

About Groups32

After the class demonstration of the Groups32 program, several students asked for some information about it.

First of all, Groups32 contains, internally, a set of tables for the groups of orders 1-32. All of the information generated by issuing commands is computed from the tables. Groups32 contains a set of commands, at various levels, which operate on group tables – the ones presented to the user are the “top level”. There are foundational commands which are used to build the top level commands.

Groups32 is extensible. It is possible to add new commands and very easy to extend the user interface to include them. (In algebra classes there has never been enough time to talk about how the system is constructed and to program. I have, however, produced several packages of custom commands which have been added for certain classes.)

History:

I started writing the program around 1990 when Kenneth Almquist posted a file containing tables of all groups of order 1-16. The tables had been generated by a computer program, apparently as a programming exercise (Almquist did not seem to have a background in group theory). He did not discuss the algorithm he was using, but I assume it used backtracking as we discussed in class. Readers of the original posting pointed out that he gave too many tables of some of the orders. In a subsequent posting, he corrected the errors (saying that his isomorphism routine had failed to detect isomorphisms of some of the tables). I became curious about what one would need to know about groups to be able to detect the duplications in the original tables. This led to a small preliminary program to extract information from the group tables.

My research area is “computer algebra”. This field is concerned with the task of making computers do symbolic mathematics. I am interested in the mathematics behind computer algorithms and also in using the computer as a tool for doing mathematics. My particular interest is abstract algebra and related fields.

The system which eventually became Groups32 was built in the same way as a research system. It serves as a good illustration of my ideas for building software systems, particularly because its subject matter is familiar to all mathematicians. Some mathematicians who saw an early version of the program felt that it also had merit as an instructional program – and encouraged me to make it easier to use. In 1995, several factors came together to make it possible to extend the original program to groups of order up to 32. I also developed and tested several user interfaces. The current “commands completion” interface proved to be easiest and quickest to use. It also has the advantage that it is very easy to add new commands to the interface.

Language:

Conventional computer languages were not designed for abstract algebra. The concepts of the mathematical area are too far from the means of expression afforded by the language. Conventional languages are also more suited to the creation of static systems. In some parts of mathematics “the program” is a fixed entity through which one passes varying data. In algebra, an interactive environment is essential. Modifying and extending the software is a typical part of use.

For this reason, Groups32 (and my research work) is based upon use of a non-conventional language, Forth. This language, itself, provides an interactive environment. Data is persistent: it does not disappear after a command is executed. It is possible to have a session in which commands are issued from the keyboard, and further action is based on the observed results. New commands can be added to extend the system during a session (and, if desired, can be saved for future use). Most new commands are defined in terms of existing commands. It is also possible to define commands in terms of assembly language to increase speed. Most of Forth is written in Forth. The tasks of a conventional compiler are distributed to words in the language. Thus even control structure words like IF and THEN are defined within the language. This leaves open the possibility of defining custom control-flow words. Forth is a language for implementing languages.

Forth is the invention of Charles Moore in the late 1960's. Forth was very popular in the early days of microcomputers. It was one of the few high level languages which could be supported by small machines. The Forth Interest Group produced compatible implementations of the language for all microcomputers popular at the time – so Forth was perhaps the most portable language available through the 1980's.

Developing Groups32:

To give some flavor of how Groups32 was developed, here are some code examples. Some things about Forth need to be understood:

1. What I have called “commands” are called “words” in Forth. The name of a word is any collection of printable characters delimited by spaces. Each word has a name and an associated action. Forth has a dictionary of all the words it currently understands (together with a description of their action).
2. Programming is equivalent to extending the dictionary.
3. Forth is stack-based. Most words communicate by taking their arguments off the stack, performing their actions, and putting their results on the stack. As a result, most Forth words can be documented by describing the before/after effect on the stack.

To reduce “magic numbers” several important constants are given names. All the tables are the same size, 32 x 32.

```
32          CONSTANT MaxOrd
MaxOrd DUP * CONSTANT Table_Size
150         CONSTANT MaxTables
```

\ Storage area for the group data

```
CREATE GroupData Table_Size MaxTables * ALLOT
\ multiplication tables

CREATE Idx        MaxTables ALLOT
\ Group orders
```

All group operations are performed on the “current group” stored in a variable Grp:

```
VARIABLE Grp          \ These are set for each group by >Group
```

To speed up computation, the groups elements are stored as indices. 0 is the identity (and it will be printed as “A”), 1 is the next element (which will be printed as “B”) etc. Here is the implementation for group multiplication which is actually used in the Windows version of the program (it is in assembly language for the 40846 but it is particular for Win32Forth). Notice that direct assembly language coding of commands is only used for the most labor intensive and frequently used commands. Only 9 words, in the current version of Groups32, are defined using assembly language – but this gives a 5x increase in speed for some of the more computation-intensive commands.

```

CODE G* ( i j -- i*j )          \ *** WE ASSUME MaxOrd=32 ***
    SHL EBX, 5                  \ Multiply j by 32
    POP EAX ADD EBX, EAX        \ Add i
    MOV EAX, Grp [EDI]         \ Add group offset
    ADD EBX, EAX
    SUB EAX, EAX                \ Find byte at this address
    MOV AL, [EBX] [EDI]
    MOV EBX, EAX
NEXT END-CODE

```

The high level (Forth) code for G* is:

```

: G* ( i j - i*j )
    MaxOrd * + Grp @ + C@ ;

```

So a group is stored in a 32 x 32 table which is stretched linearly by rows into a 1024 chunk of bytes. The groups are stored consecutively – starting with group 0. The variable Grp contains the address of the start of the selected group. Either version of the code takes two indices and returns the byte representing the product.

Internally the group elements are 0,1,... We convert the internal representations to the external A,B,C... whenever printout is required.

```

CHAR A CONSTANT ID          \ ascii code for printing identity
: .Ele ( ele -- ) ID + EMIT SPACE ;

```

This has the interesting effect of making the group elements of a group of order 32 print out as:

```

ABCDEFGHIJKLMNOPQRSTUVWXYZ[ ]^_`

```

It is easy to change the symbols printed without changing the internal representation of elements. One could, for example, use .Ele (the word which prints an element) so that the group elements become:

```

ABCDEFGHIJKLMNOPQRSTUVWXYZabcdef   or
ABCDEFGHIJKLMNOPQRSTUVWXYZ123456   or
123456ABCDEFGHIJKLMNOPQRSTUVWXYZ

```

It turned out, in practice, that it was convenient to have the group elements be “case independent” – so that either “b” or “B” would be taken to be the second element of the group. The “funny symbols” only appear in groups of order 27 or above – and students hardly ever seemed to play with groups that big.

Here is an indication of how lower level commands are used to build commands on the next level. One of the commands, ORDERS, computes the order of each element in a group and then prints the results – listing the elements by order. The orders of the elements are calculated by CALC-ORDERS. The results are printed out by PRT-ORDERS.

```

: EleOrder ( ele -- ord )
  0 For-All-Elements
  DO OVER G* DUP 0=
    IF 2DROP I 1+ LEAVE THEN
  LOOP ;

CREATE OTable MaxOrd 1+ CELLS ALLOT \ order for each element
CREATE Ocnt   MaxOrd 1+ CELLS ALLOT \ count for each order

: 'ORD ( ele -- addr ) CELLS OTable + ;
: 'Ocnt ( i -- addr ) CELLS Ocnt + ;
: +Ocnt ( ord -- ) 'Ocnt 1 SWAP +! ;
: Ord! ( ord ele -- ) 'ORD ! ;
: OClear OTable MaxOrd 1+ CELLS ERASE
      Ocnt MaxOrd 1+ CELLS ERASE ;

: Calc-Orders OClear
  For-All-Elements DO
    I EleOrder \ compute order of element i
    DUP +Ocnt \ update count for that order
    I Ord! \ save the order
  LOOP ;

: Prt-Orders Gord 1+ 1
  DO Gord I MOD 0=
    IF ( i divides order of G )
      3 SPACES I 'Ocnt @ 2 .R
      ." elements of order " I 2 .R ." : "
      For-All-Elements DO I 'ORD @ J =
        IF I .Ele THEN
          LOOP CR
        THEN LOOP ;

: Orders ( grp# -- ) CR >Group
  ." Group number " Gnum .
  ." of Order " Gord . CR
  Calc-Orders Prt-Orders ;

```

What this looks like in practice is:

```

8 orders
Group number 8 of Order 6
  1 elements of order 1:  A
  3 elements of order 2:  D E F
  2 elements of order 3:  B C
  0 elements of order 6:

```

Calc-Orders runs through the group calculating the orders of the elements. For each element it saves the order – and it also updates a count of how many times that order has occurred. [Generally, Groups32 has been coded to emphasize obviousness].

Most words at this level follow the Forth convention of removing their parameters from the stack. The word “Orders” removes the group number from the stack. It does not leave anything on the stack as a result (it produces a printout). Orders is factored into two separate words: Calc-Orders and Prt-Orders since the first of these may be useful independently. [Prt-Orders uses information in particular arrays rather than being passed through the stack – but this word is not intended to be used independently.] Suppose, for example, we wish to print a list of groups and the orders of elements that looks like this:

```

1  1  1
2  2  1  1
3  3  1  2
4  4  1  1  2
5  4  1  3  0
6  5  1  4
7  6  1  1  2  2
8  6  1  3  2  0
9  7  1  6
10 8  1  1  2  4
11 8  1  3  4  0
12 8  1  7  0  0
13 8  1  5  2  0
14 8  1  1  6  0
15 9  1  2  6
16 9  1  8  0

```

Where we have listed the group number, the order of the group, and then the number of elements of each order (for each divisor of the group order). A table like this might be used to scan the entire collection of groups to answer some questions about group orders (like whether two groups having the same number of elements of each order are isomorphic).

This custom table is obtained by

```

: New-Prt-Orders  Gord 1+ 1
  DO  Gord I MOD 0=  \ I is a divisor of order(G)
    IF  I 'Ocnt @ 3 .R
    THEN LOOP ;

: New-Orders  ( grp# -- )  CR >Group
  Gnum 4 .R
  Gord 4 .R
  Calc-Orders  New-Prt-Orders  ;

: All-Orders
  For-All-Groups DO  I New-Orders  Loop ;

```

We have produced a new printout routine but used the existing word which calculates orders.

Extensible User Interface:

To use the system at the level described above requires that the user knows the name of every important command and what parameters are needed for it to act. Groups32 was made useful to students by equipping it with an interface that gives a list of commands and prompts for any additional information.

For speed, the interface uses “command completion”. The names of the commands available to the user are stored. As the user types, the system checks if what has been typed so far matches a stored command. There are three possibilities:

1. The input matches the starting letters of more than one command.
2. The input matches the starting letters of exactly one command.
3. The input does not match the starting letters of any command.

In the first case, the system waits for the user to type more letters. In the second case, the system prompts for any input needed and carries out the command. In the third case, the offending last letter is removed and the system beeps.

One interesting feature of this interface is the ease with which new commands can be added. Suppose we want to add the command New-Order (see above) to the menu. We need to prompt for the input of a group number, and we will add a help comment. So we define a new auxiliary word %New-Order:

```
: %New-Order
  Help:
    This prints a condensed list of orders of elements of the
    given group. For group 33 we get this output:
        33 16 1 15 0 0 0
    This shows that group 33 has order 16. The divisors of 16
    are 1,2,4,8 and 16. There is 1 element of order 1,
    15 elements of order 2, and 0 elements of orders 4, 8, 16.
  Help;
  Get-Grp New-Order ;
```

This new command is installed by

```
>CMD New-Order %New-Order
```