	`module.c`	`pre.c`	`spectator.c`	`unixfile.c`
`comtool.c`	`float.c`	`mpi.c`	`proces.c`	`startup.c`	`wildcard.c`
`diagrams.c`	`function.c`	`names.c`	`ratio.c`	`store.c`	

`diawrap.cc`	`mytime.cc`	`poly.cc`	`polygcd.cc`
`grcc.cc`	`optimize.cc`	`polyfact.cc`	`polywrap.cc`

These sources contain about 166000 lines of code/comments and a bit more than 4.5 Mbytes. For such a program 89 files is not very much. How can this be maintained efficiently?

The secret is that there is some kind of file system inside the files called folding. Folding was invented around 1974 (see Wikipedia). I ran into it when trying to see whether I could connect a transputer to my Atari ST computer in 1986. The transputer had its own coding language called Occam that came with a folding editor. The implementation however was horrible (ctrl-O and ctrl-P in the code for begin and end of a fold. Other compilers and TEX don't like that) and each fold was stored as a separate file in the file system. In 1986 I was also building an editor for the Atari ST computer and together with the beta testers we decided that this editor (STedi) should also have folds, but in such a way that they could be used in any computer language known to us. Later this editor has been reprogrammed for UNIX and X-windows and its sources are available in the FORM website. Let us have a look at what defines a fold:

The opening and closing lines of a fold each start with three arbitrary characters (a tabulation character counts as one), followed by the character # and then [for the opening line or] for the closing line. After that follows the name of the fold, followed by the colon character. Whatever comes after this is irrelevant and can be used for commentary. For instance in TEX one could use

```
%  #[ chapter one : The beginning
%
\chapter{One}
     This is a very short chapter.
\newpage
%
%  #] chapter one :
```

When the fold is closed (in the STedi editor with function key F6) it looks like

```
%  ## chapter one : The beginning
```

and the fold can be opened again with F7. One can also close all folds with <shift>F6 and open all with <shift>F7. Folds can be nested as in a decent file system. Because the closing line is not displayed when the fold is closed, it rarely pays to put anything after the colon and hence the editor uses this to mark, when writing the file, whether the fold is closed by placing a blank character at the end of the closing line. This way, when the file is read again the editor knows which folds should be closed.

In C files I use the convention to place the opening and closing lines inside commentary in the following way:

```
/*
   #[ one fold :
     some commentary about the contents
*/
    some code
/*
   #] one fold :
*/
```

and closing it will look like

```
/*
   ## one fold :
*/
```

In this way I can play with blank spaces and tabulator characters to give some indentation to the folds when they are nested.

By using the folding one can for instance keep the whole preprocessor inside a single file of more than 7700 lines and almost 200 Kbytes without getting lost. Of course not all files are that extreme. This one is worst.

If you read in the pre.c file and have all folds closed it should look like:

```
/** @file pre.c
 *
 *  This is the preprocessor and all its routines.
 */
/* ## License : */
/*
    ## Includes :
    # [ PreProcessor :
        ## GetInput :
        ## ClearPushback :
        ## GetChar :
        ## CharOut :
        ## UnsetAllowDelay :
        ## GetPreVar :
        ## PutPreVar :
        ## PopPreVars :
        ## IniModule :
        ## IniSpecialModule :
        ## PreProcessor :
        ## PreProInstruction :
        ## LoadInstruction :
```

```
## LoadStatement :
## ExpandTripleDots :
## FindKeyWord :
## FindInKeyWord :
## TheDefine :
## DoCommentChar :
## DoPreAssign :
## DoDefine :
## DoRedefine :
## ClearMacro :
## TheUndefine :
## DoUndefine :
## DoInclude :
## DoReverseInclude :
## Include :
## DoPreExchange :
## DoCall :
## DoDebug :
## DoTerminate :
## DoDo :
## DoBreakDo :
```

```
## DoElse :
## DoElseif :
## DoEnddo :
## DoEndif :
## DoEndprocedure :
## DoIf :
## DoIfdef :
## DoIfydef :
## DoIfndef :
## DoInside :
## DoEndInside :
## DoMessage :
## DoPipe :
## DoPrcExtension :
## DoPreOut :
## DoPrePrintTimes :
## DoPreAppend :
## DoPreCreate :
## DoPreRemove :
## DoPreClose :
## DoPreWrite :
```

```
## DoProcedure :
## DoPreBreak :
## DoPreCase :
## DoPreDefault :
## DoPreEndSwitch :
## DoPreSwitch :
## DoPreShow :
## DoSystem :
## PreLoad :
## PreSkip :
## StartPrepro :
## EvalPreIf :
## PreIfEval :
## PreCmp :
## PreEq :
## pParseObject :
## PreCalc :
## PreEval :
## AddToPreTypes :
## MessPreNesting :
## DoPreAddSeparator :
```

```
## DoPreRmSeparator :
## DoExternal:
## DoPrompt:
## DoSetExternal:
## DoSetExternalAttr:
## DoRmExternal:
## DoFromExternal :
## DoToExternal :
## defineChannel :
## writeToChannel :
## DoFactDollar :
## GetDollarNumber :
## DoSetRandom :
## DoOptimize :
## DoClearOptimize :
## DoSkipExtraSymbols :
## DoPreReset :
## DoPreAppendPath :
## DoPrePrependPath :
## DoTimeOutAfter :
## DoNamespace :
        ## DoEndNamespace :
        ## SkipName :
        ## ConstructName :
        ## DoUse :
        ## UserFlags :
        ## DoClearUserFlag :
        ## DoSetUserFlag :
        ## DoStartFloat :
        ## DoEndFloat :
    # ] PreProcessor :
*/
```

There are more than 100 folds. The outer-most ‘fold’ ”PreProcessor” cannot be closed because we added a blank character after the # character and hence it is not a legal fold. There is indentation because the majority of the folds start with the characters <blank>,<tab>,<tab>, while the first few start with <blank>,<blank>,<tab>. If we open for instance the Includes fold we see

```
   #[ Includes :
*/
#include "form3.h"

static UBYTE pushbackchar = 0;
static int oldmode = 0;
static int stopdelay = 0;
static STREAM *oldstream = 0;
static UBYTE underscore[2] = {'_',0};
static PREVAR *ThePreVar = 0;

static KEYWORD precommands[] = {
     {"add"           , DoPreAdd       , 0, 0}
    ,{"addseparator" , DoPreAddSeparator,0,0}
    ,{"append"        , DoPreAppend    , 0, 0}
    ,{"appendpath"    , DoPreAppendPath, 0, 0}
    ,{"assign"        , DoPreAssign    , 0, 0}
    ,{"break"         , DoPreBreak     , 0, 0}
    ,{"breakdo"       , DoBreakDo      , 0, 0}
    ,{"call"          , DoCall         , 0, 0}
    ,{"case"          , DoPreCase      , 0, 0}
    ,{"clearflag"    , DoClearUserFlag, 0, 0}
    ,{"clearoptimize", DoClearOptimize, 0, 0}
    ,{"close"        , DoPreClose     , 0, 0}
    ,{"closedictionary", DoPreCloseDictionary,0,0}
    ,{"commentchar"  , DoCommentChar  , 0, 0}
    ,{"create"       , DoPreCreate    , 0, 0}
    ,{"debug"        , DoDebug        , 0, 0}
    ,{"default"      , DoPreDefault   , 0, 0}
    ,{"define"       , DoDefine       , 0, 0}
    ,{"do"           , DoDo           , 0, 0}
    ,{"else"         , DoElse         , 0, 0}
    ,{"elseif"       , DoElseif       , 0, 0}
    ,{"enddo"        , DoEnddo        , 0, 0}
#ifdef WITHFLOAT
    ,{"endfloat"     , DoEndFloat     , 0, 0}
#endif
    ,{"endif"        , DoEndif        , 0, 0}
    ,{"endinside"    , DoEndInside    , 0, 0}
    ,{"endnamespace" , DoEndNamespace , 0, 0}
    ,{"endprocedure" , DoEndprocedure , 0, 0}
    ,{"endswitch"    , DoPreEndSwitch , 0, 0}
```

```
,{"exchange"     , DoPreExchange  , 0, 0}
,{"external"     , DoExternal     , 0, 0}
,{"factdollar"   , DoFactDollar   , 0, 0}
,{"fromexternal" , DoFromExternal , 0, 0}
,{"if"           , DoIf           , 0, 0}
,{"ifdef"        , DoIfydef       , 0, 0}
,{"ifndef"       , DoIfndef       , 0, 0}
,{"include"      , DoInclude      , 0, 0}
,{"inside"       , DoInside       , 0, 0}
,{"message"      , DoMessage      , 0, 0}
,{"namespace"    , DoNamespace    , 0, 0}
,{"opendictionary", DoPreOpenDictionary,0,0}
,{"optimize"     , DoOptimize     , 0, 0}
,{"pipe"         , DoPipe         , 0, 0}
,{"preout"       , DoPreOut       , 0, 0}
,{"prependpath"  , DoPrePrependPath,0, 0}
,{"printtimes"   , DoPrePrintTimes, 0, 0}
,{"procedure"    , DoProcedure    , 0, 0}
,{"procedureextension" , DoPrcExtension   , 0, 0}
,{"prompt"       , DoPrompt       , 0, 0}
,{"redefine"     , DoRedefine     , 0, 0}
```

```
    ,{"remove"        , DoPreRemove    , 0, 0}
    ,{"reset"         , DoPreReset     , 0, 0}
    ,{"reverseinclude"   , DoReverseInclude   , 0, 0}
    ,{"rmexternal"    , DoRmExternal   , 0, 0}
    ,{"rmseparator"   , DoPreRmSeparator,0, 0}
    ,{"setexternal"   , DoSetExternal  , 0, 0}
    ,{"setexternalattr"  , DoSetExternalAttr  , 0, 0}
    ,{"setflag"       , DoSetUserFlag  , 0, 0}
    ,{"setrandom"     , DoSetRandom    , 0, 0}
    ,{"show"          , DoPreShow      , 0, 0}
    ,{"skipextrasymbols" , DoSkipExtraSymbols , 0, 0}
#ifdef WITHFLOAT
    ,{"startfloat"    , DoStartFloat   , 0, 0}
#endif
    ,{"switch"        , DoPreSwitch    , 0, 0}
    ,{"system"        , DoSystem       , 0, 0}
    ,{"terminate"     , DoTerminate    , 0, 0}
    ,{"timeoutafter"  , DoTimeOutAfter , 0, 0}
    ,{"toexternal"    , DoToExternal   , 0, 0}
    ,{"undefine"      , DoUndefine     , 0, 0}
    ,{"use"           , DoUse          , 0, 0}
    ,{"usedictionary", DoPreUseDictionary,0,0}
    ,{"write"        , DoPreWrite     , 0, 0}
};

/*
   #] Includes :
```

In each .c or .cc source file there is such a fold and it calls the necessary include files and contains local variables. The file form3.h contains calls to most of the .h files. Hence one only has to worry about very local things that are needed in only one or two files. All global variables are contained inside just a few data structs that are defined in the file structs.h. We will come back to the how and why of that later.

The most important object inside this particular Includes fold is however the precommands array. It contains the names of all preprocessor commands and the name of the routine that will process the command. The KEYWORD structure is of course defined in the file structs.h. The last two elements in each entry are used in different contexts. It is important, when you want to add a preprocessor command, to make sure that the list of commands remains strictly alphabetic, because FORM searches through it with a binary search. This holds as well for almost all other arrays of type KEYWORD in other files.

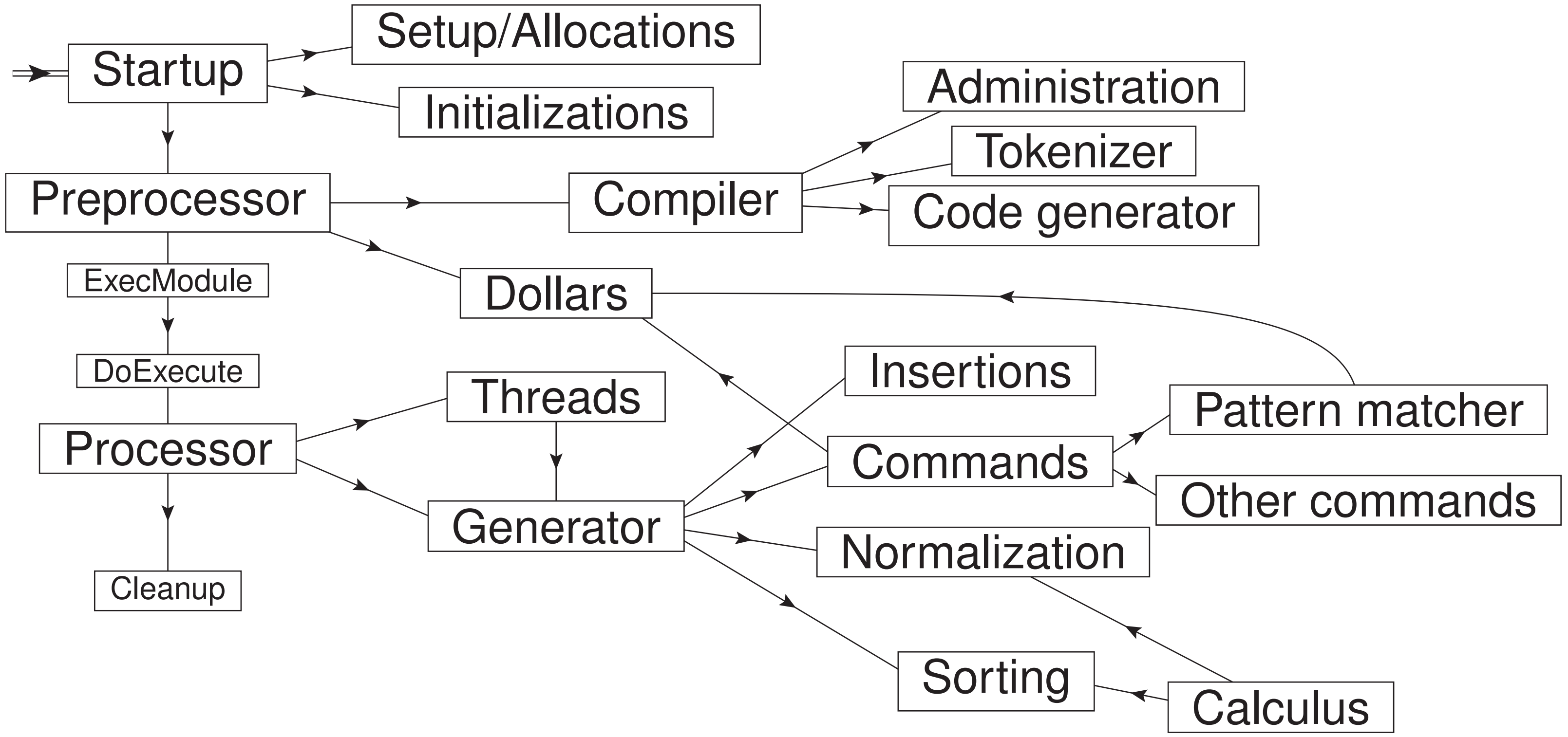
A simpler fold would be

```
  #[ DoClearOptimize :

        Clears all relevant buffers of the output optimization
*/

int DoClearOptimize(UBYTE *s)
{
    if ( AP.PreSwitchModes[AP.PreSwitchLevel] != EXECUTINGPRESWITCH ) return(0);
    if ( AP.PreIfStack[AP.PreIfLevel] != EXECUTINGIF ) return(0);
    DUMMYUSE(*s);
    return(ClearOptimize());
}

/*
  #] DoClearOptimize :
```

The first two lines check whether the instruction should be executed, because it might be inside a branch of an #if or #switch construction that is being skipped. The DUMMYUSE macro is to prevent that the compiler gives a warning that the variable is not used. I prefer code that, even with all warning levels on, keeps the compiler quiet.

When you make changes in the FORM code, it would be nice to stay with the various conventions. This will make things easier for other people. If you think that something should really be different, please discuss this first with others on the github site. One problem can be that your favorite editor refuses to write tabulation characters, thereby messing up the fold lines. Be sure you have the proper settings.

4 Layout



Of course this scheme could have contained more modules, but these are the most important ones.

5 Conventions and notations

Before we dive into specific code it is a good idea to know what constitutes a term, how terms are organized and how declarations are treated. In addition it is very important to know how the global data are structured and why. Let us start with expressions and terms.

5.1 Expressions and terms

In Form an expression is a sequence of terms. This means that there is a single linear buffer that contains all the terms of the expression and the expression is the sum of all its terms. This buffer is in principle on disk, but when all active expressions together use less than a given amount of space (SCRATCHSIZE in the setup), it is kept inside the main CPU memory and if it gets bigger the buffer is used as a cache for this file.

A term consists of a sequence of numbers. These numbers will never contain pointers. All numbers are of the type WORD, although inside the coefficient there may be numbers of the type UWORD. A WORD is usually half the size of a long and a UWORD is the unsigned version of a WORD. In the now popular 64-bits version of Form a WORD is 4 bytes. In the older 32-bits version a WORD was 2 bytes. This would often place restrictions on numbers that would have to fit inside a single WORD, like numbers of variables. Hence at the moment the preference is to drop the 32-bits version in favor of the 64-bits version. Beware of compilers that try to make life easy on users by having their own definition of WORD and other quantities (this happened to me in the past and the author who made this disastrous choice was very proud of it. Result: compiler useless).

The first WORD of a term indicates the total number of WORDs in the term. After this first WORD follow subterms. A subterm consists of a WORD that indicates the type of the subterm, next the number of WORDs in the subterm and after that its type dependent contents. The coefficient of the term comes at the end and has its length indicator as its last WORD. This is not necessarily a positive length, because it also contains the sign of the coefficient. Example:

```
    L F = 10^24/7;
    Print "%r";
    .end
8  -1593835520  466537709  54210  7  0  0  7
```

The coefficient consists of a numerator, a denominator and its length indicator. The numerator and the denominator contain the same number of UWORDs. Hence the shorter of the two may have leading zeroes. This was selected in the early stages of the development of FORM because it was realized that in most cases the coefficients are rather simple and hence having two length indicators was statistically a bit wasteful (take also function arguments into account). A consequence of this is also that the length indicator is always an odd integer. This may be negative because the sign of the coefficient is part of it. A spinoff was that potentially in the future even length indicators might be used for different type(s) of numbers, like floating point numbers. In practise this has never materialized and the implementation of floating point numbers in version 5 has followed a different path, because by then changing the code in all places where numbers are treated was way too complicated and definitely error prone. We will see the floating point implementation at a later stage. The numerators and denominators are having their least significant digits first, allowing them to be treated as arrays that can be truncated easily when the leading digits cancel in for instance subtraction.

If there is a zero in the position where a term is expected, this indicates the end of the expression. The size of the expression is the sum of the sizes of its terms plus the size of this zero. The length indicators of the terms makes expressions effectively singly linked lists of terms and each term a singly linked list of subterms.

5.2 Subterms

As mentioned, a subterm starts with a WORD that indicates its type, followed by its length in WORDs. The types of subterms can be divided into two categories: special subterms and functions. These functions include the built-in functions and the user defined functions. The internal names and their numbers of the special subterms and the built-in functions can be found in the file ftypes.h. Effectively the special subterms form a class of functions that do not follow the generic rules of functions and have each their own special treatment. Maybe it would be better to change this but considering the amount of work and the risk of errors, it was not considered to be worth the effort. It is however anticipated that no more of these special functions will be defined. The most important special subterms are (with their current code between parentheses)

- SYMBOL(1) The symbols are presented by a pair of WORDs. The first indicates the number of the symbol and the second its power. In a normalized term all symbols are packed inside a single subterm as in

  ```
  1 8 2 5 4 -2 5 7
  ```

  meaning symbol 2 to the power 5, symbol 4 to the power -2 and symbol 5 to the power 7. The symbols are ordered by their internal number.

- DOTPRODUCT(2) Dotproducts are represented by three WORDs. The first two are the two vectors in the dotproduct and the third is its power. Again, in a normalized term the dotproducts are collected inside a single subterm.
- VECTOR(3) A string of pairs of vectors with their index.
- INDEX(4) A string of indices or vectors without indices.
- EXPRESSION(5) A subterm of type EXPRESSION is to be replaced by the content of that expression. This expression must exist. Its format is:

  ```
  EXPRESSION,length,number_in_compiler_buffer,power,0,wildcard_information
  ```

  The part about the wildcards will be explained later. In the case of expressions this part will only be used for stored expressions.
- SUBEXPRESSION(6) This is an extremely common subterm, because it is used for each pair of parentheses in an input or any other type of composite object. In id statements with wildcard variables it has to be possible to deal with them and pass them on to deeper levels of subexpressions. In a sense the substitution of SUBEXPRESSION subterms by their content is the core of the Form algebra engine. Its format is

  ```
  SUBEXPRESSION,length,number_in_list_of_expressions,power,number_of_
  compiler_buffer,wildcard_information
  ```

The wildcard information will be explained in the section about wildcards and the compiler buffers in the section about the compiler.

- SNUMBER(16) A short (one WORD) number that is to be incorporated into the coefficient. For this it has the format

  ```
  SNUMBER 3 number
  ```

  It is however mainly used in function arguments to indicate an argument that is just a single one WORD number.

- HAAKJE(18) When we use brackets in the sorting of the output, each term is split into two parts: the part that is outside the bracket (haakje in dutch) and the part that is inside the bracket. These two parts are separated by a subterm of type HAAKJE which has the format:

  ```
  HAAKJE,3,0
  ```

  Technically the zero is not needed but at an early stage of the development of FORM it was put there in anticipation of the construction of levels of brackets. At a later stage some preparatory code for this was added, but in essence it was never finished, because during the project various complications kept popping up. The fact that every term that belongs to the same bracket still has this information outside the bracket is no real problem w.r.t. storage as will be seen in the subsection about data compression.

- DELTA(19) Contains pairs of indices, indicating a product of Kronecker deltas.

There are a few more special subterms that are for special use only. We will see these when needed. When we look inside the file ftypes.h we find:

```
#define SYMBOL 1
#define DOTPRODUCT 2
#define VECTOR 3
#define INDEX 4
#define EXPRESSION 5
#define SUBEXPRESSION 6
#define DOLLAREXPRESSION 7
#define SETSET 8
#define ARGWILD 9
#define MINVECTOR 10
#define SETEXP 11
#define DOLLAREXPR2 12
#define FUNCTION 20

#define TMPPOLYFUN 14
#define ARGFIELD 15
#define SNUMBER 16
#define LNUMBER 17
```

```
#define HAAKJE 18
#define DELTA 19
#define EXPONENT 20
#define DENOMINATOR 21
#define SETFUNCTION 22
#define GAMMA 23
#define GAMMAI 24
#define GAMMAFIVE 25
#define GAMMASIX 26
#define GAMMASEVEN 27
#define SUMF1 28
#define SUMF2 29
     .
     .
#define PHI 115
#ifdef WITHFLOAT
 #define FLOATFUN 116
 #define TOFLOAT 117
 #define TORAT 118
 #define MZV 119
 #define EULER 120
 #define MZVHALF 121
 #define AGMFUNCTION 122
 #define GAMMAFUN 123
 #define MAXBUILTINFUNCTION 123
#else
 #define MAXBUILTINFUNCTION 115
#endif

#define FIRSTUSERFUNCTION 150
```

The difference between MAXBUILTINFUNCTION and FIRSTUSERFUNCTION allows one to add a few more built in functions without invalidating existing .sav files. A similar construction is used for the symbols:

```
#define ISYMBOL 0
#define PISYMBOL 1
#define COEFFSYMBOL 2
#define NUMERATORSYMBOL 3
#define DENOMINATORSYMBOL 4
#define WILDARGSYMBOL 5
#define DIMENSIONSYMBOL 6
#define FACTORSYMBOL 7
#define SEPARATESYMBOL 8
```

```
#define BUILTINSYMBOLS 9
#define FIRSTUSERSYMBOL 20
```

Here one has to pay attention to the fact that i_ has the number zero. This is an unfortunate choice that stems from the time that it was the only system defined symbol. It may be possible to change this, but I have never really looked into it. It would be complicated by the fact that sometimes it is tested for by code like

```
    if ( *t == 0 )
```

without the use of the macro.

Functions have their own generic conventions. All functions start with a header of FUNHEAD WORDs. The current value of FUNHEAD is 3, but by the use of macro's it should be rather painless to increase this to 4 or even more, if this will be needed at some time in the future. This would for instance be the case if one would like to add a power to functions. Currently this header is

```
    Number_of_function, full_length_of_function_with_arguments, flags
```

The flags are there to help speeding up normalization. They can indicate whether the arguments still need to be normalized, or whether the arguments of a symmetric function still need to be symmetrized, etc. After the header follow the arguments. There are three types of arguments: the generic arguments, the fast arguments and the arguments of functions that have been declared as tensors. Let us start with the generic arguments as that will immediately show why we also have the fast arguments.

A generic argument starts with an argument header of length ARGHEAD which currently has the value 2. Again, things have been programmed such that if future developments would need this to be increased, there should be no problem, because there is a macro ARGFILL that will complete the header by writing a sufficient number of zeroes. The first WORD of the header is the number of WORDs in the argument and the second word contains the 'dirty flag', telling whether the argument still needs to be worked out or normalized. After this follows the argument which is like an expression, therefore a sequence of terms, with the exception that there is no trailing zero because the size of the argument is known already . Hence the function

```
f(a+b,a-2/3*c)
```

would be, after normalization

```
nf,39,0,18,0,8,1,4,na,1,1,1,3,8,1,4,nb,1,1,1,3
       ,18,0,8,1,4,na,1,1,1,3,8,1,4,nc,1,2,3,-3
```

in which nf,na,nb,nc are the internal numbers of f, a, b, c. One can also obtain this result running the following program:

```
Symbols a,b,c;
CFunction f;
On HighFirst;
L   F = f(a+b,a-2/3*c);
Print "%r";
.end
```

```
43  150  39  0  18  0  8  1  4  20  1  1  1  3  8  1  4  21  1  1  1  3
                18  0  8  1  4  20  1  1  1  3  8  1  4  22  1  2  3  -3
    1  1  3
```

in which `%r` indicates raw format. One can see that the user defined objects start indeed at 20 and 150 respectively.

In this notation

```
    f(a)+f(2)
```

would become

```
    17,nf,13,0,10,0,8,1,4,na,1,1,1,3,1,1,3, 13,nf,9,0,6,0,4,2,1,3,1,1,3
```

This seems a bit wasteful, specially because it occurs so often. Hence we have the fast notation for simple arguments. In this fast notation the above would become

```
    Symbols a;
    CFunction f;
    L   F = f(a)+f(2);
    Print "%r";
    .end
9  150  5  0  -1  20  1  1  3
9  150  5  0  -16  2  1  1  3
```

with the convention that an argument that starts with a negative number is a fast argument. Fast arguments can have only length 2 or length 1. If it is a SYMBOL, VECTOR, INDEX or SNUMBER (= short number) it has length 2 and if it is a function it has length 1. There are routines ToGeneral and ToFast in the file tools.c to make/try conversions when needed and a macro NEXTARG to skip to the next argument (when for instance counting arguments).

Tensors are special functions that can have only indices and/or vectors for their arguments. When there are vectors, a tensor is supposed to be linear in those vectors and hence, when things are normalized each argument is either a single index or a single vector. Therefore a tensor has the regular function header but after that a string of single WORD arguments only.

Tables are also special functions. In the data structs that keep track of the functions both tensors and tables have special markings.

There exist two special functions without an external name. They are the denominator function and the exponent function. These functions have some automated action built in, but are not very powerful. The problem with them is that for more complicated algorithms one would need them to have extra properties. To have to spend lots of time on regularly considering such properties, while an user defined function with user defined rules will do the job as well, has been considered inefficient. Hence these functions are tolerated, but will take only limited types of action. They are mainly leftovers of version 1 and an inheritance of Schoonschip.

5.3 Compression

We mentioned data compression already when discussing the subterm HAAKJE. It should be noted that in a sorted expression sequential terms usually start with a number of WORDs that are identical. Let us take the example

```
    f(a)*x+f(a)*y
```

This would be

```
    13,nf,5,0,-SYMBOL,na,SYMBOL,4,nx,1,1,1,3
   ,13,nf,5,0,-SYMBOL,na,SYMBOL,4,ny,1,1,1,3
```

If we adopt the notation

```
    13,nf,5,0,-SYMBOL,na,SYMBOL,4,nx,1,1,1,3
   ,-7,5,ny,1,1,1,3
```

indicating that we have to copy the first 7 relevant WORDs from the previous term and then paste on the 5 following WORDs, we can then reconstruct the full length to obtain the complete second term. This convention needs one restriction: this copying is not allowed to run into the coefficient. In the section on sorting I will show why. Ignoring this rule led to some interesting bugs in the past.

It should now be clear that the problem with the brackets is not very relevant:

```
    f(a)*(x+y+z)
```

is without compression

```
    16,nf,5,0,-SYMBOL,na,HAAKJE,3,0,SYMBOL,4,nx,1,1,1,3
   ,16,nf,5,0,-SYMBOL,na,HAAKJE,3,0,SYMBOL,4,ny,1,1,1,3
   ,16,nf,5,0,-SYMBOL,na,HAAKJE,3,0,SYMBOL,4,nz,1,1,1,3
```

but with compression

```
    16,nf,5,0,-SYMBOL,na,HAAKJE,3,0,SYMBOL,4,nx,1,1,1,3
    ,-10,5,ny,1,1,1,3
    ,-10,5,nz,1,1,1,3
```

This compression technique is used in all files and during the sorting in the large buffer. Only in the small buffer it is not used because the sorting in the small buffer is done by pointers. It is also not used inside generic function arguments. It should be noted that when the coefficients don't take up an important fraction of the length of the terms, this compression technique saves typically about a factor 2 in storage space at very little cost in CPU time.

For the sort file we use, in addition to the above compression method the gzip compression (lately it is also possible to use bzip). Because there are many small numbers, that, rather than a single byte, use a full WORD, this additional compression saves about a factor 4. When one does not have very fast disks, it also saves execution time, in spite of the time needed for the compression and the decompression. When the first level compression is not used, but only gzip is used the compression ratio is typically 5 to 6.

One may wonder why the gzip compression is only used in the sort file. The problem with using it in the scratch files is that at times we want to be able to access the contents of brackets of expressions in the scratch system. It is, to my knowledge, not possible to jump to an arbitrary position in a gzipped file and gunzip from there. Maybe it is possible to make separate gzipped brackets, although that would break down when there are very many brackets with each very short contents. It should however be possible to use gzip as well for saved files. This has not (yet?) been implemented (See one of the exercises).

It is possible to invent better byte oriented compressions, based on knowledge of what the various objects stand for. There has been some experimenting with this, specifically for multivariate polynomials. In combination with the indigenous FORM compression it can compete with the best techniques that are designed purely for polynomials. And those techniques allow typically only a limited number of variables and a limited maximum power. In FORM it can be made general, but it would still require quite some work to implement this and the question will be whether the shorter stored expressions will gain back the coding and decoding time. These compactified expressions could still be made shorter with gzip. The conclusion is that it is possible to gain extra compactification and this method would allow its use in the scratch files.

5.4 Structs

Nearly all structs are defined in the file structs.h. The main exceptions are the .h files that come with the C++ libraries in which all C++ objects are defined. There are structs for symbols, for vectors etc. And there are structs for the various files, dollar variables, compiler buffers and so on. The major thing to notice here is that nearly all external variables are organized inside a very few structs which again are inside a single struct A for sequential FORM and two structs A and B for TFORM. There are two reasons for this and programmers are advised to keep this in mind. The first reason was discovered when TFORM did not exist yet. When an external variable is used the computer has to load an absolute address into a register and then obtain the data from the address that register points to, with or without an offset if we have an array and we refer to an array element. With a data struct things go a bit differently. The address of the struct has to be put inside a register again and now the variables in the structs are obtained with an offset to the content of that single register. This is usually faster and takes less space than when another address has to be loaded. By putting everything in the A struct the compiler can make somewhat faster code because the adressing is faster and it has more address registers available for optimizations. When this was put in, FORM became 5-10% faster.

By putting variables according to topic inside substructures the design of TFORM became relatively easy. All private data for a worker would be put in a struct *B (which would now not be part of A any longer) and with the definition of a few macro's the sources of FORM and TFORM are still more or less identical. For FORM we use

```
    #define AT A.T
```

while for TForm we use

```
    #define AT B->T
```

when T refers to one of the substructs that would be private to the worker. The identity of the worker is fixed by its B struct which is passed to each routine that needs it with a macro BHEAD which contains 'B,' in TForm and is empty in sequential Form. In declarations PHEAD takes this role because there the data type has to be included. In the case that the B struct cannot be passed to a given routine there is the macro GETIDENTITY which asks the system which worker number is the current number and then the proper B gets picked up from the array AB which contains the addresses of the various B structs. This is slightly costlier than passing the B by means of BHEAD.

All structs are properly padded at the end with the use of a few macro's in which the numbers of the various variable types are specified. This avoids potential problems with arrays of structs, although modern compilers should do this automatically, possibly with a warning. In the past it was a necessity.

5.5 External declarations and macro's

External declarations of subroutines and most macro's have been put together in a single file declare.h. The convention used is that macro's that are constants have their names fully capitalized. There are no case distinctions between variables and subroutines, because the parentheses make subroutines easily recognizable. Hence we do not use camel or snake conventions as in for instance the Rust language. Again, declarations that are private to C++ libraries are given inside the header files that come with those libraries. When there are files that connect the C++ libraries to the regular C parts of Form, reference to the C++ parts may be restricted to those files, while the C bindings will be in the declare.h file.

To see which names correspond to macro's it is best to use grep on the include files as in "grep #define *.h > out" and then look in the file out what names are basically protected.

5.6 Offsets

There are several types of offsets in Form. The first one is for functions. Because the type of a subterm is the number of a function, but the function numbers start currently at 20, the number that a function has in the array of functionproperties, has to be corrected by the amount FUNCTION (=20). This means that when we look up a name in the subroutine GetName and the return code is that it is a function, the number returned is the number in the array and hence in the subroutine GetFunction we add FUNCTION before we return its value in the variable funnum. Similarly we have to subtract FUNCTION when we want to look up the properties of a function that we encounter inside a term by looking in the functions array. Actually the functions array is a rather complicated object. This will be explained later. To give an example: to check whether a function in `*t` has symmetric properties:

```
    if ( functions[*t-FUNCTION].symmetric > 0 )
```

It is not safe to change the value of FUNCTION, because the numbers of all built in functions contain this offset. Hence one would have to change those numbers as well. It would also invalidate all previously saved files.

The next offsets deal with indices and vectors. Because of the Schoonschip convention of placing vectors in the position of an index when the index of the vector is contracted with that index, we would like an uniform treatment. In addition we have fixed indices. Hence for indices we have AM.OffsetIndex which by default is 128 but can be changed in the setup. And we give vectors a negative value with a very negative AM.OffsetVector. Because we only allow a maximum number of variables of a given type, we can avoid vectors to ever have a positive number. Actually the offset here is even stricter, because we also have to leave space for wildcard vectors. More about that when we deal with wildcards.

If any of these offsets are changed in the future, all stored/saved files should be regenerated by reading them in the 'old' version, printing them out as text files and then reading and saving them in the 'new' version. For very big files this may be time consuming and one may need to allow compiler buffers to be extra large. It might be worthwhile to write a special conversion program that does not need textual representation, but that can deal with the old and new values to make the conversions.

When we have an array of buffers, like in a compiler buffer, we have no idea how large such an array should be. The same holds for the name lists. Hence the name of a variable is held inside a buffer and the struct that describes this variable refers to the position of the name by its offset from the start of the names buffer. This way, when the names buffer is full and we have to add more names, we allocate a new buffer that is typically twice as large as the old buffer, copy the content of the old buffer to the new buffer, release the old buffer and after that the only address we have to update is the address of the buffer. We see here how this is done when we store values of the fact_ (factorial) function in reken.c:

```
if ( j > AT.mfac ) {  /* double the pfac buffer */
    LONG *p;
    p = (LONG *)Malloc1((AT.mfac*2+2)*sizeof(LONG),"factorials");
    i = AT.mfac;
    for ( i = AT.mfac+1; i >= 0; i-- ) p[i] = AT.pfac[i];
    M_free(AT.pfac,"factorial offsets"); AT.pfac = p; AT.mfac *= 2;
}
if ( AT.pfac[j] + nc >= AT.sfact ) { /* double the factorials buffer */
    UWORD *f;
    f = (UWORD *)Malloc1(AT.sfact*2*sizeof(UWORD),"factorials");
    ii = AT.sfact;
    c = AT.factorials; b = f;
    while ( --ii >= 0 ) *b++ = *c++;
    M_free(AT.factorials,"factorials");
    AT.factorials = f;
```

```
        AT.sfact *= 2;
    }
```

AT.sfact is the size of the buffer with the values of the factorials and AT.pfac is the array with offsets to find the values. We use the internal routine Malloc1 and M_free because they allow us to print messages if anything does not go as it should. The technique shown above is used in many places in the Form code. In some cases, like in the compiler, it can happen that in a single module very much memory is needed and the buffer gets expanded to a great size. In that case, in later modules, slowly parts of that memory can be returned with the use of realloc.

The most complicated offsets are those for the indices. The reason is that we have an extra type of indices: dummy indices, and those have to be indicated as well. And in addition we have to keep track of how many we have, to make sure that if we add one, we will indeed get a new one.

```
    I   i1,i2,i3;
    Off Statistics;
    CF  f;
    L   F = f(i1,i2)*f(i2,i1);
    Sum i1,i2;
    .sort
    L   FF = F^3;
    Print;
    .end

   F =
      f(N1_?,N2_?)*f(N2_?,N1_?);

   FF =
      f(N1_?,N2_?)*f(N2_?,N1_?)*f(N3_?,N4_?)*f(N4_?,N3_?)*f(N5_?,N6_?)*f(N6_?,N5_?);
```

To indicate that an index is a dummy (summed over) index we have an extra offset which is AM.IndDum.

Starting address	Variable	What is in it/remarks
-AM.OffsetVector		Regular vectors
-AM.OffsetVector+WILDOFFSET		Wildcard vectors
-AM.OffsetVector+2*WILDOFFSET		Special indices like 5_ in $\gamma\_5(i)$
0		Fixed indices
AM.OffsetIndex		Regular indices
AM.OffsetIndex+WILDOFFSET	AM.WilInd	Wildcard indices
AM.OffsetIndex+2*WILDOFFSET	AM.DumInd	Contractable indices
AM.OffsetIndex+3*WILDOFFSET	AM.IndDum	Dummy indices
	AN.IndDum	Used for Keep Brackets
	AR.CurDum	Currently highest dummy index
AM.OffsetIndex+4*WILDOFFSET		Should still be positive

With these offsets all the hybrid handling of vectors and indices can be done. It does put a maximum on the value of WILDCARDOFFSET. In the 64 bits version this maximum is not very restrictive though. In the 32-bits version it would be $\mathcal{O}(2^{12})$ which could potentially cause problems in programs that have very many indices or vectors.

Keep Brackets presents a special problem when there are dummy indices inside and outside the bracket. In the next example one can see it working. I have to admit: it did not work at all in the beginning. I had completely forgotten about it until somebody reported the bug.

```
    CF  f1,f2;
    I   i1,i2,i3;
    Off Statistics;
    Format nospaces;
    L   F = f1(i1)*f2(i1,i2,i2);
    Sum i1,i2;
    B   f1;
    Print;
    .sort
   F=
      +f1(N1_?)*(f2(N1_?,N2_?,N2_?));
    Keep brackets;
    id  f1(i1?) = f1(i1,i2,i2);
    Sum i2;
    Print " >>> %t";
    B   f1;
    Print;
    .end
 >>> +f1(N1_?,N3_?,N3_?)
   F=
      +f1(N1_?,N2_?,N2_?)*(f2(N1_?,N3_?,N3_?));
```

6 Calculus

Calculus is usually over the rationals. There is also the possibility to work modulus a positive integer. In the last case certain operations, like dividing, can only be done if the number is a prime number. The relevant routines are all in the file reken.c. There are routines for the addition, subtraction, multiplication, division and taking the GCD of what are called long integers in FORM. Those are integers that can be arbitrarily long, with the restriction that they cannot take more than half the maximum term size minus a few WORDS. To help with these calculations there are several buffers and there is a special Malloc and Free to get such a buffer. These buffers are not returned to the system afterward, allowing FORM to provide them very quickly when needed. There is a macro that helps with obtaining the address of a coefficient in a term (GETCOEF), and there are routines to pack(Pack) and unpack(UnPack) the numerator and denominator into complete coefficients as stored in a term.

The routine that is most time consuming is the GcdLong routine. It is very hard to improve on the Euclidean algorithm and/or its binary equivalent. However I found how one can improve on the generalized Euclidean algorithm. It turned out that that had been discovered already in the 1930's and is also described in the book by Knuth, but the extra improvement on it had only been published the year before I found my own version of it. This to show that the algorithms in FORM are often state of the art.

When the use of very big fractions became more common, it was deemed better to start using the GMP library for several reasons:

1. For multiplication and division of very big numbers there are rather complicated but more efficient algorithms

that I do not want to program.

2. The library comes with the systems library and contains assembler routines that are more efficient than what can be done from the C language.

3. It is maintained by experts which absolves me from that task.

There is also a disadvantage: the notation of its numbers is different from the Form notation, starting with the native word size. Hence each time the GMP needs to be used one has to translate which costs time. Therefore the GMP is only used when numbers take more than 5 UWORDs. It is not known whether 5 is the optimal number here, but considering that the Form algorithms are quite good one cannot loose too much this way.

Having the routines for long integers, the routines for rationals are not very complicated. This leaves the routines for modulus calculus. Here there are two complications. The first is division, or in other words, determining the inverse of a number in modulus calculus. This is done with the extended Euclidean algorithm. If the modulus number is not very high one can even tabulate the inverses. The second complication is when one would like to print the numbers as powers of a generator. For this it is best to tabulate all powers. This puts of course restrictions on the size of the prime number used for the modulus calculus.

The file reken.c contains also a few special numerical functions like fac_, bernoulli_, moebius_, random_ and prime_. Factorials are kept in a table which is expanded whenever needed. The same holds for the Bernoulli numbers, the Moebius function and some lists of prime numbers. The random numbers are generated according to the algorithm advocated in the book by Knuth, but not using the magic pair 24,55 which apparently suffers from a number "very close to the edge" rather early in the sequence. Here we use the pair 38,89 which also gives a much larger cycle. Nowadays there exist other good random number generators, allowing the possibility to build in a variety of them and giving the user a choice which one to apply.

The implementation of arbitrary precision floating point numbers is dealt with in a different way and is described in a separate section. It uses a special function float_ and the GMP and the mpfr libraries.

7 Names

Form has several types of names. The names of variables are most complicated because of the possibility that their names can be enclosed in a matching pair of braces as in [a+b]. Then there is the use of the underscore character in systems defined names. The names of preprocessor variables and the names of dollar variables are purely alphanumeric with the first character being alphabetic. The names of labels can be any alphanumeric string. In addition the names of statements and instructions are mostly alphabetic.

In the case that a name is (mostly) alphabetic the name can be read out on the spot. For variables, with their more complicated rules, it is advised to use the SkipAName routine to obtain their full name and search whether it exists already with the GetName routine. It takes care of tricky brackets problems as in variables with a name like `[({]`. It also brings a consistency and if ever the conventions need adaptations, one can basically implement them at a single location. At this point we are thinking about separated name spaces, run by preprocessor instructions, that would allow the construction of program segments with local variables. The idea here would be that if one would define a namespace as in

```
#namespace L2
   Symbol a,b,c;
   .....
#endnamespace
```

the 'local' variables would actually get the names L2_a,L2_b,L2_c. Namespaces could be nested this way. Some routines have been constructed for this but because there are still problems with what the proper conventions should be, and what to do with local variables that survive beyond their name space, the project was never finished. Because of this one may encounter some unexpected code that can read this convention.

The declaration of the GetName routine (in names.c) is:

```
int GetName(NAMETREE *nametree, UBYTE *namein, WORD *number, int par)
```

We have to specify in which nametree the program has to search, because the variables, the preprocessor variables and the dollar variables each have their own (balanced) tree for searching. Maintenance of these trees is also in the file names.c. The return value of the routine is the type of the variable if it has been located in the tree, and its number is in the variable number. The parameter par tells us if we have to apply automatic declarations if the variable is not encountered as in

```
AutoDeclare Symbol x;
Local F = (x1+...+x10)^2;
```

Names are stored in a linear array, but a balanced tree is built to search in this array. Hence the tree elements have the offsets of the actual names in this array. The tree presents us the names in alphabetical order, while the array allows .store to eliminate the names that were added since the most recent .global in an easy fashion. It also allows the doubling of the size of the array when its size turns out to be insufficient. After the .store the tree is reconstructed from scratch. This is usually more efficient than removing elements one by one while keeping the tree balanced.

To enter algebraic variables in their respective tables one can use the EntVar routine which calls the proper Add routines with the properties as requested. One can look up what those properties are in the code of the EntVar routine, the Add routines and the datastructs in structs.h.

The symbols, vectors and functions have a relic from Schoonschip in the sense that they can be declared complex or imaginary. This was in anticipation of a complex conjugation, which was never built, because eventually it was not seen as very useful. It is rather easy to program such conjugations externally and often one would run into extra features that would have to be programmed anyway as in creating the complex conjugation of an amplitude that contains contracted indices. I have never seen a program that uses this complex feature, but I do not dare to remove it either.

There is one special type of variables. If the compiler encounters an undefined variable, or a variable that is used improperly, it can be given the type 'dubious'. This will of course give a compiler error, but the fact that from now on the type is dubious avoids getting a lengthy accumulation of extra error messages.

Because preprocessor variables and dollar variables are declared when defined the first time, undefined variables of these types are only detected when one tries to use them. With dollar variables there are extra complications in the sense that internally they can represent different types of objects. This will only be detected during use and wrong use is a fatal execution error. It would be rather difficult to have the compiler try to intercept this, because these things may go beyond the boundaries of modules, procedures, switches, if constructions or do-loops.

8 Streams

The way the preprocessor gets its character input is by means of streams. The main stream is of course the input file. But the contents of an include file or of a procedure define other types of streams. The contents of a preprocessor variable or a textually interpreted dollar variable is yet another stream and the substitution of the contents of a #do loop is also a stream. In the case there is communication with external programs the input from such a channel is also a stream. The basic stream files can be found in the Streams fold in the file tools.c. The stream concept exists to allow the GetInput routine in the file pre.c to deal with input in an uniform way and allow closing a stream and falling back to a previous stream when the end of the current stream is reached. The opening of streams is done by the appropriate preprocessor command, which calls the OpenStream routine in tools.c with the correct arguments. The stream of a preprocessor variable is for instance opened when its closing quote character is encountered. The stream is then pointed at the content of the variable. What happens with the characters that are read from the stream depends on what the preprocessor is doing. Example:

```
#define a10 "xx"
#define num "10"
  +y`a`num''+x
```

The preprocessor will read the +y and put that inside the inputbuffer. It then encounters the backquote and start a new buffer in which it then puts the a. Next it encounters another backquote and start yet another buffer into which it reads the characters "num". Then it encounters a quote, indicating that the last buffer is complete and has to be interpreted as a preprocessor variable. Hence it points the input to the stream defined by the content of num which are the characters 1 and 0. These are now added to the previous buffer which, when the 10 is finished contains a10. Next the reading continues in the previous stream and the second quote is encountered. Hence this buffer is now complete, the preprocessor variable a10 is looked up and its stream is started causing the characters x x to be read into the main buffer. Finally, when this stream is finished the program falls back to the main input stream and the main buffer reads now +yxx and continued reading in the main stream makes this into +yxx+x and whatever follows.

The preprocessor routine that has to recognize the preprocessor variables and other constructions like the escape character `\` is GetChar. Because of the interaction of the escape character and linefeeds, it has to remember the previous character. This also holds for the `~` character for delayed substitutions. This routine also has to recognize potential input for the preprocessor calculator which, if it is recognized as such creates a separate stream with the result of the calculation. In addition there is the raising or lowering operation for preprocessor dollar variables with the `'$a++'` or `'$a--'` construction that is taken care of in the GetChar routine.

Similarly the content of a #do-loop is a stream that is repeated as long as needed, each time with the new value of the loop parameter which is of course a preprocessor variable. And the #breakdo command tells basically how many #do streams have to be closed and popped off the stack.

9 Startup and Finish

When Form starts up. it has to make sure that all fixed buffers have been allocated and that it knows where to find files, where to make its temporary files etc. This is mostly done in the files startup.c and setfile.c. The startup sequence is basically:

- Set the termination signal handler. This is needed to allow Form to cleanup its temporary files when the program crashes or terminates on error.
- Initiate threads (TForm and MPI).
- Start the clock.
- Start up the Form internal file handling.
- Define all internal variables.
- Analyse the command tail.
- Determine the setup variables from a possible form.set file.
- Open the input file.
- Look in the environment for setup parameters. Overwrite those from the file.
- Look in the input file for setup parameters. If so, overwrite what was obtained before.

- Determine the proper size of the allocations and make those.
- If we use checkpoints, look whether we are recovering an old run. If so, we make the recovery.
- Reserve the names for the temporary files.
- Start all threads (TFORM).
- Initialize a factorization cache buffer.
- Print header.
- Initialize more variables, specially those that may be changed during runtime.
- Make all current variables global (as if there was a .global at this point).
- Start the timers.
- Call the preprocessor.

One of the important things here is that under a UNIX-like system Form will intercept a number of signals. The most important signal is the signal for a fatal error that makes the program crash. We use this throughout the code to make the program terminate with a call to Terminate (in startup.c). The Terminate routine prints an error message, does some cleanup, prints the final statistics, resets the most important signal handler and exits. The code for the signal handling was implemented by Misha Tentyukov before the introduction of TForm. The introduction of TForm must have caused a subtlety that makes that very occasionally there must be a deadlock during the abortion of a TForm run. Because this is extremely hard to reproduce, to my knowledge it has not been tracked down.

When the program terminates irregularly, it will try to remove all temporary files with the exception of the .str file, provided it is not empty. This allows one to more or less recover data. Of course this may not work when the setting of $-variables is also essential, unless one has made those recoverable as well.

One common problem is that one has a directory for the temporary files in which Form also puts its .str files and because one rarely looks into this directory, eventually there are very many files there. An occasional cleanup of this directory is no luxury, because Form tries for unique file names by trying a fixed sequence. This means that if there are still 1000 .str files Form will have to try 1001 times to see whether a file exists until it runs into a name it can use.

When the buffers are allocated, FORM has to calculate a consistent set of sizes first. An example of an inconsistent set would be to have SmallSize less than MaxTermSize. This would make no sense. This 'recalculation' takes place in the routine RecalcSetups in the file setfile.c. The algorithm there is more or less ad-hoc. Hence if one has a better algorithm, please discuss this in the issues in the github repository. The complete list of all setup parameters can be found in the #includes fold in the file setfile.c and if one would like to see what the actual setups are, one can use the "on setup" statement. One thing to realize is that in TFORM the large buffer is allocated for the master, and a second large buffer is allocated for the combined workers. But in addition the sortbots (see the section on multiple threads) need a large buffer that they will share. If the computer on which TFORM is running has a decent size swap space, this is usually no problem because FORM/TFORM is very swap-friendly, but when there is no swap space, as on most batch computers, this may become a problem when the large buffer needs to be very large.

10 More about the preprocessor

After all initializations have been finished the main routine (in startup.c) calls the preprocessor (PreProcessor). The task of the preprocessor is to read input characters, process them and compose complete instructions or statements. Preprocessor instructions are to be executed immediately. Complete statements are sent to the compiler which then converts these statements into virtual machine code. Once a module has been processed completely and there are no error conditions, the preprocessor calls the routine ExecModule (in module.c) to execute the complete module. Depending on the type of module more routines may be called. In the case of a fatal error or when a .end instruction is encountered, the preprocessor will return, possibly with an error condition.

In the section about streams we have seen already that the input characters can come from a variety of places. It is the preprocessor that determines from which places on the basis of #procedure, #do, #if, #switch, #external constructions or preprocessor variables. If none of such constructions are active the input is from the regular input file. When there is an interactive session the input would be from the stdin. The types of streams are given in the file ftypes.h.

The input is handled by the routines GetChar and GetInput. Because there are various conventions about what constitutes a new line (0x0a = '\n' = LINEFEED, or 0x0D = '\r' = CARRIAGERETURN+0x0A = '\n' = LINEFEED to name the two most popular ones) Form can read either and converts the CARRIAGERETURN+LINEFEED and other conventions into LINEFEED for internal purposes. When the output is written it uses the local convention of the computer on which it is running. At the moment that is either LINEFEED when WITHRETURN is not defined or CARRIAGERETURN+LINEFEED when WITHRETURN is defined. The macro AddLineFeed(s,n) in the file declare.h can do this automatically. It is possible to redefine the LINEFEED and/or CARRIAGERETURN characters in the corresponding .h files in the case that other conventions are encountered.

When the instructions and statements are collected there is a little complication in that one can define an instruction between the various lines of a single statement as in

```
L F = a
#$n = 1;
    + b
    +c;
```

or more commonly:

```
L F =
#do i = 1,10
    +a‘i’
#enddo
    ;
```

This means that there are two buffers to collect the input: the buffer in AP.preStart and the buffer in AC.iBuffer. This last buffer has a double use. When a #assign statement is read, it puts its characters after the current position and the preprocessor assignment of a dollar variable takes place from there. When a dollar variable is given a value in the preprocessor the preprocessor generates internally a #assign instruction to deal with this case. This instruction is not for external use.

Once the # character is encountered the remaining part of the instruction is read by the PreProInstruction routine which makes use of the LoadInstruction routine. It knows various modes as explained in the commentary. When the instruction is complete, it is executed. The routine reads in principle only until the end-of-line, unless it has been escaped with a \ character, or we have the assignment of a dollar variable. In the last case it reads until the ; character:

```
    Symbols a,b,c;
    L   F =
        +a
    #$x =
            2*a
            +3*b;
        +b
        +c;
    Print +f +s;
    .sort
   F =
       + c
       + b
```

```
        + a
       ;
    #write "<<%$>>",$x
<<3*b+2*a>>
    .end
```

Once the reading of the #$ instruction is complete, this instruction is executed immediately. This means that the dollar variable is defined/looked up and the rhs is compiled and executed. The result of the running is put inside the dollar administration. When we have the #inside construction the statements inside are executed as if they are a complete module. This means that this 'module' must be read and compiled and afterwards executed. This requires a certain amount of recursions and a stack of storing partial compilation information. The execution part resides of course in the DoEndInside routine. Currently #inside constructions cannot be nested, although it may be possible to make it that way. Much of the necessary infrastructure exists already.

The LoadStatement routine has to compose a complete statement. One of the problems here is of course that we have no idea how big the input buffer AC.iBuffer should be. Because of this we allow dynamic expansion when needed with the DoubleLList routine (in tools.c) which can double the size of buffers. The condition for this to work is that we have no pointers pointing to positions in the old buffer that will be released. If there are any they have to be corrected immediately. Other positions should be remembered by offsets. This input buffer is also used for the reading of dollar variables that are stored after the current input and the current input position has to be placed on a stack. For this we have the array AP.PreAssignStack and the variable AP.PreAssignLevel. This way we can use the assignment of dollars in the preprocessor in a recursive way:

```
 Symbols a,b;
 L   F =
     +a
     #$t1 =
         +2*b
         #$t2 =
             +3*a
             +3*b
             ;
         +2*a;
     +b
     ;
 Print;
 .sort
F =
   b + a;
 #write <> "   $t1 = %$",$t1;
$t1 = 2*b+2*a
 #write <> "   $t2 = %$",$t2;
$t2 = 3*b+3*a
 .end
```

One problem in the reading of the statements is the conversion of blank spaces. Next to an operator like + - * / or a comma they are eaten and between variables they are converted into a comma. The variable AP.eat is meant to keep track of whether we may have to eat the previous character because of this.

The preprocessor calculator is a rather primitive calculator. On the other hand, we do not want it to be too sophisticated, because we also need the curly brackets for the sets in the algebra part of FORM. The #if constructions have relatively primitive conditions. It could be possible to improve this in the future. I do not think this will break any existing code. The code for the dictionaries is also rather simple and we can skip that here. Some code exists already for namespaces. Again, we are not going to treat that here because it is neither complete nor active.

An important routine in the preprocessor is PutPreVar. It is used to (re)define preprocessor variables. In the routine StartVariables in the file startup.c one can see which variables are the system defined variables. Some of those variables need special treatment in the routine GetChar, because their value is not a constant. An example would be the preprocessor function/macro tolower_ which converts the string in the argument to lower case as in

```
#write "    say no to ‘tolower_(CamelCase)’"
say no to camelcase
.end
```

A number of the systems defined preprocessor constants concerns the generation of diagrams. This makes things easier to select the proper type of diagrams by adding the various options in the preprocessor calculator.

11 Exercises part 1

The exercises are just suggestions for practising and making yourself familiar with the FORM sources. Just take one or a few of them and see how far you get.

1. Put in your own random number generator. Put in a method to select this other generator.
2. FORMTMP_ ???? Looks in -t, then in environment for FORMTMP
3. Investigate what would be needed to allow #inside,#endinside constructions to be nested.
4. What would be needed to remove preprocessor variables from the system? Remark: what about when this is done from procedures, do loops etc.? Is there an easy method?
5. Is it possible to change the number of workers from either the setup file, the input file or the environment?
6. Can you build a #continuedo to skip to the end of a #do-loop?
7. What needs to be done to create a #return inside a procedure?
8. try to find where the macro's LINEFEED and CARRIAGERETURN are used. What would you have to do if the system that is used is not suitable for your work (because the systems library adds already things)?