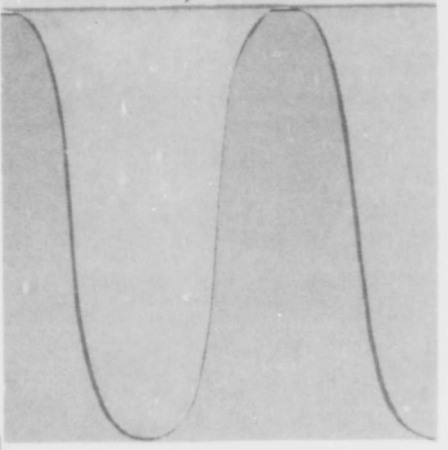
THE UNIVERSITY
OF WISCONSIN
madison, wisconsin



A COMPUTER PROGRAM FOR INTERACTIVE MANIPULATION OF FINITE GROUP PRESENTATIONS

F. D. Crary

MRC Technical Summary Report # 1045
February 1970

UNITED STATES ARMY

MATHEMATICS RESEARCH CENTER

THE UNIVERSITY OF WISCONSIN
MATHEMATICS RESEARCH CENTER

Contract No.: DA-31-124-ARO-D-462

A COMPUTER PROGRAM FOR INTERACTIVE MANIPULATION OF FINITE GROUP PRESENTATIONS

F. D. Crary

MRC Technical Summary Report # 1045
February 1970

Madison, Wisconsin 53706

ABSTRACT

An interactive computer program for manipulating finite group presentations is described. A brief mathematical background is provided along with operating instructions and notes on the programming. An illustrative example is included.

CONTENTS

TABLES

APPENDICES

A COMPUTER PROGRAM FOR INTERACTIVE MANIPULATION OF FINITE GROUP PRESENTATIONS

F. D. Crary

Chapter I. Introduction

This report describes an interactive computer program which reduces the amount of clerical labor involved in the manipulation of finite group presentations (Chapter II contains definitions of this and other terms). From an overall point of view, this program makes the computer an automatic scratch pad for working on problems of this type. The program, called GROUP, is written for the Burroughs B-5500 computer and is operated by a user at a remote teletype.

In this report, we give a brief sketch of the mathematical background of the program, instructions concerning the operation of the program, and some notes on the programming structure. The user who is familiar with the topic of group presentations and Tietze Equivalences may choose to omit Chapter II and refer to it for definitions as they are used in this report.

Directions for Further Development: This program can be seen as a first step in treating problems dealing with group presentations. During the writing, debugging, use and documentation of the program, several possibilities for further work have been seen. Perhaps the greatest need is for the capability of working within some given structure of the relations. This would allow one to work on the general case of a problem rather than being limited to special cases. This is especially true since the current program may use

Sponsored by the United States Army under Contract No. DA-31-124-ARO-D-462.

peculiarities in the structure of a special case to perform operations that may not be applicable to the general case. A second possible direction for development would be to make the program itself programmable. This would, in effect, create a stored program mode in addition to what is now, in essence, a desk calculator mode. Among the other benefits of such an addition would be the possibility of the user defining new operations in terms of those already available without the necessity of instructing the program in their details each time that they are used.

Disclaimer: This program has been tested and is believed to be correct. It is certain, however, that not all paths through the program have been tested under all conditions. Therefore, neither the programmer, the Mathematics Research Center, United States Army, nor the University of Wisconsin assumes any responsibility for any errors, omissions, malfunctions, or difficulties which may arise in its use.

Chapter II. Mathematical Background

In this chapter, we sketch the group theory necessary for describing the operation of the program. Proofs of assertions are almost entirely omitted. More precise definitions and missing proofs can be found in [1] and [3]. See especially [1] for a topological application. Those who are familiar with the notions of group presentations and Tietze equivalences may omit this chapter.

Free Groups: Suppose that $\underline{A}$ is a set. We wish to construct a group $\mathfrak{F}(\underline{A})$ which contains $\underline{A}$ in such a manner that the elements of $\underline{A}$ are as independent as possible and such that $\mathfrak{F}(\underline{A})$ is as "small" as possible. By this we shall mean that $\mathfrak{F}(\underline{A})$ must satisfy the

Mapping Property: If $\mathfrak{Q}$ is a group and if $f: \underline{A} \to \mathfrak{Q}$ is any function, then there is a unique homomorphism $h: \mathfrak{F}(\underline{A}) \to \mathfrak{Q}$ which extends f. Such a group $\mathfrak{F}(\underline{A})$ is called the free group with generating set $\underline{A}$. The mapping property can serve as the definition of the free group, however one is then left with the problem of showing the existence of such a group $\mathfrak{F}(\underline{A})$ for any set $\underline{A}$. Since we are concerned in the sequel with a means of representing the free group over $\underline{A}$, we give a construction for $\mathfrak{F}(\underline{A})$ below which will be used later in discussing free groups.

The set $\underline{A}$ is called the alphabet of $\mathfrak{F}(\underline{A})$; and its members, letters. A syllable will be a symbol of the form $\underline{a}^n$ where $\underline{a}$ is a letter and n is an integer. A word is a finite ordered sequence of syllables; e. g.,

$$\underline{a}^1 \underline{b}^{-3} \underline{a}^0 \underline{c}^4 \underline{c}^{-2} \underline{b}^3 .$$

The unique word with no syllables is called the empty word and is written $\underline{1}$.

The product $\underline{w}\,\underline{w}'$ of two words $\underline{w}$ and $\underline{w}'$ is formed by writing first the word $\underline{w}$ and then the word $\underline{w}'$. If $\mathfrak{W}$ is the collection of all words, it is clear that, under the operation described, $\mathfrak{W}$ is not a group. We now show how to obtain a group from $\mathfrak{W}$.

We regard two words in $\mathfrak{W}$ as equivalent if one can be obtained from the other by a finite sequence of contractions or inverses of contractions. A contraction is either (a) the elimination of a syllable of the form $\underline{a}^0$, or (b) the replacement of $\underline{a}^p\,\underline{a}^q$ with $\underline{a}^{p+q}$ (where $\underline{a}$ is any letter). Then $\mathfrak{F}(\underline{A})$ is defined to be the group obtained from $\mathfrak{W}$ by imposing the equivalence relation described above. Each letter $\underline{a}$ is identified with the equivalence class containing the word $\underline{a}^1$. The identity element of $\mathfrak{F}(\underline{A})$ is the class containing $\underline{1}$. There is, of course, something left to be proved in order to show that $\mathfrak{F}(\underline{A})$ is actually a group and even that it is well-defined. Proofs of the necessary facts can be found in [1] and [3]. It is, however, not difficult to see that the constructed group satisfies the mapping property.

We adopt the notation of writing the syllable $\underline{a}^1$ as $\underline{a}$. We also write each word after all possible contractions have been performed. It can be shown that performing all possible contractions on different members of an equivalence class always yields the same result. We slightly abuse the terminology introduced above by calling the equivalences classes of $\mathfrak{F}(\underline{A})$ words.

Presentations: The notion of a presentation of a group depends on the following

Theorem: Any group is a homomorphic image of some free group.

The proof of this theorem is very easy: If $\mathfrak{G}$ is a group, then $\mathfrak{G}$ is the image

of $\mathfrak{F}(\mathfrak{G})$ under the extension of the identity function on $\mathfrak{G}$ which is given by the mapping property. This theorem says that for any group $\mathfrak{G}$, there is a free group $\mathfrak{F}(\underline{A})$ and a normal subgroup $\underline{n}$ of $\mathfrak{F}(\underline{A})$ such that $\mathfrak{G}$ is isomorphic to the quotient group $\mathfrak{F}(\underline{A})/\underline{n}$.

We pause here to introduce another definition. If $\mathfrak{G}$ is a group and $\underline{S}$ is a subset of $\mathfrak{G}$, then the consequence $C_{\mathfrak{G}}(\underline{S})$ of $\underline{S}$ in $\mathfrak{G}$ is the smallest normal subgroup of $\mathfrak{G}$ which contains $\underline{S}$. Equivalently, the consequence is the intersection of all normal subgroups of $\mathfrak{G}$ which contains $\underline{S}$. If there is no ambiguity as to the identity of $\mathfrak{G}$, we will write the consequence as $C(\underline{S})$.

A presentation of a group $\mathfrak{G}$ is defined to be a pair $(\underline{A} : \underline{R})$, where $\underline{A}$ is a set, $\underline{R}$ is a subset of $\mathfrak{F}(\underline{A})$ and $\mathfrak{G}$ is isomorphic to $\mathfrak{F}(\underline{A})/C(\underline{R})$. We write $\mathfrak{F}(\underline{A})/C(\underline{R})$ as $|\underline{A} : \underline{R}|$. Clearly, any group has a large number of presentations (infinitely many, in fact). A natural question to ask is how to tell if two given presentations present the same group. It has been shown ([4]) that this question, as well as many related ones, is recursively unsolvable. Therefore the program which this report describes makes no claim to answering this question.

A presentation $(\underline{A} : \underline{R})$ is said to be finitely generated if $\underline{A}$ is finite, finitely related if $\underline{R}$ is finite, and finite if it is both finitely generated and finitely related. A group is said to be finitely generated if it has a finitely generated presentation, finitely related if it has a finitely related presentation, and finitely presented if it has a finite presentation. Suppose that $(\underline{A} : \underline{R})$ is a finite presentation,

$$\underline{A} = \{\underline{a}_1, \underline{a}_2, \ldots, \underline{a}_m\}, \text{ and}$$

$$\underline{R} = \{\underline{w}_1, \underline{w}_2, \ldots, \underline{w}_n\}.$$

Then we also write (A: R) as

$$(\underline{a}_1, \underline{a}_2, \ldots, \underline{a}_m : \underline{w}_1, \underline{w}_2, \ldots, \underline{w}_n).$$

The elements of $\underline{A}$ are called generators; and the elements of $\underline{R}$, relators.

It is often more convenient when working with presentations to consider relations instead of relators. An equation of the form $\underline{u} = \underline{v}$ is a relation if $\underline{u}\,\underline{v}^{-1} \in C(\underline{R})$. Conversely, each relator $\underline{w}$ has the corresponding relation $\underline{w} = \underline{1}$. It will be seen that each relator may be considered as a relation and further that defining relations (those which correspond to relators) may be manipulated as equations (with due regard to the fact that a free group is not abelian). Relations will be used in place of relators whenever convenient.

Tietze Equivalences: We now describe two important types of operations on group presentations, the Tietze Equivalences. These operations are important because of the Tietze Theorem which states that (for finite presentations) two presentations present isomorphic groups if and only if there is a finite sequence of Tietze equivalences taking one to the other.

A Tietze equivalence of type I consists of adding a relation which is in the consequence of the original relations:

$$\text{I: } (\underline{A} : \underline{R}) \rightarrow (\underline{A} : \underline{R}, \underline{w}), \text{ where } \underline{w} \in C(\underline{R}).$$

A Tietze equivalence of type I' is the inverse of a type I equivalence; i. e., removing a relation which is in the consequence of the remaining relations.

A Tietze equivalence of type II consists of adding a generator

and a relation in the following manner: Let $(\underline{A} : \underline{R})$ be a presentation, let $\underline{x}$ be a symbol not in $\underline{A}$, and let $\underline{w} \in \mathfrak{F}(A)$. Then a type II equivalence is

$$\text{II} : (\underline{A} : \underline{R}) \rightarrow (\underline{A}, \underline{x} : \underline{R}, \underline{x} = \underline{w}) .$$

We may view a type II equivalence as defining a new generator as a word in the free group generated by the original generators. A type II' equivalence is the inverse of a type II equivalence.

It is not difficult to see that the Tietze equivalences induce isomorphisms on the groups presented. Thus in order to check that an operation induces an isomorphism, it suffices to show that it is a finite composition of Tietze equivalences. While it seems that these equivalences are nearly trivial, it is not actually so. Deciding whether a word is in the consequence of some given set of words is a special case of the word problem, and is recursively unsolvable ([4]).

Chapter III. Program Operation

The GROUP program is designed to accept an instruction from a user at a teletype and execute it immediately in response to the information contained in the instruction, the presentation currently in storage, and the status of previously set switches. The program's input is free-format as much as possible (see section 2 for a brief description of the input system). An attempt has been made to have the program accept full sentences whenever possible. Examples are given to show both brief and long forms of the instructions.

Output from the program is obtained both from the teletype and from the line printer. To keep the teletype from becoming clogged with output, only summary information and final results of instructions are typed. Details and intermediate results are relegated to the printer for later use. Intermediate presentations are always available at the teletype via the disk save files and the Read Presentation Instruction (see sections 1 and 4) .

This chapter includes general concepts of the operation of the program (section 1), a description of the free-format input system (section 2), definitions of the general input components (section 3), and instructions for program use (section 4).

1. General Concepts

Representations of Words and Presentations: When printing or typing words or presentations, the program follows standard mathematical notation as closely as the limitations of the machine permit. In particular, exponents of 1 are always omitted and the trivial word (word with no syllables)

is typed and printed as $\underline{1}$. Exponents have been raised on the printed output to make them superscripts (see examples below), however this was not feasible on the teletype. Hence each exponent is typed after its base with an intervening space. Since syllables with exponent zero are eliminated and numbers are not valid generators, there is no confusion as to what is a base and what is an exponent.

Presentations are typed and printed in the same format except that relations differ between the output devices as described in the paragraph above. First the name of the presentation (discussed later in this section) is given; second, the table of generators; and third, the list of relations. Each relation in the presentation is numbered consecutively, beginning with 1. As an example, a standard presentation for the group of a trefoil knot is

$$(\underline{a}, \underline{b}, \underline{c} : \underline{a}\,\underline{c}\,\underline{a}^{-1}\,\underline{b}^{-1}, \underline{b}\,\underline{a}\,\underline{b}^{-1}\,\underline{c}^{-1}, \underline{c}\,\underline{b}\,\underline{c}^{-1}\,\underline{a}^{-1}).$$

The program's teletype representation of the same presentation is

```
PRESENTATION GROUP/DEMO-2
GENERATORS  A, B, C
RELATION
    1.  A C A -1 B -1
    2.  B A B -1 C -1
    3.  C B C -1 A -1
```

On the line printer, this appears as

```
PRESENTATION GROUP/DEMO=2
GENERATORS  A, B, C
RELATION

1.  A C A**-1 B**-1

2.  B A B**-1 C**-1

3.  C B C**-1 A**-1
```

In this document, the teletype form of output will usually be used because it corresponds more closely with the program's input format.

Storage of Relations: The relations of a presentation are stored by the program as relators. Since a conjugate of a relator is in the consequence of the set of relators, replacement of a relator by one of its conjugates may be accomplished by two Tietze equivalences. Hence the operation preserves the group which is presented. A corollary of this observation is that relators may be considered as circular words.

The program takes advantage of this corollary in two ways: (1) In addition to making reductions of the form

$$\underline{u}\,\underline{a}^{p}\,\underline{a}^{q}\,\underline{v} \rightarrow \underline{u}\,\underline{a}^{p+q}\,\underline{v},$$

the program also makes reductions of the form

$$\underline{a}^{p}\,\underline{w}\,\underline{a}^{q} \rightarrow \underline{a}^{p+q}\,\underline{w}.$$

These reductions are made automatically by the program. The operation of performing them is called circular reduction. (2) The relators are shifted circularly whenever it is convenient to do so. The user of the program may

cause a relation to be circle-shifted by using the Shift Instruction (see section 4).

Disk Storage: The program is provided with semi-permanent storage of presentations and other information by means of magnetic disk storage. This storage is provided for three reasons: (1) It is often necessary or desirable to return to a previous point in a series of transformations and try an action different than that performed previously. (2) It is often useful to determine the effect of a series of transformations upon a word that is not a relator and a convenient means of repeating one's steps is then desirable. (3) On those occasions when the computer operating system "dies", it is desirable to have a quick means of recovery.

Objectives (1) and (3) are accomplished in the GROUP program by having the program store each presentation as it is obtained. Objective (2) and its implementation are discussed in the next part (Chaining).

The information stored on the magnetic disk (hereafter referred to as the "disk") is organized into files. Files on the B-5500 have names consisting of a prefix and suffix, and are usually written in the form

prefix/suffix .

For the purposes of this program, the prefix and suffix are restricted to a maximum of six legal non-blank characters not including "-" and "/" .

The files that are used for the program's storage are referred to as disk save files. At the beginning of each program run there is no disk save file established for the results of that run. There may exist previously created disk save files, but such files may not be used by this run to store additional

information. As soon as the program has some information to store, it will create a new disk save file. At any time, the user may force the program to create a new disk save file via the New Disk File Instruction (see section 4). If a disk save file becomes full, additional information to be stored will cause the program to create a new disk save file. Any disk save file in which the program is storing information is closed upon the creation of a new disk save file and may not be reopened to store additional information.

The naming of disk save files is controlled by the Disk File Name Instruction. This instruction selects which of two naming conventions is to be used when the next disk save file of a run is created. The convention selected is reflected in the global switch STANDARDNAMES. The naming conventions are:

Standard naming convention: This convention causes the program to create a disk save file named

GROUP/xyyddd

where x is a letter of the alphabet and yyddd is the current date expressed as the last two digits of the year (yy) followed by the number of the day in the year (ddd). The letter x is chosen to be the earliest letter in the alphabet that does not cause duplication of file names. The standard naming convention is used by the program unless changed through the Disk File Name Instruction.

Nonstandard naming convention: This convention causes the program to ask the user to supply a name for the disk save file to be created. The program will type "PLEASE ENTER A DISK SAVE FILE NAME". The user then has three choices: (a) he may enter a file name of the form prefix/suffix ,

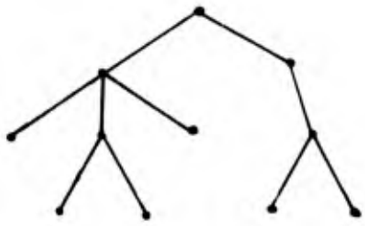
(b) he may enter a file name in the form suffix in which case the prefix "GROUP" is assumed, or (c) he may type "↵" in which case the program selects the standard naming convention for this and all subsequent files to be created unless changed by the Disk File Name Instruction or as indicated below. If a name is entered under choices (a) or (b) which duplicates the name of a file already in existence, the program will repeat the request for a file name. The nonstandard convention is used either if it has been selected through the Disk File Name Instruction or if all names tried under the standard convention duplicate names already in use.

After a disk save file is created, the program types a message giving the name of the file. For example (nonstandard convention selected):

```
PLEASE ENTER A DISK SAVE FILE NAME DEMO↵
DISK SAVE FILE IS GROUP/DEMO
```

The underlined portion of the example was typed by the user.

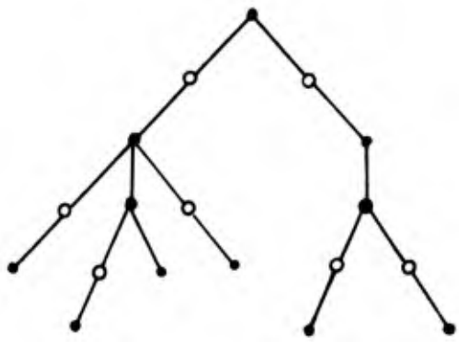
Chaining: To be able to determine the effect of a series of transformations upon a word that is not a relation (objective 2 of disk storage), it is necessary for the program to be able to remember the transformations which led from one presentation to another. Since many different transformations may be performed on any presentation by returning to it again and again, the general structure of the relationships among the presentations is a tree (since the program considers two copies of the same presentation different if they arise from a different source or by different means).



● represents a presentation

Thus it is necessary that this tree structure be remembered by the program.

The program must also remember the transformation that obtained one presentation from its predecessor. To accomplish this, additional nodes which give the transformation are introduced into the tree (represented by o in the diagram):



Each node in the tree is represented by an entry in the disk save file. Notice that not all transformations generate a node since certain operations do not change any essential feature of the presentation (e. g., inverting a relation) or act in a manner that is not uniform on the presentation (e. g., editing a relation). Those instructions (see section 4) that always generate a transformation entry are the Solve Instruction, Simplify Instruction, and Define Instruction. Those instructions that sometimes generate a transformation entry are the Remove Instruction, Replace Instruction, Add Instruction and Abelianize Instruction. The description of each instruction specifies those

conditions that result in the generation of a transformation entry (see section 4).

In the tree is stored a latent set of instructions telling how to get from the root of the tree to any presentation in one of the branches. In order to use these latent instructions, the program must begin at the presentation in the branch and follow the tree backward to the root and map the path that it follows. The resulting set of instructions is called a chain, and the process of creating it is called building the chain. Once a chain has been built, it is retained by the program until a different chain is needed or until the user instructs the program to discard the chain (see Trace Instruction). By using the Build Chain Instruction, any presentation preceeding the end presentation may be considered as the root of the tree.

Presentation Names: Each presentation entered into the program or generated by it is given a name by which it may be referred (see, for example, the Read Presentation Instruction). The complete name is of the form

prefix / suffix - number,

where prefix / suffix is the name of the disk save file in which the presentation is stored, and number is the number of the entry in that disk save file. The presentation name is assigned automatically by the program.

The program will always refer to a presentation by its complete name, but two abbreviated forms are available to the user. These are

suffix - number , and

number .

In the first case, a prefix of "GROUP" is assumed, and in the second case, the program assumes that the disk save file meant is that in which the program is currently storing information (if there is such a file). The name of the current disk save file being used for storage is available via the Disk Summary Instruction in case of doubt. If anything other than a positive integer is used in the number position, the name is assumed to refer to the last presentation in the disk save file; for example, GROUP/DEMO-LAST.

Subword (definition): Suppose that $\underline{w}_1$ and $\underline{w}_2$ are words in some free group. Then $\underline{w}_1$ is said to be a subword of $\underline{w}_2$ if the expanded form of $\underline{w}_1$ is a subset of the expanded form of $\underline{w}_2$, each being considered as an ordered set of syllables. The expanded form of a word is found by rewriting the word as a reduced word so that each syllable has as exponent either +1 or -1.

In the case where $\underline{w}_2$ is considered as a circular word (e.g., a relator), $\underline{w}_1$ is a subword of $\underline{w}_2$ if it is a subword (in the first sense) of some circle shifting of the expanded form of $\underline{w}_2$.

Containment Operation: The containment operation is performed only if the CONTAINSWITCH is set. This switch is initially set off because the operation can be extremely time-consuming. The value of this switch is controlled by the Contain Switch Instruction (see section 4). If the switch is set off, an attempt to perform a containment operation is said to fail.

In general, this operation attempts to shorten relations by replacing major parts of relations with shorter equivalents found from other relations. For example, if relation 1 has the form $\underline{w}_1 \underline{w}_2 = \underline{1}$ where $\underline{w}_1$ is

longer than $\underline{w}_2$ (in their expanded forms), and relation 2 has the form $\underline{w}_1 \underline{w}_3 = \underline{1}$, then containment will change relation 2 to $\underline{w}_2^{-1} \underline{w}_3 = \underline{1}$, a shorter relation. If there is only one relation in the presentation, then clearly no containment is possible. It is not difficult to see that the containment operation can be realized as a sequence of Tietze equivalences.

For the purposes of this operation, the length of a word is the number of syllables appearing in its expanded form. For ease in describing the operation, it should be assumed that all words are written in the expanded form (in general, the program does not store words in that form). We will now describe the containment operation in more detail.

Write relation 1 in the form $\underline{s}\ \underline{r}$ so that the lengths of $\underline{s}$ and $\underline{r}$ are as nearly equal as possible, but with $\underline{s}$ longer than $\underline{r}$. For each relator, other than relation 1, determine if $\underline{s}$ is a subword of that relator. If so, replace the subword with $\underline{r}^{-1}$ and reduce the result. [It is left as an (easy) exercise for the reader to show that if a word larger than $\underline{s}$ could have been replaced, then the replacement described will accomplish the larger replacement.] Now circle shift relation 1 left by one syllable and repeat the procedure described above with the resulting representation of relation 1. When all circle shifts of relation 1 have been tested, invert relation 1 and test all circle shifts of the inverse similarly.

After completing relation 1, test the remaining relations (possibly altered) in the same way. After all relations have been tested in this way, the process repeats beginning at relation 1. The operation is completed when a complete cycle is performed in which no replacements are made.

Each time that a replacement is performed, the details of the replacement are printed if the DETAIL switch is on (see Detail Switch Instruction). These details show the relation from which $\underline{s}$ and $\underline{r}$ were formed, the relation that was shortened, and the words $\underline{s}$ and $\underline{r}^{-1}$. The format of this information is:

RELATION i IS CONTAINED IN RELATION j

$\underline{s}$

REPLACED BY $\underline{r}^{-1}$.

For example:

RELATION 3 IS CONTAINED IN RELATION 6

A1 B1 A1 B1^{-1} A1^{-1} B1 A1 B1 A1^{-1}

REPLACED BY B1 A1 B1 A1^{-1} B1^{-1} A1 B1

The DETAIL switch is set on at the beginning of each program run.

Since the containment operation can take a large amount of time, a "sense switch" has been put in this part of the program. This switch is set by typing any message of 28 characters or less during the containment operation. After completing the testing of each non-trivial relation, the sense switch is tested. If it has been set, the program types the amount of time used by the program (see Time Instruction), the number of the current pass through the relations, the number of the relation that it just finished testing, and the number of containments performed. The program then asks "SHALL I CONTINUE?" If the response is "NO", the operation is terminated, otherwise the sense switch is reset and the operation continues. Example:

```
CPU TIME IS    0 MIN 27 SEC
I/O TIME IS    0 MIN 12 SEC
PASS NUMBER 1
COMPLETED RELATION 5
3 CONTAINMENTS PERFORMED
SHALL I CONTINUE?  NO←
```

Redundant Relations: The program recognizes a relation as redundant if some manipulation of the presentation causes it to become trivial (i. e., 1). However, relations which become trivial as the normal part of a manipulation (e. g., solving in a relation for a generator) are not considered redundant. After the program performs a manipulation which can result in recognizing redundant relations, it checks to see if there are any. If so, the programs types and prints their numbers and marks them so that they will not be listed again. This process is referred to as "listing redundant relations". An example of such a listing is:

```
THESE RELATIONS WERE FOUND REDUNDANT:  2  4  5
```

Simplification Step: A simplification consists of solving a relation for a generator. A generator x is solvable in a relation if the relation can be written in the form $x = w$ where x does not appear in w. In terms of relators, this is equivalent to x appearing just once in the relator and that occurrence having an exponent of $+1$ or -1.

If any generator appears solvably in any relation, then a simplification step is performed on a generator-relation pair chosen as follows: The generator is the first generator in the generator table for which there is a relation in which it occurs solvably. The relation is the first (i. e., lowest numbered) relation in which that generator occurs solvably. It should

be noted that the user may force the program to regard any desired generators as being unsolvable by using the Preserve Instruction.

If a generator is found that can be solved for, the program takes the following steps:

(1) a message is typed and printed stating which generator is being solved for and in which relation it is solved for,

(2) a message is typed and printed giving the word which is equal to the generator (except as described below),

(3) the relation is changed to 1 ,

(4) the generator is removed from the generator table,

(5) each remaining occurrence of the generator in the presentation is replaced by its equivalent word,

(6) a transformation entry is written into the disk save file showing the operation performed,

(7) a containment operation is attempted,

(8) redundant relations are listed,

(9) the resulting presentation is written into the disk save file,

(10) the name of the presentation is typed, and

(11) the presentation is printed.

The typed output of step (2) may be suppressed by setting the TYPEENABLE switch off. This may be accomplished by the Type Switch Instruction or by pressing the teletype BREAK key during the output of a simplification step. If the latter method is used, it is possible to lose the typed output from the next simplification step (if any) because of timing problems inherent in the

use of the BREAK key.

Error Messages: Error messages are typed when the program recognizes an error. These messages are identified by two leading asterisks (see examples below). The program recognizes two basic types of errors, syntax errors and operation errors.

(1) Syntax errors are errors in the form of the input received from the user. These may be caused either by typing errors on the part of the user or by error in the transmission from the user to the program (fortunately, a rare occurrence). Since syntax errors may occur in many ways, the usual response of the program is the generalized error message:

```
** PLEASE REENTER INSTRUCTION
```

Under some circumstances, the error message may be more specific and is intended to be self-explanatory.

(2) An operation error is recognized when the program is unable to execute an instruction. Operation errors fall into two classes: (a) Mathematically impossible instructions (e. g., solving for a generator in a relation which does not contain it) generate a self-explanatory message. (b) If the program does not have enough storage space to perform an operation, it will abandon the operation and type a message of the form:

```
** NOT ENOUGH ROOM TO REPLACE

** OPERATION ABANDONED
** DUMP ON PRINTER FILE
```

A message will be printed specifying what part of the program abandoned the operation (this message will often require reference to the program code for interpretation) followed by a diagnostic dump (see Chapter IV). Because the presentation is in a very uncertain state if abandoned, the dump is followed by clearing the storage areas and setting a trivial presentation (named ???/???-0) as the current presentation. Thus the user should follow an abandoned operation with either an Enter Presentation Instruction or a Read Presentation Instruction if he wishes to continue.

2. Input System Description

This section gives a brief description of the free-format input system used by the GROUP program. All input to the program, except for the "sense switch", is processed by this system. A more detailed description is given in Chapter IV (RDNUM package). This system is adapted from that described in [2]. The input system divides a line of input into a sequence of character strings. These strings are assigned one of the following classifications:

1. integer
2. real number
3. special symbol
4. alphabetic symbol
5. illegal symbol
6. no input.

These classifications will be used to describe the general input components and instructions (sections 3 and 4). Below we describe those character strings

that fall into each classification.

In decoding the input, the program scans the line from left to right. The first symbol begins with the first non-blank character encountered in the line and each successive symbol begins where the previous symbol stops. A blank always terminates a symbol, but is never a character of a symbol. Leading blank characters are ignored. Each input line is ended by a group mark (←) which is replaced by a blank to determine the end of a symbol.

An integer is a sequence of digits and may be preceded by a sign (+ or -). The first character which is not a digit, other than a period (decimal point) or exponent symbol (@), begins the next symbol.

A real number may have one of the following forms:

sd.	sd. x
sd. d	sd. dx
s. d	s. dx
	sdx

where s represents an optional sign, d represents a string of digits (at least one) and x represents an exponent, which has the form @sd (s and d as before). The real number extends as far as one of these patterns continues. (Real numbers are not used by the GROUP program, but were not removed from the system.)

A special symbol is one of the following characters:

, # : = [ ] ( )

and in addition, the period (or decimal point) and the signs (+ and -) are also special characters if not part of an integer or real number.

An alphabetic symbol is a string which begins with a letter or the exponent symbol and continues to the first character that is not a letter, digit or exponent symbol. The term "letter" above includes the letters of the alphabet as well as the characters &\ $ * ; /%" (note that \ appears on the printer as the multiplication symbol ×). Regardless of the length of an alphabetic symbol, the program uses a maximum of six characters which are taken from the beginning of the string.

No input refers to an input line whose first non-blank character is the group-mark.

An illegal symbol is a character string that contains the illegal character (?) or that does not fall into any of the above classifications. If the program encounters an illegal symbol, it usually types an error message and discards the remainder of the line.

3. General Input Components.

In this section we define the main components of the program's input. These components are used in the descriptions through the remainder of this chapter. The formal syntax of these terms is given in Appendix A. The same terminology is used here as in the formal syntax.

Comment: The term "comment" is used to refer to optional words and phrases which may make the instructions to the program more readable. For example, "SUPPRESS PRESENTATION TYPING" is somewhat clearer than "SUPPRESS", although both mean the same to the program (see Section 4, Type Switch Instruction). The descriptions of the instructions specify where comment may appear and any restrictions that may apply to it. It should be

remembered that RDNUM is used to break each instruction into its component character strings and the restrictions stated are in terms of these components. For example, in the Build Chain Instruction, a comment is allowed that does not contain "TO". The following string:

"UPTO PRESENTATION 5"

does not contain "TO" according to RDNUM, while

"UP TO PRESENTATION 5"

does contain "TO".

Generators: An acceptable generator is any alphabetic symbol. An acceptable generator is a generator if it appears in the table of generators (see section 1) of the current presentation. An acceptable generator may be placed in this table by (1) appearing in a define generator list, (2) appearing in a word (see below for restrictions), or (3) appearing as a new generator in a Define Instruction (see section 4).

A define generator list consists of a sequence of symbols. When a define generator list is precessed, alphabetic symbols are added to the generator table, commas are ignored, and other symbols (except : and #) cause an error message to be typed. If an alphabetic symbol is already a generator, a message is typed alerting the user to this condition.

If a define generator list is too long for one line, a colon (:) will instruct the program to ignore everything following it and to continue processing on a new line. To indicate that it is ready for a new line to be entered, the program will type "CONTINUE: " and wait for the new line. The construction

: comment ←

is referred to as a continuation.

If the special symbol "#" appears in a define generator list, the program immediately ceases execution of the current instruction and accepts a new instruction. Any alphabetic symbols encountered before the termination character (#) remain generators, however the presentation is not written into the disk save file. This feature is usually used only if the user changes his mind about an instruction after he has begun entering it and often requires some corrective action to return the presentation to its earlier status.

A generator list, to be distinguished from a define generator list, is a sequence of symbols. However, only those symbols that are generators are used, the remainder being comment. (See, for example, the Preserve Instruction and Release Instruction). The continuation and termination characters are ignored in a generator list.

Words: A word is a sequence of factors. A factor, in turn, may be either an acceptable generator, a term, or a commutator, possibly followed by an exponent. An exponent, if it appears, is an integer. A term has the form

$$(\underline{w})$$

where $\underline{w}$ is a word, and a commutator has the form

$$[\underline{w}_1, \underline{w}_2]$$

where $\underline{w}_1$ and $\underline{w}_2$ are words. It should be noted that this definition is recursive and it is permitted to have commutators within terms, terms within terms, etc. The identity word (word with zero syllables) is typed "1".

The continuation and termination characters (: and #) are included as in the description of the define generator list. In contrast with that description, however, the termination character causes the part of the word preceding it to be ignored rather than stored.

A term that is not followed by an exponent is treated as though the enclosing parentheses were absent. If an exponent is present, the word enclosed by the parentheses is repeated as specified by the exponent. For example:

$$\underline{x}\,(\underline{a}\,\underline{b}\,\underline{c})\,\underline{y} = \underline{x}\,\underline{a}\,\underline{b}\,\underline{c}\,\underline{y}$$

$$\underline{x}\,(\underline{a}\,\underline{b}\,\underline{c})^{2}\,\underline{y} = \underline{x}\,\underline{a}\,\underline{b}\,\underline{c}\,\underline{a}\,\underline{b}\,\underline{c}\,\underline{y}$$

$$\underline{x}\,(\underline{a}\,\underline{b}\,\underline{c})^{-1}\underline{y} = \underline{x}\,\underline{c}^{-1}\,\underline{b}^{-1}\,\underline{a}^{-1}\,\underline{y}$$

$$\underline{x}\,(\underline{a}\,\underline{b}\,\underline{c})^{0}\,\underline{y} = \underline{x}\,\underline{y}\,.$$

The commutator is defined to be

$$[\underline{w}_1, \underline{w}_2] = \underline{w}_1\,\underline{w}_2\,\underline{w}_1^{-1}\,\underline{w}_2^{-1}\quad .$$

If either $\underline{w}_1$ or $\underline{w}_2$ is the identity, then $[\underline{w}_1, \underline{w}_2] = \underline{1}$. An exponent following a commutator causes the commutator to be repeated as specified by the exponent. For example:

$$\underline{x}\,[\underline{a}, \underline{b}]\,\underline{y} = \underline{x}\,\underline{a}\,\underline{b}\,\underline{a}^{-1}\underline{b}^{-1}\,\underline{y}$$

$$\underline{x}\,[\underline{a}, \underline{b}]^{2}\,\underline{y} = \underline{x}\,\underline{a}\,\underline{b}\,\underline{a}^{-1}\,\underline{b}^{-1}\,\underline{a}\,\underline{b}\,\underline{a}^{-1}\,\underline{b}^{-1}\,\underline{y}$$

$$\underline{x}\,[\underline{a}, \underline{b}]^{-1}\underline{y} = \underline{x}\,\underline{b}\,\underline{a}\,\underline{b}^{-1}\,\underline{a}^{-1}\,\underline{y}$$

$$\underline{x}\,[\underline{a}, \underline{b}]^{0}\underline{y} = \underline{x}\,\underline{y}$$

In describing some of the instructions, the phrase "restricted word" is used. A restricted word is a word with the added requirement that each acceptable generator that appears must actually be a generator. If an

acceptable generator that is not a generator occurs within a restricted word, it is considered an error. Such an appearance in a word, however, will cause the program to ask whether the symbol should be made a generator (i. e., added to the generator table). If the answer is "YES", it is added to the table and processing continues. If the answer is not "YES", then the symbol is considered an error as with a restricted word. Examples (from Read Presentation Instruction):

```
ENTER GENERATORS:  A B C D E←
RELATION 1.  A B G C D←
SHOULD "G" BE A GENERATOR?  YES←
RELATION 2.  A B F C D←
SHOULD "F" BE A GENERATOR?  NO←
     ** PLEASE REENTER
```

(The portions typed by the user are underlined).

Relations: Relations in a presentation are referred to by the number that gives the position in the table of relations. Each time that a relation is typed or printed by the program, it is numbered in this way. Such a number is called a relation number and should be typed as an integer when used. A relation list is a sequence of relation numbers with any desired comment interspersed. A relation designator is a relation number, preceded by the alphabetic symbol "RELATION" if desired.

4. Operating Instructions.

To initiate the B-5500 program, called GROUP/GROUP, it is sufficient to type "??EXECUTE GROUP←" unless it desired to change some of of the compilation parameters. In that case, the user should refer to Chapter IV

and the appropriate machine manuals (at the University of Wisconsin, User Manual, Vol VII). After the system beginning of job message, the program will identify itself and its version number, then type a colon and wait.

```
??EXECUTE GROUP←
 7:GROUP/GROUP=03 BOJ 1505 FROM 03/02
GROUP PRESENTATION PROGRAM - VERSION 9.3
:
```

At this point the program is waiting for an instruction to be entered. The user should type an instruction according to the descriptions that follow. The formal syntax of each instruction is contained in Appendix A. If more information is needed to process the instruction, the program will direct the user to type it in or ask questions of the user. When the program completes an instruction and has output its results, it will again type a colon and wait for another instruction (except, of course, after a Quit Instruction).

The set of instructions has been divided into eight classes by their function (see Table 1). This classification is somewhat arbitrary in certain cases; for instance, the Contain Switch Instruction might have been placed in the Switch Control group. The assignments have been made to indicate their most common usage. The descriptions of the individual instructions are given in the order of Table 1. Except where indicated otherwise, each instruction entered is copied to the printer output file.

Input/Output:
- Enter Presentation
- Read Presentation
- Print
- Type
- Comment
- Print Width

Manipulation:
- Solve
- Simplify
- Check
- Preserve
- Release
- Define
- Contain Switch

Editing:
- Remove
- Alter
- Replace
- Add
- Shift
- Invert

Chain:
- Follow
- Build Chain
- Trace

Switch Control:
- Detail Switch
- Type Switch

Disk Save File Control:
- Disk File Name
- New Disk File
- Disk Summary

Debugging:
- Cleanup
- Dump

Miscellaneous:
- Abelianize
- Time
- Suspend
- Quit

Table I

Classification of Instructions

Enter Presentation Instruction: The Enter Presentation Instruction allows the user to type in a presentation. The verb initiating this instruction is "PRESENTATION" and the remainder of the line may be any descriptive comment desired. If no disk save file is attached to the program for storage, one will be created at this point.

The program will type "ENTER GENERATORS:". The user should respond with the generators of the presentation in the form of a define generator list. After the generators are entered, the program will type "RELATION 1." and the user should type in the first relation in the form of a word. The program will then type "RELATION 2." and the user should again respond with a word, and so on until all relations have been entered. To indicate that all relations have been entered, type only a group mark (←) when the program asks for the next relation. The program will then type the entire presentation, which should be carefully checked for errors.

After typing the presentation, the program will ask "DO YOU WANT TO MAKE CORRECTIONS?". If an error has been found, the user should answer "YES", otherwise "NO". If the answer is "YES", the program will type a reminder of the correction procedure and begin a new line with "RELATION". The user should type the number of the relation to be corrected, the character "=", and the correct relation. The program will type back the relation and again begin a new line with "RELATION". The process should be repeated until all corrections have been made (see examples). Any relation may be corrected as many times as necessary. If a relation has been omitted, it may be added by giving a relation number one greater than the number of

relations already entered. As many relations as necessary may be added in this manner (up to the program limit) by "correcting" successively higher numbered relations. If an extra relation has been typed, it may be removed by setting it to 1 or by using the Remove Instruction after this instruction is completed. To indicate that all corrections have been made, type "↵" when the program types "RELATION".

After the presentation has been completely entered and corrected, each relation is circularly reduced and redundant relations are listed. The presentation is then written into the disk save file and printed. A containment operation is attempted and, if successful, redundant relations are listed, and the resulting presentation is written into the disk save file, typed and printed.

Example: (underlined portions were typed by the user)

```
: PRESENTATION FOR A TREFOIL KNOT←
ENTER GENERATORS: A, B, C←
RELATION 1.  A C A-1 B-1←
RELATION 2.  B A B-1 C-1←
RELATION 3.  C B C-1 B-1←
RELATION 4.  ←

PRESENTATION GROUP/DEMO-3
GENERATORS  A, B, C
RELATION
    1.  A C A -1 B -1
    2.  B A B -1 C -1
    3.  C B C -1 B -1
```

```
DO YOU WANT TO MAKE CORRECTIONS? YES←
TYPE RELATION NUMBER, "=", AND CORRECT RELATION

RELATION 3 = C B C-1 A-1←
RELATION 3.  C B C -1 A -1

RELATION ←
```

Read Presentation Instruction: The Read Presentation Instructions allows the user to return to a presentation previously stored in a disk save file. The instruction consists of a verb followed by the name of the presentation desired (see section 1). The verbs that are recognized for this instruction are "READ" and "RECALL". Upon successful completion of the instruction, "READ" causes the program to type and print the presentation, while "RECALL" causes only the name to be typed and printed. Note that generators are not preserved regardless of their state when stored in the disk save file (see Preserve Instruction and Simplification Step). Example:

```
: READ DEMO-2←
REMINDER:  NO GENERATORS ARE PRESERVED

PRESENTATION GROUP/DEMO-2
GENERATORS  A, B, C
RELATION
    1.  A C A -1 B -1
    2.  B A B -1 C -1
    3.  C B C -1 A -1
```

Print Instruction: This instruction causes the current presentation to be printed. The instruction consists of "PRINT" followed by any desired comment.

Type Instruction: The Type Instruction allows the user to have all or part of the current presentation typed. No printer output is produced. The instruction must start with "TYPE". If this is the entire instruction, the whole presentation is typed. If more than "TYPE" is entered, the next component specifies the part of the presentation to be typed. The recognized options are:

option	portion typed
PRESENTATION	entire presentation
GENERATORS	table of generators
relation designator	specified relation
SIZE or LENGTH	number of generators and relations, and number of syllables in each relation.

Examples:

```
: TYPE RELATION 2←
: TYPE GENERATORS←
```

Comment Instruction: This instruction allows the user to insert comments into the typed and printed listings. The instruction begins with an asterisk (*) which must be followed by a special character or a blank. The entire line of input is copied onto the printer with the asterisk indented 20 spaces.

Print Width Instruction: This instruction allows the user to specify the width of the printed output produced by the program. For example, output restricted to 75 columns is easily trimmed to loose leaf size. The instruction consists of "USE" followed by an integer giving the number of columns desired. Any additional input following this is treated as comment and ignored.

The number of columns specified must be at least 50 and may not exceed the number of columns available on the hardware (132 at the University of Wisconsin). The width is set to the maximum allowed at the beginning of each program run. The only reason that the program may exceed the imposed limit is in printing a dump (see Chapter IV).

Example (the significant portion of the instruction is underlined:

: USE 75 PRINTER COLUMNS←

Solve Instruction: The Solve Instruction commands the program to solve for a generator and substitute its equivalent for each occurrence in the presentation. It is similar to a simplification step (see section 1) except that the user may specify what simplification is to be performed.

The instruction begins with "SOLVE". The user may then specify the generator for which the program is to solve and/or the relation in which to solve. To specify the generator, the instruction must include "FOR" followed by the desired generator. To specify the relation, the instruction must include a relation number. If either construct appears more than once, the last one of each that is encountered is used. Comments are permitted anywhere in the instruction.

If no generator or relation is specified, the program will attempt

one simplification step and, if a simplification is performed, types the resulting presentation. If only a generator is specified, the program finds the first (i. e., lowest numbered) relation in which the generator can be solved and types the action to be taken. If the relation only is specified, the program determines the first generator which can be solved for in that relation and types the action to be taken. If the specified action cannot be performed, an error message is typed.

If the action can be performed and either a generator or a relation is specified, the following steps are taken:

(1) the word which is equivalent to the generator is typed and printed,

(2) the relation is changed to $\underline{1}$,

(3) the generator is removed from the generator table,

(4) each remaining occurrence of the generator is replaced by its equivalent word,

(5) a transformation entry is written into the disk save file showing the replacement that was performed,

(6) a containment operation is attempted,

(7) redundant relations are listed,

(8) the presentation is written into the disk save file, and

(9) the resulting presentation is typed and printed.

A generator that is preserved (see Preserve Instruction) may be solved for by specifying it.

Examples:

: SOLVE←

: SOLVE FOR A←

: SOLVE RELATION 2←

: SOLVE FOR A IN RELATION 2←

Simplify Instruction: This instruction specifies that the program should perform a number of simplification steps. It begins with "SIMPLIFY" and the remainder specifies the number of steps to be attempted. If the remainder contains a positive integer, the first such encountered is the maximum number of steps to be taken. If no positive integer is contained in the instruction, the program will perform simplification steps as long as solvable generators remain. After the simplifications are performed, the resulting presentation is typed.

Examples:

: SIMPLIFY 5←

: SIMPLIFY A MAXIMUM OF 5 TIMES←

The two examples above are equivalent to each other, as are the two below.

: SIMPLIFY←

: SIMPLIFY AS FAR AS POSSIBLE←

Check Instruction: The Check Instruction allows the user to determine whether a generator is solvable and, if so, which relations may be solved for it. The instruction has two forms: (1) "CHECK", and (2) "CHECK" followed by a generator list. In the first form, for each generator in the generator table, the program types a list of the relations in which the generator

appears solvably. If the generator does not appear solvably in any relation, a message is typed stating that fact. The second form acts as the first except that only the generators appearing in the generator list are checked. The typed output is also printed.

For example, suppose the presentation is

```
PRESENTATION GROUP/DEMO-4
GENERATORS  A, B, C, D
RELATION
    1.  A B A -1 B -1 C
    2.  C 2 B -1 A
    3.  A -1 C B 4
    4.  D 2 C -1 D -1 A
```

then we could obtain

```
:  CHECK WHETHER A, B, C OR D CAN BE SOLVED FOR←
A IS SOLVABLE IN RELATION  2  3  4
B IS SOLVABLE IN RELATION  2
C IS SOLVABLE IN RELATION  1  3  4
D CANNOT BE SOLVED FOR
```

Preserve Instruction: This instruction allows the user to specify that certain generators may not be solved for unless the program is explicitly instructed to do so (see Solve Instruction). The instruction consists of "PRESERVE" followed by a generator list. Each generator in the generator list is typed and preserved. The status of generators not in the generator list is not affected. A generator may be returned to the normal state by using the

Release Instruction. Note that any presentation read from a disk save file has no generators preserved regardless of their state when written into the file.

Example:

```
: PRESERVE A AND B←
PRESERVED  A B
```

Release Instruc 'on: The Release Instruction negates the effect of the Preserve Instruction. This instruction has two formats: To release all generators, type "RELEASE" only. To release only certain generators, type "RELEASE" followed by a generator list containing the generators desired. In the second form, all generators in the generator list are typed and released.

Examples:

```
: RELEASE GENERATORS B AND C←
RELEASED  B C
: RELEASE←
```

Define Instruction: This instruction allows the user to define a new generator as some word in the previous generators (a type II Tietze equivalence). The format of the instruction is:

DEFINE $\underline{x} = \underline{w}$

where $\underline{x}$ is the new generator (an acceptable generator that is not already a generator), and $\underline{w}$ is a restricted word.

The program will repeat the definition for verification. If it is correct, the following actions will be taken:

(1) the new generator is added to the generator table,

(2) a relation of the form $\underline{x}^{-1}\underline{w}$ is added to the presentation ($\underline{x}$ and $\underline{w}$ as above),

(3) a transformation entry is written into the disk save file showing the relation that was added,

(4) the new generator is inserted into the previous relations by an operation similar to the containment operation, except that only the new relation is tested as a subword, and

(5) the resulting presentation is written into the disk save file, typed and printed.

Example:

```
: DEFINE X = A B ( A C ) 2←
X = A B A C A C
IS THIS CORRECT?  YES←

PRESENTATION GROUP/DEMO-6
GENERATORS  A, B, ...
RELATION
    1.  ...
```

Contain Switch Instruction: This instruction controls the CONTAINSWITCH (see Containment Operation). The recognized verbs for this instruction are "CONTAIN, "*CONTAIN", and "NOCONTAIN". Any desired comment may follow the verb. The first two verbs cause the CONTAINSWITCH to be set; and the last, reset. A containment operation is then attempted. If it is successful (in the case of the last verb, it is never successful), redundant

relations are listed and the resulting presentation is written into the disk save file, typed, and printed. If the verb is "*CONTAIN", the CONTAINSWITCH is returned to its value prior to the instruction after the containment operation has been attempted.

Because the containment operation can be very time consuming, the CONTAINSWITCH is reset at the beginning of each program run.
Examples:

```
: CONTAIN←
: *CONTAIN←
: NOCONTAIN OPERATIONS←
```

Remove Instruction: This instruction allows the user to remove from the presentation (1) trivial relations, (2) specific relations, or (3) specific generators.

To remove trivial relations (relations which are 1) form a presentation, the user should type either "REMOVE" or "REMOVE TRIVIAL". The second form may be followed by comment, if desired. If the presentation has no trivial relations, a message is typed and printed to that effect. Otherwise, the non-trivial relations are renumbered, and the presentation is written into the disk save file, typed and printed. Examples of input are:

```
: REMOVE←
: REMOVE TRIVIAL RELATIONS←
```

To remove specific relations from a presentation, a remove verb should be typed followed by "RELATIONS" and a relation list. The recognized verbs are "REMOVE" and "DELETE". "REMOVE" specifies that the

relations whose numbers appear in the relation list should be removed and the remaining relations renumbered. "DELETE" specifies that the relations whose numbers appear in the relation list should be made trivial; i. e., replaced with the identity. In either case, the resulting presentation is written into the disk save file, typed and printed. Examples:

: REMOVE RELATIONS 1, 2 & 4←

: DELETE RELATIONS 1, 2 AND 4←

To remove generators from a presentation, the user should type a remove verb ("REMOVE" or "DELETE") followed by "GENERATORS" and a generator list. The following actions are taken by the program:

(1) the generators appearing in the generator list are removed from the generator table and each occurrence of such a generator in a relation is replaced by 1,

(2) a transformation entry is written in the disk save file giving the generators removed,

(3) redundant relations are listed,

(4) the resulting presentation is written into the disk save file, typed and printed, and

(5) a containment operation is attempted.

If the containment operation is successful, the following actions are also performed:

(6) redundant relations are listed again, and

(7) the resulting presentation is written into the disk save file, typed and printed.

An example of the input for this option is:

: REMOVE GENERATORS A, B AND X←

Alter Instruction: The Alter Instruction allows the user to edit a relation of the presentation. The portion of the relation to be edited must appear as the first part of the relation (i. e., on the left as typed by the program). This may be achieved by using the Shift Instruction, if necessary. The user must specify (1) the relation to be edited, (2) the portion of the relation to be edited, and (3) the new value of the affected portion.

The relation to be edited is specified by a relation key which is "RELATION" immediately followed by a relation number. The portion of the relation to be edited is specified by a size key which is "FIRST" immediately followed by an integer. The new value of the affected portion is specified by "TO" immediately followed by a word.

The instruction must begin with "ALTER" and end with the specification of the new value of the affected portion. Between these two parts come the relation and size keys, and any desired comment. At least one relation key must appear and the last relation key in the instruction specifies the relation to be altered. No size key need appear in the instruction, but the last to appear specifies the number of syllables of the relation which are to be altered beginning with the left-most syllables. If no size key appears, the entire relation is specified and is replaced completely by the word specified in the instruction (this effect may also be achieved by specifying a number of syllables which is greater than or equal to the length of the relation). A size key specifying zero syllables causes the new value to be appended to the front of the relation.

After the relation is altered, the following actions are performed:

(1) the relation is circularly reduced,

(2) redundant relations are listed,

(3) the new presentation is written into the disk save file,

(4) the altered relation is typed,

(5) the presentation is printed, and

(6) a containment operation is attempted.

If the containment operation causes any replacements to be made, the following additional actions are performed:

(7) redundant relations are listed, and

(8) the presentation is written into the disk save file, typed and printed.

Example (ths significant portions of the instruction are underlined):

```
PRESENTATION GROUP/DEMO-6
RELATION 4.  D 2 C -1 D -1 A
: ALTER THE FIRST 2 SYLLABLES OF RELATION 4 TO B 2 A-3 C 2←
PRESENTATION GROUP/DEMO-7
RELATION 4.  B 2 A -3 C 2 D -1 A
```

Replace Instruction: The Replace Instruction allows the user to edit relations by replacing all occurrences of one word with another word. The instruction consists of:

(1) "REPLACE",

(2) a non-trivial word ("old" word) followed by a delimiter,

(3) a restricted word ("new" word) followed by a delimiter, and

(4) a specification of the relations affected.

The non-trivial ("old") word must be a restricted word which does not reduce to $\underline{1}$. The delimiters of the words may be either "," or "←". In addition, if "WITH" is not a generator, it may be used as the delimiter following the "old" word. Similarly "IN" may be used as the delimiter following the "new" word if it is not a generator. (See examples) If "←" is used as the delimiter in (2) or (3), the program will type "WITH:" or "IN:", respectively, and wait for the remainder of the instruction.

Specification of the affected relations begins with either "ALL", "RELATIONS", or a relation number. Any desired comment that does not include one of these symbols may appear first. If "ALL" is the first of these symbols to appear, the replacement is performed in all relations and a transformation entry showing the replacement is written in the disk save file. All relations are also specified if the specification part is "no input" as described in section 2 (see also the last example below). If "RELATIONS" or a relation number is found first, it signals the beginning of a relation list specifying the affected relations. In this case, no transformation entry is written in the disk save file.

The replacement is made in each specified relation in two operations: First, all occurrences of the "old" word and its inverse are found and marked. Second, the marked occurrences are replaced by the "new" word (or its inverse as appropriate). This method prevents the possibility of the program "looping" as might be expected if an attempt were made to replace, for example, $\underline{a}$ with $\underline{a}^2$.

After the specified replacements have been made, the program

performs the following operations:

(1) redundant relations are listed,

(2) the presentation is written into the disk save file, typed and printed, and

(3) a containment operation is attempted.

If the containment operation causes any further replacements, steps (1) and (2) are repeated.

Examples: The first four examples are equivalent.

```
: REPLACE D E F WITH G H I-1 IN RELATIONS 1, 3 AND 5←
: REPLACE D E F, G H I-1, 1 3 5←
: REPLACE D E F WITH G H I-1←
    IN: RELATIONS 1, 3, AND 5←
: REPLACE D E F←
  WITH: G H I-1←
    IN: RELATIONS 1 3 AND 5←
```

The following examples are the same as those above except that all relations are specified for the replacement.

```
: REPLACE D E F WITH G H I-1 IN ALL RELATIONS←
: REPLACE D E F←
  WITH: G H I-1←
    IN: ←
```

Add Instruction: This instruction allows the user to add generators or relations to a presentation.

To add generators to a presentation, the user should enter "ADD GENERATORS" followed by a define generator list giving the new generators. The acceptable generators in the define generator list are added to the

generator table, the resulting presentation is written into the disk save file, the new generators are typed and the presentation is printed.

To add a relation to a presentation, the user enters "ADD RELATION" followed by a word. The word will be typed for verification. If the word is correct, the program will

(1) circularly reduce the new relation,

(2) write a transformation entry into the disk save file giving the new relation,

(3) write the presentation into the disk save file and print it,

(4) type the name of the presentation, and

(5) attempt a containment operation.

If the containment operation is successful, the program will

(6) list redundant relations, and

(7) write the presentation into the disk save file, and type and print it.

Examples:

```
: ADD GENERATORS X, Y, Z←
PRESENTATION GROUP/DEMO-11
ADDED GENERATORS  X, Y, Z
: ADD RELATION A 2 B-1 C←
RELATION 6.  A 2 B -1 C
IS THIS CORRECT?  YES←
PRESENTATION GROUP/DEMO-13
```

Shift Instruction: This instruction allows the user to change the format of a relation by shifting it circularly. A relation may be circle shifted in either direction by any number of (reduced) syllables. The instruction consists of

(1) "SHIFT" ,

(2) a relation designator,

(3) either "RIGHT" or "LEFT", giving the direction of the shift,

(4) an integer specifying the size of the shift in syllables, and

(5) any desired comment.

After performing the shift, the resulting presentation is written in the disk save file, the shifted relation is typed, and the presentation is printed.

Example (the significant portion is underlined):

: SHIFT RELATION 2 LEFT 5 SYLLABLES←

Invert Instruction: This instruction consists of "INVERT" followed by a relation designator. Any comment desired may follow the relation designator. The specified relation is inverted and typed. The resulting presentation is written into the disk save file and printed.

Example:

: INVERT RELATION 2←

Follow Instruction: This instruction causes the program to take a word of one presentation and determine its equivalent in the current presentation by retracing the transformations which created the current presentation from the other (see section 1, Chaining). If a chain is already built ending at the current presentation, that chain is used. Otherwise a chain is built that begins at the original input presentation (created by an Enter Presentation Instruction). To construct a chain that begins at a presentation other than the original input, the Build Chain Instruction must be used.

The instruction consists of "FOLLOW" and a restricted word

in the presentation at the beginning (top) of the chain (obtained as described above). The program will ask if the word is circular. An affirmative answer is "YES" , all other answers are negative. The word will then be subjected to those operations that are specified in the chain to obain its equivalent in the current presentation. If the tracing option has been selected (see Trace Instruction), each operation performed on the word will be printed, as well as the word's equivalent in each intermediate presentation. The final equivalent word is then typed and printed.

The program then asks:

DO YOU WISH CONTAINMENT OR ABELIANIZATION?

If the answer is not an alphabetic symbol or is "NO", the program exists from this instruction. Otherwise, if the answer is not "CONTAINMENT" o· "ABELIANIZE", the program will type

PLEASE SPECIFY WHICH:

The answers to this question are the same as to the first question. If the answer to either question is "CONTAINMENT", the program attempts to shorten the equivalent word by an operation similar to the containment operation. This operation differs from the containment operation in the order in which the tests are performed. Details of the replacements performed are printed in the same manner as in the containment operation. This feature is under the control of the DETAIL switch (see Detail Switch Instruction). The equivalent word after containment is typed and printed. If the answer to either question is "ABELIANIZE", the program abelianizes the word, and types and prints the resulting word. After either operation, the program returns to the first question.

Build Chain Instruction: This instruction causes the program to build a chain that ends at the current presentation. If desired, the user may specify the presentation that is to be at the beginning of the chain.

The instruction must begin with "BUILD" and may be followed by any comment not containing "TO". To specify the presentation at the top (beginning) of the chain, the last components of the instruction must be "TO" followed by the name of the presentation. If the beginning presentation is not specified, the program assumes that the original input presentation is meant.

If the beginning of the chain is specified, but the current presentation was not derived from the specified presentation, the specification is ignored. If the chain is successfully built (evidenced by a lack of error messages), the name of the presentation at the top of the chain is typed. If the tracing option is selected (see Trace Instruction), the chain will be printed.

Examples:

```
: BUILD CHAIN←
TOP OF CHAIN IS PRESENTATION GROUP/DEMO-1

: BUILD CHAIN TO GROUP/DEMO-15←
TOP OF CHAIN IS PRESENTATION GROUP/DEMO-15
```

Trace Instruction: The Trace Instruction controls the tracing of chain operations (Follow and Build Chain Instructions). If tracing is selected, each time that a chain is successfully built, it will be printed, and the intermediate steps in each Follow Instruction will be printed. The global Boolean variable TRACECHAIN stores the current setting of this option.

The instruction consists of a verb followed by comment. The

recognized verbs are "TRACE" and "NOTRACE" which select and suppress tracing respectively. If tracing is selected, the current chain (if any) is discarded, forcing the program to rebuild and print the chain at its next use. Tracing is initially suppressed at the beginning of each program run.

Examples:

```
: TRACE CHAIN OPERATIONS←
: NOTRACE←
```

Detail Switch Instruction: This instruction controls the printing of details of replacements performed in containment operations by affecting the DETAIL switch (see Containment Operation and Follow Instruction). The recognized verbs for this instruction are "DETAIL" and "NODETAIL" which may be followed by comment if desired. These verbs respectively select and suppress detail printing. The option is initially selected at the beginning of each program run.

Examples:

```
: NODETAILS←
: DETAIL SWITCH ON←
```

Type Switch Instruction: This instruction controls the setting of the TYPEENABLE switch, which in turn controls the typing of presentations and the typing of equivalent words in a simplification step. If TYPEENABLE is set on, all such typing is performed; if off, it is suppressed. At the beginning of each program run, the switch is set so that all typing is performed.

The instruction begins with a verb, either "SUPPRESS" or

"RESUME", and may be followed by comment. The verbs respectively set TYPEENABLE to suppress the typeout that it controls and to resume the typeout. The simplification step discussion describes another means of setting the switch to suppress the typing.

Examples (the significant portions are underlines):

```
: SUPPRESS←
: RESUME TYPEOUTS←
```

Disk File Name Instruction: This instruction specifies the naming convention to be used in name disk save files (see section 1, Disk Storage). The instruction consists of a verb and comment, if desired. The verbs recognized are "STANDARD" and "NONSTANDARD" and select the standard and nonstandard conventions respectively. The standard convention is specified at the beginning of each program run. This instruction generates no printer output.

Examples (the significant portions are underlined):

```
: STANDARD DISK SAVE FILE NAMES←
: NONSTANDARD←
```

New Disk File Instruction: This instruction, consisting of "FINDFILE" and a comment, causes a new disk save file to be created and used for storage (see section 1, Disk Storage). No printer output is generated.

Disk Summary Instruction: This instruction causes the program to type the name of the current disk save file being used for storage, the amount written in the file (number of records), the maximum size of the file

(number of records), and the number of entries that have been written into the file. The instruction consists of "DISKSUMMARY" followed by a comment if desired. No printer output is generated. For example,

```
: DISKSUMMARY←

DISK SAVE FILE IS GROUP/DEMO
27 OF 1000 RECORDS HAVE BEEN WRITTEN
13 ENTRIES ARE STORED
```

Cleanup Instruction: This instruction is mainly used for debugging purposes. It causes all storage syllables not in use for the current presentation to be marked "never used" (see Chapter IV). This instruction consists of "CLEANUP" followed by any desired comment.

Dump Instruction: This instruction is used mainly for debugging purposes. It causes the program to print a dump of the switch settings, presentation pointers, main storage arrays, and current chain (see Chapter IV). The dump is labeled

DUMP TAKEN AT USER REQUEST

in addition to the comments (if any) in the instruction. The format of the instruction is "DUMP" followed by comment (if desired).

Abelianize Instruction: This instruction causes specified relations to be abelianized. If all the relations are to be abelianized, the entire instruction is "ABELIANIZE". Otherwise, the instruction consists of "ABELIANIZE" followed by a relation list specifying those relations that are to be abelianized. If the first form is used, a transformation entry is written into the disk save file showing the operation. After the operation is completed, redundant relations are listed and the presentation is written into the disk save file, typed and printed.

Examples:

```
: ABELIANIZE RELATIONS 1, 3 AND 4←
: ABELIANIZE←
```

Time Instruction: This instruction, consisting of "TIME" followed by comment, causes the program to type the amount of central processor (CPU) and input/output channel (I/O) time that has been used by the program run. No printer output is generated by this instruction.

Example:

```
: TIME←

CPU TIME IS   2 MIN 20 SEC
I/O TIME IS   0 MIN 27 SEC
```

Suspend Instruction: If a disk save file is needed that is not available in disk storage, it may be necessary for the user to free himself from the GROUP program in order to retrieve it for use. If he wishes to do this without terminating the current program run, he may cause the program to suspend itself through the Suspend Instruction. A suspended program remains "alive" but is allowed no computer time for execution for the duration of the suspension.

The format of the instruction is "SUSPEND" followed by a positive integer specifying the duration of the suspension in minutes. Any desired comment may follow this integer. The program then types

SUSPENDING FOR n MINUTES

and instructs the B-5500 operating system to suspend execution. For example:

```
: SUSPEND 3 MINUTES←
SUSPENDING FOR 3 MINUTES
```

At the conclusion of the suspension, the program types:

GROUP PRESENTATION PROGRAM - SUSPENSION ENDED

If the program is unable to re-establish contact with the user, it terminates the current program run. No printer output is generated by this instruction.

Quit Instruction: This instruction, consisting of "QUIT" followed by any desired comment, causes the program to type the amount of central processor and input/output channel time used (as in the Time Instruction), followed by termination of the program run. No printer output is generated.

Chapter IV. Program Structure

In this chapter, we describe some of the programming aspects of the GROUP program. This is intended for a person who intends to extend the program by making additions to it or who intends to adapt the program to a computer other than a B-5500. Also, it may be useful in interpreting diagnostic dumps, should they occur. We discuss storage allocation, the internal representation of the various structures that comprise a presentation, input/output systems and capabilities, and program capabilities for manipulation of words and presentations. In addition, a few special topics are briefly considered.

1. Defined and Compilation Parameters

The sizes of the storage arrays and disk save files are specified at compilation of the program by the use of the B-5500 ALGOL "define" construct to establish constant values for certain identifiers. These identifiers, their values in the current version of the program (version 9), and a brief description and/or reference to a description are given in Table 2. The arrays are declared in such a way that the sizes may be changed by changing the value of these constants, with no other changes necessary. Certain defined constants that are associated with the hardware, such as the printer, are discussed in later sections and do not appear in Table 2.

When the program is compiled, certain parameters are specified to the B-5500 operating system to control the compilation and the restrictions that the system places upon the program when it is run. These values are given in Table 3. For a more complete description of the meaning of these parameters, the user should refer to the User Manual for the B-5500 (at the University of Wisconsin, volume VII of the UWCC User Manual).

identifier	value*	description and/or reference
MAXGEN	60	maximum number of generators in a presentation (page 60)
MAXRELN	60	maximum number of relations in a presentation (page 61)
NSECTN	30	control the size of the bulk storage arrays (page 59) NOTE: MAXSYL must be defined to be NSECTN × SECLNGTH
SECLNGTH	100	
MAXSYL	3000	
AREASIZE	50	control the size of the disk save files (page 63) NOTE: The B-5500 operating system restricts the value of NUMAREAS to be no greater than 20
NUMAREAS	20	
MAXCHAINLENGTH	150	The maximum number of entries in a chain may be no greater than MACHAINLENGTH + 1 .

*The values given are those for Version 9 of the program

Table 2

Defined Constants.

compilation parameter*	value	meaning
XALGOL STACK	700	700 words are reserved for the ALGOL compiler's stack
XALGOL PROCESS	10	The time limits for the compiler in minutes. (These values are 3 to 4 times the usual requirement)
XALGOL IO	20	
PROCESS	99	The time limits, in minutes, for program execution. (These values are infinite for most practical purposes)
IO	99	
CORE	6000	The amount of core storage space to be used by the program (in words)
LINES	1000000	The maximum number of print lines that the program may produce (this value is infinite for most practical purposes)

*The B-5500 Compatible ALGOL compiler at the UWCC is named "XALGOL"

Table 3

Compilation Parameters

2. Storage

In this section, we give a description of the storage used by the GROUP program, both core storage arrays and disk save files. The storage system was designed with two objectives in mind: (1) to make the program as efficient as possible in terms of both real time and computer time, and (2) to make the system as easy as possible to program. It is difficult to judge the effectiveness of the programming in terms of the first objective because of the irreproducibility of the program environment in a multi-programming system. In terms of ease of programming, certain features have been found to be cumbersome, but in general it has been found relatively easy to make additions and changes to the program.

Syllable and Word Storage: The bulk of storage for syllables and words is provided by the arrays BASE and EXPONENT, which hold bases and exponents respectively. Individual syllables may be held in temporary locations provided by the procedure or block that is operating on a presentation. The arrays BASE and EXPONENT are both two-dimensional arrays and are declared to have limits [0:NSECTN-1, 0:SECLNGTH-1], and provide room for a maximum of MAXSYL = NSECTN × SECLNGTH syllables (the value of these three constants may be found in Table 2). MAXSYL is defined separately from NSECTN and SECLNGTH for program efficiency since the compiler would otherwise cause the product to be computed each time that MAXSYL is referred to, which is quite often. The arrays are made two-dimensional in order to overcome the B-5500 limitation restricting linear (one-dimensional) arrays to 1023 elements.

For all purposes except actual references to elements in the

arrays, these arrays are considered to be linear and numbered from zero to MAXSYL-1. The linear order is related to the actual subscripts as follows:

section = lexpos div SECLNGTH

word = lexpos mod SECLNGTH

where lexpos is the lexicographical (linear) position of the subscript and [section, word] is the actual subscript of the syllable. A global macro-instruction POSITION is available to convert from lexicographical index to actual index. A second global macro-instruction INCREMENT is provided to advance the actual index by one lexicographical position. These macro-instructions are declared in the initial portion of the program.

A word is generally referred to by a position vector (start, length) where "start" points to the first syllable in the word (linear order), and "length" is the length of the word in syllables. Thus such a word occupies "length" consecutive positions in the BASE and EXPONENT arrays, beginning in syllable number "start". Note that a word may lie in more than one "section" of the arrays. If "length" is non-positive, then the word is taken to be the identity word, 1 .

Presentation Storage: The generators of a presentation are stored in the array GENERATOR, which is one-dimensional and declared [1:MAXGEN]. Thus a maximum of MAXGEN (see Table 2) generators may appear in any presentation which the GROUP program is asked to handle. An attempt to use more will result in a diagnostic error message. The number of generators in a presentation is held in the global variable NUMGEN. The generators appear in the first (lowest subscripted) elements of the array. Generators are

usually stored as positive integers with no more than six characters (right-justified and zero-filled), but they may occasionally be made negative (see the discussion of Preserve and Release Instructions).

The relations of a presentation appear as words in the BASE and EXPONENT arrays. Their position vectors are held in an array of pointers, LOCATION, which is two-dimensional and declared [1:2, 1:MAXRELN] (see Table 2 for MAXRELN). If the i-th relation in the presentation has position vector (start, length), then these are stored in the following manner:

LOCATION [1, i] = start

LOCATION [2, i] = length .

The number of relations in the presentation is contained in the global variable NUMREL. It should be noted that the position vectors of the relations need not appear in the same order in the LOCATION array as the relations appear in the syllable storage arrays (usually they do not).

The name of the current presentation is contained in three global integer variables: CURRPREFIX, CURRSUFFIX, and CURRPRESNO. These hold respectively the prefix and suffix of the disk save file containing the presentation, and its entry number in that file. The contents of these variables are controlled by the disk file input/output procedures handling the storage and retrieval of presentation storage entries, and, to a lesser extent, by the procedure READPRESENTATION which reads a presentation typed by a user on a remote teletype.

Disk Save Files: In this section, we discuss the organization and handling of the disk save files and the entries contained in them. The reader should see Chapter III, General Concepts (Disk Storage, and Chaining), for a discussion of the uses of the disk save files. We will discuss this topic in the following order: types of entries, organization of the files, organization of entries, and handling of the files.

Types of Entries: Each entry in a disk save file is of one of five types. The types are numbered from zero to four and are named according to the following table:

type number	type name
0	presentation storage entry
1	replacement entry
2	remove generators entry
3	abelianize entry
4	new relation entry

The entry types 1 through 4 are the transformation nodes referred to in Chapter III (Chaining). Each entry in a disk save file contains its type number, a pointer to the entry from which it was obtained (the chain pointer), and the additional information that it requires to describe the transformation or data.

We now give a brief description of each type of entry and the additional information that it contains and will leave a detailed account of the format of the entry to a later section. A presentation storage entry contains a complete presentation and is the storage node referred to in Chapter III. A replacement entry is generated when one word is replaced by another, as in

solving for a generator or in a Replace Instruction (see Operating Instructions), and contains the two words. A remove generators entry is created when generators are removed from presentation other than by solving for a generator, and it contains the list of generators so removed. An abelianize entry specifies that the presentation (i. e., all relations) was abelianized and requires no further information. A new relation entry is created when a new relation is added to a presentation and contains that relation.

Organization of the files: Each disk save file created by the GROUP program has a maximum of AREASIZE × NUMAREAS (see Table 2 for these values) records of thirty words each. A thirty word record length was chosen because the B-5500 operating system (Master Control Program or MCP) allocates disk space in units of thirty words. NUMAREAS is the number of areas that the file may contain and is restricted by the MCP to be not greater than 20. When the program creates a disk save file, the MCP allocates one area of disk space containing AREASIZE records. When that area is filled, it allocates another area of the same size, and continues in this manner up to the maximum number of areas declared, in this case NUMAREAS.

The records in a disk save file are numbered beginning with zero. Since an entry may require more than one record to store all the necessary information, a directory is used to locate information in the file. The first record of the file (numbered zero) is always a directory record. It contains the beginning record numbers of the first 29 entries in the file (or part thereof). If there are more than 29 entries in the file, the last word is the record number of the next directory record, which holds the beginning record numbers of the

next 29 entries (or part thereof); and so on. It should be noted that this is the only structure imposed on the file aside from the fact that each entry occupies consecutive records beginning with the record given in the directory. In fact, the second and subsequent directory records follow the first entry to which they point. All unused words in a directory record are zero.

Organization of the Entries: Each entry consists of as many consecutive records in the file as are necessary to contain the information to be stored. No entry is broken between two files. If, while storing the information for an entry, it is found that there is not enough space remaining in the file, a new disk save file is created and the entry is stored in the new file. For this reason, presentations are not named until they have been stored in a disk save file.

The first four words in each entry have the same meaning. The first word contains the type number of the entry (see p. 62) and is followed by the chain pointer, the name of the entry from which the current entry was obtained. This name requires three words which contain, respectively, the prefix and suffix of the name of the file containing the predecessor entry and the entry number of the predecessor. Should these words contain "?", "?", and 0 (respectively), then there is no previous entry; i. e., this entry is the "root" of the tree.

Information is stored within an entry without regard for record boundaries; i. e., the records comprising an entry may be regarded as being "strung end-to-end" and proceeding from left to right. Lists of generators are stored by giving first the number of generators in the list and then the elements

of the list (if any). Words are stored by giving the length of the word in syllables followed by the syllables of the word, each base immediately preceeding its exponent. If the length of a word is not positive (i. e., the word is ± 1), then the length is all that appears.

A presentation storage entry contains the table of generators of the presentation in the format of a list (as above). If the number of generators is not positive, then the presentation is taken to present the trivial group and no more information is stored. If the number of generators is positive, the number of relations comes next. If the latter is not positive, the group presented is free and no further information is required. If the number of relations is positive, each relation appears as a word (as described above) in the order that the pointers appear in the LOCATION array.

A replacement entry is generated by a transformation which replaces all occurrences of one word in the free group with another word. After the chain pointer in the entry, these two words from the free group appear; first the word that was replaced and then the word that replaced it.

The remove generators entry indicates that a list of generators has been removed from the GENERATOR array and from each relation in which they appeared. The entry contains this list in the format described above for a list.

The abelianize entry indicates that each relation in the presentation has been abelianized. No other information is stored in this type of entry except for the type number and chain pointer.

The new relation entry specifies that a relation has been added

to a presentation. The entry contains this relation in the format described for a word.

Handling of the Files: For storage in and retrival from a disk save file, sets of bottom level procedures are provided to handle the data on a one-machine-word-at-a-time basis refilling the buffers as necessary without the intervention or knowledge of the program section using the data. Sets of second level procedures are provided to handle data in several sturctures, such as words (since we will refer to both machine words and words of a free group, we will henceforth distinguish them by underlining the latter).

For output, the program section must first save the record number of the first record to be written. This is available as the global variable NEXTADDR and will be passed to the directory building procedure after the entry has been written into the file. (The directory is written last to provide for the possibility that the entry may not fit into the current file.) The program then prepares the entry by calling procedure PLACECHAINDATA which performs the following actions: opens the internal (declared) file, places the type number (one of the arguments), sets the chain pointer, and initializes the buffer. Information is then added to the file by calling either of the following procedures:

procedure	action
ADDTOFILE	adds one word of information
WRITEWORD	adds one word to the file

After all information has been passed to the file through these procedures, the

directory entry is prepared by calling procedure WRITEDIRECTORY and providing it with the first record address that was saved initially. Procedure WRITEDIRECTORY writes the last buffer of information into the entry, prepares the directory information creating a new directory record when required, and maintains the information for creating chain pointers. Each of these procedures has as an argument the identifier of the internal (declared) file. Those procedures that store information in the file also have as an additional argument a "failure" label. Control is transferred to this label if the entry is not able to fit into the file. In this case, a new disk save file is created by the bottom level procedures and the entry is stored by restarting the program section.

To recall entries from disk save files, a set of procedures is provided that examine files and entries for existence and type, as well as a set that recall various components of an entry. These procedures and their actions are:

procedure:	action:
EXISTSFILE	determines if a file exists on the disk and if the program may use it
LASTPRESENTATION	locates the last presentation storage entry in a disk save file
NEXTFROM	returns the next word from the disk save file
READDISKPRESENTATION	reads a presentation storage entry
READDISKWORD	reads a word from a file
TYPEOFENTRY	determines the type of an entry and sets up the input buffer.

The usual order of input is to first check that a file exists on the disk and that the program may use it, issuing an error message if either of these requirements is not met. Then the entry is checked to verify that it is of the type required, with an error message if it is not. Finally the data in the entry is read and processed as desired. Reading a presentation storage entry also entails the setting of the name of the entry as the name of the current presentation (see Presentation Storage).

3. Storage Allocation

Allocation of bulk syllable storage in the GROUP program is performed by specifying a point in the storage which separates "in use" space from "free" space, and moving this pointer as the amount of storage "in use" changes. Each procedure in the program that has need of temporary or permanent storage (or references such a procedure) has as an argument a pointer to the beginning of "free" space. Such a procedure that must modify a relation or other word to be retained and that is unable to store the modified version in the original space occupied by the word places the modified version in free storage and changes the pointer to the new beginning of free space.

It is easily seen that such a method of storage allocation can easily exhaust the amount of space available if the relations have any tendency to grow in size (which they usually do). This could have been avoided by the use of more sophisticated list processing techniques of storage allocation. These techniques, however, have the disadvantage that information tends to become scattered throughout the sectioned arrays, thus increasing the amount of overlaying that the system must perform to obtain information. In the GROUP

program, enough space was provided to (hopefully) contain the results of a single instruction and "garbage collection" is periodically performed. The garbage collection consists of packing all the relations into the beginning portions of the storage arrays and adjusting the pointers accordingly. Garbage collection is performed after each instruction that could have "unpacked" the storage and in the middle of a few instructions where it was felt to be a necessary precaution. This garbage collection, performed by procedure CONDENSE, can only be performed when the only information to be retained is the presentation relations.

This method of allocation has caused storage space to be exceeded only once in an actual problem. This problem consisted of a very large presentation (a knot group of 58 generators and 58 relations) and storage space was exceeded after a large number of simplification steps had created a number of extremely large relations. This experience showed that Simplification Steps interspersed with Containment Operations seemed to be the best way to avoid presentations with large unmanageable relations.

4. Free Format Input Package (RDNUM)

This section describes the free format input package used by the GROUP program for processing input data. This routine is similar in operation and structure to a FORTRAN subroutine of the same name described by A. Hassitt [2]. This version of RDNUM differs from that of Hassitt in that it makes more information about the input available to the main program and permits more main program control of the operation of the routine. The package is described in some detail because it can be used in a wide variety of programs. For

general use, however, some modifications in the tables would probably be desirable. Some possible changes are discussed later.

Philosophy: The basic philosophy of RDNUM is based on two premises: (1) that a user's input should be interpreted in a "reasonable" manner, and (2) that as far as possible the user should be able to decide what is "reasonable". For these reasons, the package was written with all variables that might be useful to the main program declared globally. Also, in order to avoid the necessity of reinitializing the arrays holding the decision table and the character types each time that the procedure was called, these also were made global and were initialized by requiring the main program to call an initialization procedure before using the package. This had the unanticipated benefit of allowing the main program to change to definition of "reasonable" during the course of the computation.

Operation: The operation of RDNUM can be divided into two sections, the recognition section and the decoding section. In the recognition section, the location of the next string (i. e., the next real number, special symbol, alphebetic symbol, etc.) and its type (real number, special symbol, etc.) is determined. The operation of the recognition section is somewhat complicated to describe, thus the algorithm is given at the end of this section. When the location and type of the string is determined, control is passed to the decoding section where the usual form of the string is generated according to the type of the string (i. e., real numbers are converted to floating point binary numbers, alphabetic symbols are packed into words right-justified and

zero-filled, etc.). In case errors are detected in the recognition section, the decoding section may be bypassed.

In the recognition section, the action under certain circumstances is determined by the "mode" of the program. The recognized modes are: "standard" and "remote". In the "standard" mode, the "no input" response from RDNUM occurs only if an attempt to obtain a new input image is unsuccessful. In the "remote" mode, this response is also caused by an input image containing only blanks. The GROUP program operates in the "remote" mode.

In the recognition section, the variable STAGE is obtained from the decision table in a manner described more fully in the next section ("Action of the Decision Table"). STAGE has the property that it is zero if and only if no nonblank characters have been encountered in the current string. In addition, certain values indicate that the end of the string has been found (Step 12). The algorithm of the recognition section is as follows:

1. Set the variable STAGE to zero. If the "end of card" flag is not set, skip to step 5.
2. Try to obtain a new input image. If the attempt is successful, skip to step 4.
3. (Unsuccessful attempt to obtain a new input image) Set the "end of card" flag, return a "no input" response, and exit from RDNUM.
4. (Successful attempt to obtain new input image) Set the character pointer to the first character of the new input image, and reset the "end of card" flag.
5. Is the character pointer pointing to a character in the input image? If so, skip to step 11.

6. Set the "end of card" flag. If the character pointer is pointing to the first character position past the end of the input image, skip to step 8.

7. (Character pointer is pointing past the first character after the end of the input image) Store the length of the string scanned (variable RDLONG), return an "illegal symbol" response, and exit from RDNUM.

8. (Character pointer is pointing to the first character after the end of the input image) If non-blank characters have been found in this scan (i. e., if STAGE $\neq$ 0), then skip to step 10.

9. (Only blanks have been found in this scan) If the "remote" mode is set, then return a "no input" response and exit from RDNUM, otherwise go back to step 2.

10. Assume that the next character is a blank and skip to step 12.

11. Pick up the next character (the one that the character pointer is pointing to). If this character is the first non-blank character found in this scan (i. e., the character is not blank but STAGE is still zero), then save the position of the character (variable RDBEGIN) as the beginning of the string to be decoded.

12. Obtain the new value of STAGE from the decision table, and advance the character pointer to the next character position. If the new value of STAGE indicates that the end of the string has not been found, go back to step 5.

13. (Decoding section) If the value of STAGE is odd, backspace the character pointer to the preceding character. Determine the type of the string from the value of STAGE, decode the string, and exit from RDNUM.

It should be noted that the "end of card" flag (Boolean variable RDLAST) is set when RDNUM is initialized. In the GROUP program, this is done at the beginning of the program run.

Action of the Decision Table: To begin, the program initializes the local state variable STAGE to zero. The program then examines the next character (determined by RDPOS) and determines its type from array RDCTYPE (Table 4) and enters the decision table RDECIDE (Table 5) with the current value of STAGE and the character type and determines the next value of STAGE. This scan continues until a termination value of STAGE ($\geq$ 54) is obtained.

If the termination value is odd, then the character pointer RDPOS is backspaced. The reader should easily be able to convince himself that the pointer should be backspaced most of the time since the end of a symbol is usually recognized by encountering a character that does not belong to it. The type of symbol encountered, i.e., the value of RDTYPE, is found by dividing the termination value of STAGE by 2, discarding the remainder, and subtracting 26 (see the bottom of Table 5).

For example, the string __-1.2_ (underlines indicating blanks) goes through stages 0, 0, 1, 4, 5, 5, 57 indicating that a real number has been encountered. Similarly the alphabetic symbol _X12_ goes through stages 0, 9, 9, 9, 61.

The values in the table of character types (Table 4) were designed to fit the requirements of the GROUP program and should probably be changed for general use. In particular, "E" can be permitted as an exponent symbol, and the collection of letters should probably be restricted to the members of

the alphabet. Also, the decision table (Table 5) can be changed if desired. One should take care in making any changes in these tables that strings yielding the types integer or real number be readable via the "R" format of B-5500 ALGOL.

Global Variables and Arrays: The global variables and arrays can be divided into three groups: (1) those describing the last symbol returned by the routine, (2) those describing the progress made in decoding the current input string, and (3) those that control the operation of the package. These are all integer variables or arrays unless otherwise noted.

The variables that discribe the last symbol returned are RDTYPE, RDBEGIN and RDLONG. RDTYPE is the type of the symbol and is given by the following table:

RETYPE	symbol type
1	integer
2	real number
3	special symbol
4	alphabetic symbol
5	illegal symbol
6	no input (depends partly on RDREMOTE)

Except for no input, these types depend only on the decision procedure (Tables 4 and 5) and the form of the symbol. They do not depend on the type of variable in which the decoded value is stored (see "Use of the Package"). RDBEGIN and RDLONG respectively give the first character location and the number of characters in the symbol. The first character of the input string is numbered zero, hence RDBEGIN can be interpreted as the number of characters preceding the symbol.

The variables describing the progress made in decoding the input string are RDLAST and RDPOS . RDLAST is a Boolean variable that has the value true if the current input string has been finished. If RDLAST is true, the routine will fetch a new input string upon the next call, hence this may be used as a control variable if desired. RDPOS is the character pointer and points to the first character to be examined on the next call to RDNUM. As with RDBEGIN, RDPOS can be interpreted as the number of characters preceding the next character to be examined. If RDLAST is true, then the value of RDPOS has no meaning.

The control variables are RDSTRINGMAX, RDMAX and RDREMOTE, as well as the table of character types (RDCTYPE) and the decision table (RDECIDE). RDSTRINGMAX controls the truncation of alphabetic and special symbols by specifying the maximum number of characters to be stored in a word. RDSTRINGMAX must be positive and less than 8 since B-5500 words have 47 usable bits and each character requires 6 bits; a recommended value is 6 , although 7 may be used with real variables. RDSTRINGMAX is set to 6 by the initialization procedure, but may be changed by the program if desired. RDMAX and RDREMOTE must be set by the main program before using the RDNUM package. RDMAX specifies the maximum number of characters in an input string (see "Use of the Package" for a discussion of how this may be set). RDREMOTE is a Boolean variable which specifies whether the routine is to operate in the "remote" mode. In the "remote" mode (RDREMOTE = true), input strings consisting only of blanks are recognized and generate a "no input" response (RDTYPE = 6); otherwise they are ignored.

In a class by itself is the array RDBUF used to hold the input string. The package has its own array for this purpose since the input buffers might be used for input not passing through the RDNUM package.

Use of the Package: In order to use the RDNUM package, the user must provide a procedure RDFETCH that obtains input strings and places them in the array RDBUF. If no input string is available, the global variable RDTYPE should be set to 6, causing the package to return a "no input" symbol. The user may elect (1) to have RDFETCH set RDMAX (the maximum number of characters in an input string) to the number of characters in the input string, or (2) to set RDMAX once and have RDFETCH fill out shorter input strings with trailing blanks (in particular, the group mark ending teletype input messages must be removed). Of course, RDMAX must never be larger than the size of RDBUF. In the GROUP program, the second choice was elected.

Before using the package, the main program should set RDREMOTE to specify whether input strings containing all blanks should be ignored (see "Global Variables and Arrays"), and must call the initialization procedure RDINITIAL. The main program should also set RDMAX if the second choice above was elected. If the value of RDSTRINGMAX set by the initialization procedure is not desired, the main program should also reset it.

After completing these preliminaries, a symbol may be decoded by either of the following statements:

RDNUM (symbol)

RDINT (symbol)

The form used depends on whether symbol is a real or integer variable respectively. If the symbol decoded is an integer or a real number (RDTYPE = 1 or 2) then C(symbol)† is the usual representation of that integer or real number that appeared in the input string. If the symbol is a special or alphabetic symbol (RDTYPE = 3 or 4) then C(symbol) contains the first (i. e., leftmost) n characters of the symbol in the input string, where n is the smaller of RDLONG and RDSTRINGMAX. These characters are right justified and zero-filled in C(symbol). If the symbol is illegal (RDTYPE = 5) then C(symbol) is zero. If the symbol is "no input" (RDTYPE = 6), then C(symbol) is undefined; that is, unchanged by the package.

When using any package, such as RDNUM, in a program, there is the problem of duplication of identifiers. With this in mind, RDNUM was written so that the first two characters of each global identifier are "RD".

Useful Techniques: The straightforward use of RDNUM described in the last section sufficed for nearly all the input to the GROUP program, but in a few circumstances it was found useful to "jigger" with the routine to cause non-standard decoding. We will describe the modifications which were used in this program and leave it to the ingenuity of the user to find ways to twist the routine to his own purposes.

In some cases it may be found desirable to suspend the processing of one input string, process another, and then return to the first at the point where processing previously left off. This was the case when the

†If var is a variable, then C(var) will represent the contents of the machine word in which var is stored.

type	type no.	characters
digits	0	0 1 2 3 4 5 6 7 8 9
decimal point	1	.
signs	2	+ -
exponent character	3	@
letters	4	A B ... X Y Z &× $ * ; / % "
blank	5	
special characters	6	# : , ( ) [ ] > ≥ < ≤ = ≠ ←
Illegal character	7	?

TABLE 4

RDNUM Character Types as used in GROUP program

Type of Next Character

Current State of Routine (Stage)	0	1	2	3	4	5	6	7
0	4	2	1	9	9	0	58	62
1	4	3	59	59	59	59	59	62
2	5	59	59	59	59	59	59	62
3	5	62	62	62	62	62	62	62
4	4	5	55	6	55	55	55	62
5	5	57	57	6	57	57	57	62
6	8	62	7	62	62	62	62	62
7	8	62	62	62	62	62	62	62
8	8	57	57	57	57	57	57	62
9	9	61	61	9	9	61	61	62

Termination Values

no backspace	backspace	symbol type	RDTYPE
54	55	integer	1
56	57	real number	2
58	59	special symbol	3
60	61	alphabetic symbol	4
62	63	illegal symbol	5

Table 5

RDNUM Decision Table

program was reading a word and encountered an acceptable generator that was not a generator. The processing of the word was stopped and RDNUM was used to process the answer to the question of whether to make the acceptable generator a generator. To accomplish this, it was necessary to save the data describing the input data and the state of the processing; that is, RDBUF, RDPOS, and RDLAST. If RDMAX had depended on the input string, then it too would have to be saved.

To read disk save file names, there arose the necessity of accepting strings containing any characters except blanks, "/" or "-". This was accomplished by changing the table of characters types (RDCTYPE) so that all characters, except the illegal character and those mentioned above, were considered letters (type 4). Thus the parts of the name were returned as alphebetic symbols.

In reading presentation names, it was occasionally found necessary to reread symbols under a different decoding scheme. This occurred when reading abbreviated forms of the names, since the first symbol encountered could equally well have been a file name prefix or suffix, or a presentation number. Since the location of the beginning character of the symbol was available, it sufficed to set the appropriate table of character types, reset RDPOS to RDBEGIN, and call RDNUM again.

5. Free Format Output System

In order to provide the most readable output in normal mathematical notation (whenever feasible), the output of the GROUP program has been designed so that the programmer can take complete control of all the output if he desires. To make this easier, a collection of procedures have been written to handle the bulk of the output. As may be expected, the coding of this system is strongly dependent on the character manipulation facilities offered by the compiler.

Integer procedures (functions) that are used by all parts of the program to prepare output are INTSIZE and LITSIZE. These procedures respectively determine the number of characters necessary to contain an integer when it is expressed in decimal notation, and determine the number of characters stored in an integer variable (right-justified and zero-filled).

Teletype Output: Teletype control is provided by two procedures, INPUT and OUTPUT, which were adapted from routines provided by the University of Wisconsin Computing Center. These routines, respectively, read data typed at the teletype and type program responses on the teletype. The results of INPUT are handled by RDNUM. For much of the output done to the teletype the formatting desired can be obtained through format declarations with variable field lengths that are provided in B-5500 ALGOL .

The output of words from free groups, however, requires more work because such a word may extend beyond a single line. To provide the bookkeeping necessary to advance lines and indent properly, two procedures are available: These are TYPEUNIT and TYPESTRING. Both of these procedures

output strings of characters to the teletype. TYPESTRING does not bother with the problem of spacing and is only a convenient method of moving a string to the teletype buffer and typing it. TYPEUNIT, on the other hand, handles the spacing bookkeeping. Before it is called, however, the user must set the global variable TYPESPACE to number of spaces remaining in the current teletype line (this may be non-positive if a new line is desired immediately). On each call to TYPEUNIT the user provides a string of characters and the number of spaces the procedure should indent after advancing the paper (double spaced) should the string be too long to fit in the current line. If there is not enough room after such an indention, the procedure will break the string into pieces small enough to fit, indenting as specified before each piece. After a call to TYPEUNIT, TYPESPACE is the number of spaces remaining in the line. If TYPESPACE = 0 , the paper is not advanced. This makes the coding easier in that the type unit is always in the "middle" of a line after any use of TYPEUNIT.

A third procedure TYPELINE is provided to advance the teletype paper by a given number of lines. The type unit is positioned to the beginning of the line and TYPESPACE is set to 72 (the number of spaces in a complete line) after a call to TYPELINE.

A major failing of the coding of this system is that the number of spaces per line was not made a defined constant. Instead, the known value of the length of the line (72 characters) was used in writing the code. This resulted in somewhat more efficient code, but makes it difficult to change to a system with a different number of spaces per line. The author believes that all occurrences of this type can be found in statements involving TYPESPACE

and hence can be found by a compiler.

Printer Output: The printer control system is more elaborate than that for the teletype. In addition to keeping track of the position in each line, it is also necessary to keep track of the number of lines printed in the page and to provide for changes in the maximum width of the lines. The maximum sizes are defined constants and their values (in version 9) are given below:

identifier	value
PRINTBUFFERSIZE	132
PRINTWORDS	17
LINESPERPAGE	81

PRINTBUFFERSIZE is the number of spaces available on the hardware line printer; the B-5500 is available with either a 120 or a 132 space printer. PRINTWORDS is the number of words necessary to hold PRINTBUFFERSIZE characters and is equal to

PRINTWORDS = (PRINTBUFFERSIZE + 7) div 8

for an 8-character-per-word machine. LINESPERPAGE is the number of single spaced lines per page. The declarations and computations are independent of the system except for these defined constants.

For bookkeeping purposes concerned with the number of lines remaining in the page (global variable PRINTLINES), three macro-instructions are provided: NEWPAGE, SKIPLINE, and SKIPTWOLINES. The functions of these macro-instructions are given by their names (the latter two refer to single spaced lines).

The print control system is initialized by calling procedure

PRINTINITIAL and specifying the number of spaces on the printer to be used. This procedure should be called whenever the maximum number of spaces changes (e. g., Print Width Instruction).

To provide for raised exponents in the printed output, two printer buffers, PRINTBUFFER and UPPERBUFFER, are used. The first is used for the bulk of the output, with the second being used only for exponents. A Boolean variable PRINTUPPER specifies whether there is currently any information in the UPPERBUFFER and is controlled by procedures PRINTLINE and PRINTSYLLABLE. If PRINTUPPER is true, then each procedure that places information into PRINTBUFFER must also place the same number of blanks into UPPERBUFFER to maintain the proper positioning of pointers.

The various procedures and their functions are described below. Except where noted, at the return from each procedure, the print buffer pointers are in the "middle" of the line; that is, the line is not usually printed unless (1) specified by the main program or (2) not enough room is available in the current line to hold the output string to be stored in the buffer.

Procedure PRINTLINE prints a line of information when any part of the system or program requires the buffers to be emptied or reinitialized. Carriage control is effected through the global integer variable PRINTCARRIAGE which may have the values 1, 2, or 3. Upon entry to PRINTLINE, PRINTCARRIAGE - 1 blank lines are printed, thus effecting single, double or triple spacing respectively. Before printing, the buffers are filled out with blanks. If PRINTUPPER is true (i. e., there are raised exponents to be printed), the line count is checked to make sure that there is enough room remaining on the

page to hold both lines of print (exponents and bases). If there is enough room, an additional blank line is printed for greater legibility; otherwise, the printer is skipped to the next page. After printing, the pointers are set to the beginning of the line and PRINTUPPER is set false.

To print a string that is required to fit on a single line, procedure PRINTUNIT is used. If there is not enough room to fit the string on the current line, the procedure causes the current line to be printed and indents the next line by an amount specified in the calling parameters. Then the string is inserted into the input buffers. Should there not be enough room in the new line, the string is broken between lines until it is all printed or stored in the buffer (each additional line is indented as specified).

Syllables are printed with raised exponents by calling procedure PRINTSYLLABLE. This is the only procedure in the system that stores non-blank information in the UPPERBUFFER. Indentation is specified in the same manner as in procedure PRINTUNIT. However, it is assumed that the syllable will fit on one line, if not the current line then the next line after indention. The procedure takes the following steps: (1) Determines whether the syllable will fit on the current line and, if not, causes the current line to be printed and indents the new line as specified in the arguments. (2) Places the base in the line and, if PRINTUPPER is true, places the same number of blanks in the UPPERBUFFER as there are characters in the base. (3) If the exponent is not 1, then it (3a) fills the initial portion of the UPPERBUFFER with blanks and set PRINTUPPER to true if PRINTUPPER was initially false, (3b) places the exponent in the UPPERBUFFER, and (3c) places the same number of blanks in the PRINTBUFFER as

there are characters in the exponent. (4) Finally, the procedure adjusts the count of the number of spaces that remain in the line. Note that a line containing syllables causes the UPPERBUFFER to be printed if and only if at least one of the syllables in the line has an exponent that is not 1.

Blank spaces may be inserted into the printed line by use of procedure PRINTSKIP. The blanks are inserted beginning at the current position of the pointers and are continued into succeeding lines (without indention) if the number of blanks specified overflows the line.

Procedure PRINTSTRING causes a string to be printed. This procedure causes the line to be printed at the end of each call (this is an exception to the general rule noted above). The string requested is truncated if necessary to fit it in the current line.

Procedure PRINTTOMARK is similar to procedure PRINTSTRING except that the string continues to a group mark "↩" or the end of the current line, whichever comes first. This procedure also causes the line to be printed at each call.

6. Special Syllables

The GROUP program uses two special syllables that cannot be typed into the program by the user and should not appear in any output of the program except for a dump of the storage arrays. These syllables are $\underline{:}$ and $\underline{::}$. These syllables were chosen because the character ":" is defined in RDNUM as a special character and generators must be alphabetic symbols (as defined in RDNUM). Thus symbols of this type may be used by the program without infringing upon any symbols employed by the user of the program.

The syllable : is used to mark the storage spaces that have not been used by the program. When printing a dump, they show the necessary extent of the dump by scanning the arrays from the end toward the beginning until a syllable not : is found. They may also be useful in debugging if it suspected that spurious syllables are creeping into the computation through errors in computing the position vectors of words. The Cleanup Instruction causes all syllables not being used to store the current presentation to be set to :. In addition, the arrays are initialized to this state at the beginning of each program run.

The syllable :: is used as a temporary syllable when making substitutions of one word for another. In the Replace Instruction (see Chapter III), for example, occurrences of the word to be replaced are "marked" by this syllable and then replaced by the new word. The marking is performed to eliminate the possibility of the program looping, as was indicated in the discussion of the Replace Instruction. This is the only use that is currently made of this syllable.

7. Diagnostic Dump

Diagnostic dumps are produced automatically by the GROUP program when it finds that there is not enough room in the syllable storage arrays to hold the information needed. Diagnostic dumps can also be produced by the user through use of the Dump Instruction (see Chapter III). A description of the dump is given below.

If the dump is caused by exhaustion of the storage arrays, an error message will be typed on the remote teletype and the dump will be

preceded by the message

OPERATION ABANDONED BY "message"

where the "message" gives the procedure or program section where the exhaustion was detected.

The dump procedure DUMPS may be called for debugging purposes from any part of the program. The procedure has three parameters, ident, length, and howfar. The first two arguments specify an identifying message which is printed in the form

DUMP TAKEN AT "message"

where "message" is the first length characters of the array row ident [*]. The final argument, howfar, specifies the number of syllables of the storage arrays to be printed in the dump (see below). In all cases that are built into the program, howfar is the larger of ten and the number of syllables that have used by the program (as indicated by the syllable : ; see Special Syllables). An example of a diagnostic dump is given in the sample program run (Appendix C).

A diagnostic dump is indented into the printer page 30 spaces and uses the remainder of the page up to the maximum number of characters available (PRINTBUFFERSIZE). Any smaller limits imposed by the user are reestablished upon exit from the dump procedure. The diagnostic dump consists of the following parts:

(1) The identifying message described above.

(2) The name of the current disk save file.

(3) The name of the current presentation. Since the program may be in the middle of an operation on a presentation when the dump is taken,

this name should be interpreted as the name of the presentation from which the operation started.

(4) The settings of the program switches CONTAINSWITCH, DETAIL, TYPEENABLE, TRACECHAIN, CHAIN IS BUILT, and STANDARDNAMES. With the exception of CHAIN IS BUILT, these switches are discussed in Chapter III. CHAIN IS BUILT is a defined expression whose value is true if and only if the bottom of the current chain is the current presentation.

(5) The number of generators stored in the table of generators and the contents of the table.

(6) The position vectors of the relations of the presentation (if any).

(7) The contents of the syllable storage arrays, if howfar is positive. The storage arrays are printed in sections of twenty syllables each, each section beginning a new line and preceded by the syllable number of the first syllable in that line.

(8) The current chain. If the bottom of the chain is the entry ?/?-0, this indicates that the chain has been discarded, either through the use of the Trace Instruction or because an error was found during the last attempt to build a chain. The chain is initialized to "discarded" at the beginning of each program run.

8. Program Capabilities

In this section, we describe some of the more widely useful capabilities that are already built into the GROUP program. Each of these capabilities may be used by a call to a single procedure, which may farm out

part of the work to other procedures.

The procedures of this program, except for those appearing in previously described packages, are arranged alphabetically within three general sections: (1) input/output for the user's remote teletype, (2) disk save file input/output, and (3) manipulation routines. All of these procedures are declared "forward" after the package declarations. The procedures that we describe in this section all belong to section (3), the manipulation routines.

The only recursive programming in the GROUP program occurs in the input of a word. Here, recursive calls are employed to read commutators and terms.

Word Reduction: There are two canonical forms for words in the GROUP program depending on whether they are to be considered as circular words (e. g., relations are circular words; see Chapter III, General Concepts - Storage of Relations). Circular words are stored in the circularly reduced form and non-circular words are stored in the collapsed form.

The collapsed representation is obtained from a word by eliminating syllables with zero exponent and combining adjacent syllables with the same base. This representation is unique. In the GROUP program, collapsed words are formed from arbitrary representations by procedure COLLAPSE.

The circularly reduced representation is obtained by performing these contractions as though the first and last syllables of the word were adjacent. This representation is unique up to circular permutation of the syllables of the word. Circular reduction is performed in the program by procedure REDUCE and is accomplished by collapsing the word as though it were not

circular and then combining the first and last syllables if possible. If the resulting syllable has zero exponent, it is discarded and the new first and last syllables are considered. This process continues until either there is only one syllable remaining or until the first and last syllables have different bases.

Word Movement: It is often necessary to move words from one part of the syllable storage area to another part. In some of these moves, it is also desirable to change the form of the word. We describe three procedures which perform functions of this type.

To move a word without changing its form is probably the most common movement. This is accomplished by procedure COPY which simply makes another copy of the word in a position specified by the calling program section. If the areas occupied by the original word and the copy overlap, then one or both of the "copies" of the word will be destroyed (i. e., will not be the same as the original word before the copy operation). Procedure COPY does not check for overlap, but performs the copy regardless. The copy operation proceeds from the lower end of the storage arrays. Thus, if the area to be occupied by the copy lies lower in the storage arrays than the area occupied by the original word, then the copy will be correct but the original will be destroyed.

Procedure COPY also does not check that sufficient space remains in the storage arrays to contain the copy. This test must be performed by the program section ordering the copy operation. The choice not to check in COPY was made for two reasons: (1) Many sections of the GROUP program order copy operations when the nature of the operation being performed guarantees that sufficient space is available and checking is not necessary. (2) Should

insufficient space be available and checking were included in procedure COPY, then the diagnostic information obtained from an error message generated by COPY would be meager since it does not have available the identity of the program section that called it. If code were included to report the situation to the calling program section, then no savings in code would be realized. If this problem could have been easily solved, then the test would have been placed in procedure COPY in order to eliminate the large amount of code that checks for storage overflow.

To obtain the inverse of a word, procedure INVERT may be used. This procedure produces the inverse of the specified word in a position specified by the calling program section. The areas occupied by the original word and the inverse may not overlap without causing both the word and its inverse to be destroyed. As in procedure COPY , no check is performed to insure that sufficient space exists for the inverse.

To obtain the expanded form of a word (see Chapter III, General Concepts - Subword), procedure EXPAND is provided. Since the number of syllables in the expanded form of a word is not immediately available in the canonical form of a word, this procedure returns the length of the expanded word and also checks to be sure that the expanded word will fit into the storage arrays. In the case where the expanded word will not fit, an error message is generated and the operation is abandoned.

Subword Detection: Boolean procedure SUBWORD determines whether one word is a subword of another word (see Chapter III, General Concepts for a definition). More exactly, given two words, called the S-word

and the C-word, the procedure determines whether there is a copy of the S-word that is contained as a subword in the C-word. The outcome of the test is returned through the name of the procedure and, if the S-word is a subword, the location of the first syllable of the contained copy is returned through an argument in the parameter list. In addition to the S- and C-words, and the variable to hold the beginning position of the contained copy, the parameter list also contains a Boolean argument specifying whether the C-word is a circular word and a pointer to the beginning of "free" space. The determination of whether the S-word is a subword of the C-word is divided into two cases, which are discussed below.

A procedure used in both cases is procedure SCANS which determines whether a given generator appears in a word. SCANS is a Boolean procedure which returns through its name whether the generator appears and if so, returns through a parameter the location of the first such occurrence. This procedure is used in many places other than in procedure SUBWORD.

Case I: The S-word has only one (reduced) syllable.

In this case SCANS is used to determine whether the base of the S-word appears in the C-word. If not, then false is returned as the value of SUBWORD. Otherwise, each such occurrence of the base is tested until one is found with exponent of the same sign as that of the S-word and not smaller in magnitude. If no such occurrence exists, then false is returned as the value of SUBWORD. Otherwise, SUBWORD is true and the location of the first such occurrence is returned through the parameter list.

Case II: The S-word has more than one (reduced) syllable. First, SCANS is used to determine whether the base of the first syllable of the S-word is contained in the C-word. If not, then SUBWORD is false. If so, then the words are tested against each other, syllable by syllable, for agreement. Each syllable of the S-word must match exactly the corresponding syllable of the C-word, except that the first and last syllables of the S-word must have the same base as the corresponding C-syllables and the exponents must have the property that the corresponding exponents must have the same sign and the S-exponent must not be larger in magnitude than the C-exponent. If any of these conditions fails to be satisfied, then the search is continued through the portion of the C-word following the occurrence of the first base. If, during the checking, the end of the C-word is encountered before exhausting the S-word and the C-word is circular, then a flag is set marking the condition and the initial syllables of the C-word are used for checking against the S-word; otherwise, containment cannot occur and SUBWORD is set false. If it is found that the S-word is a subword of the C-word so that the contained copy is in two pieces (e.g., a b c d in c d x y a b), then the C-word is circle-shifted so that the contained copy is in one piece (e.g., a b c d x y in the example above). It should be noted that this procedure requires the location of free space only in order to perform the circle-shifting. If either the S-word consists of a single syllable or the C-word is not circular, then the parameter giving the beginning of "free" space may be replaced by a dummy.

Replacement: Replacement in the GROUP program consists of finding all copies of a given word (the S-word) in another given word (the C-word) and replacing these copies with a third word (the R-word). Two Boolean procedures perform this type of replacement. These are REPLACER and SUBSTITUTE. REPLACER is the basic routine for performing replacements and is called by SUBSTITUTE and other routines. In addition to the three words, these procedures also require a Boolean parameter specifying whether the C-word is circular and a pointer to the beginning of free space (this pointer may be changed by the action of either of these procedures -- see Storage Allocation). Since REPLACER is the basic routine, we will discuss its operation first. WARNING: Either procedure can lock into an endless loop if the S-word is a subword of the R-word. The GROUP program as released has been designed so that no such event will occur.

REPLACER first uses procedure SUBWORD to determine whether the S-word is a subword of the C-word. If not, then REPLACER is set to false and control is returned to the calling section. If the S-word is a subword, then two cases are considered depending on the length of the S-word: (1) If the S-word consists of a single syllable, then the R-word is inserted into the C-word immediately before the occurrence of the contained copy of the S-word and the exponent of the contained copy is reduced by the exponent of the S-word. If the S-word is contained more than once in a single occurrence (e.g., a^2 is contained "twice" in $u\ v\ w\ a^5\ x\ y\ z$), then only a single replacement is made (e.g., the above example would yield $u\ v\ w\ r\ a^3\ x\ y\ z$ where r is the R-word). This choice is made because experience has shown that

making all the possible replacements (giving, for the above, $\underline{u}\,\underline{v}\,\underline{w}\,\underline{r}^2\,\underline{a}\,\underline{x}\,\underline{y}\,\underline{z}$) could lead to significantly more time being required for other operations because possible cancellations would not occur. (2) If the S-word consists of more than one syllable, then the C-word is broken to discard the middle syllable of the contained copy and inserting the R-word in their place. The exponents of the first and last syllables of the contained copy are reduced by the exponents of the first and last syllables of the S-word. In each of these cases, after the actual replacement is performed, the new C-word is collapsed (circularly reduced if the C-word is circular) and the operation is begun again using the new C-word.

Procedure SUBSTITUTE causes each occurrence of the S-word in the C-word to be replaced by the R-word and each occurrence of the inverse of the S-word to be replaced by the inverse of the R-word. This procedure makes copies of the inverse of the S- and R- words and uses procedure REPLACER to perform the actual replacements.

9. Preserve and Release Instructions

The Preserve Instruction was implemented by noting (1) that the procedure (SOLVABLE) that determines whether a generator is solvable in a word first uses procedure SCANS to determine whether the generator appears in the word in any form, and (2) that the argument to SOLVABLE in the Simplification Step giving the generator is the form of the generator that appears in the table of generators. Thus a generator can be made so that SCANS will not detect its presence in any word through a statement of the form

GENERATOR [i] := - ABS(GENERATOR [I]).

This also accounts for the use of the absolute value function ABS in most other references to the generator table.

10. Program Listing and Deck

Since the source deck of the GROUP program is quite long (over 5500 cards at the last resequencing), the listing of the program is not included in this report. Program listings and decks are available upon request from the Mathematics Research Center.

References

1. Crowell, R., and R. Fox, Introduction to Knot Theory , Ginn and Company, Boston, 1963.

2. Hassitt, A., Design and Implementation of a General-Purpose Input Routine, Comm. ACM, Vol. 7, 6(June, 1964), 350-355.

3. Magnus, W., A. Karrass and D. Solitar, Combinatorial Group Theory, Interscience Publishers, New York, 1966.

4. Rabin, M. A., "Recursive Unsolvability of Group Theoretic Problems", Annals of Mathematics, Vol. 67 (1958), 127-194.

Appendix A

Formal Input Syntax

It should be remembered that input messages to the GROUP program are broken down into their component parts by RDNUM. The divisions indicated by the syntax refer to component strings as determined by RDNUM. A blank will always serve as a separator.

Primitives:

<empty>

<space>

(The following primitives are defined by RDNUM)

<integer>

<real number>

<special symbol>

<alphabetic symbol>

<illegal symbol>

<no input>

General:

<comment> ::= <empty> | {any text, subject to restrictions}†

<continuation> ::= : <comment> ←

†See the note on Comment in Chapter II, Section 3.

Generators:

<define generator list> ::= <empty> | <define generator list>
<acceptable generator> | <define generator list> , |
<define generator list> <continuation>

<acceptable generator> ::= <alphabetic symbol>

<generator> ::= {<acceptable generator> which appears in the array of generators}

<generator list> ::= <empty> | <generator list> <generator>|
<generator list> {<comment> not containing a <generator>}

Words:

<restricted word> ::= {<word> in which each <acceptable generator> is a <generator>}

<word> ::= <identity> | <non-identity>

<identity> ::= 1

<non-identity> ::= <factor> | <continuation> | <non-identity> <factor> | <non-identity> < continuation>

<factor> ::= <simple factor> <exponent>

<simple factor> ::= <acceptable generator> | <term> | <commutator>

<term> ::= (<word>)

<commutator> ::= [<word> , <word>]

<exponent> ::= <empty> | <integer>

Relations:

<relation number> ::= {<integer> which is ≥ 1 and $\leq$ the number of relations in the current presentation}

<relation list> ::= <empty> | <relation list> <relation number> | <relation list> {<comment> not containing a <relation number>}

<relation designator> ::= RELATION <relation number> | <relation number>

Presentations:

<presentation part> ::= <disk file prefix> <separator> <disk file suffix> - <presentation identifier> | <disk file suffix> - <presentation identifier> | <presentation identifier>

<separator> ::= <space> | /

<presentation identifier> ::= <presentation number> | <last presentation designator>

<presentation number> ::= {positive <integer>}

<last presentation designator> ::= {non-positive <integer>} | <real number> | <special symbol> | <alphabetic symbol> | <illegal symbol>

<disk file prefix> ::= <disk file identifier>

<disk file suffix> ::= <disk file identifier>

<disk file identifier> ::= {up to 6 legal non-blank characters except "/" or "-"}

Instruction Syntax

Input/Output Instructions:

< enter presentation instruction> ::= PRESENTATION < comment>

< read presentation instruction> ::= < disk read verb>
< presentation part>

< disk read verb> ::= READ | RECALL

< print instruction> ::= PRINT <comment>

< type instruction > ::= TYPE | TYPE < output designator> < comment>

< output designator> ::= PRESENTATION | < relation designator> |
GENERATORS | < size designator>

< size designator> ::= LENGTH | SIZE

< comment instruction> ::= * < comment>

< print width instruction> ::= USE < width> < comment>

< width> ::= {< integer> which is $\geq$ 50 and $\leq$ PRINTBUFFERSIZE† }

Manipulation Instructions:

< solve instruction> ::= SOLVE < solve detail>

< solve detail> ::= < empty> | < solve detail> < relation number > |
< solve detail> < solve generator part> | < solve detail>
{< comment> not containing a < relation number> or < solve
generator part>}

< solve generator part> ::= FOR < generator>

---

†see Chapter IV, Free Format Output System

Manipulation Instructions (continued):

<simplify instruction> ::= SIMPLIFY < simplification steps>

<simplification steps> ::= <indefinite simplification steps> |
<indefinite simplification steps> <simplification times>
<comment>

<indefinite simplification steps> ::= {<comment> not containing
<simplification times>}

<simplification times> ::= {positive <integer>}

<check instruction> ::= CHECK <generator list>

<preserve instruction> ::= PRESERVE <generator list>

<release instruction> ::= RELEASE | RELEASE <generator list>

<define instruction> ::= DEFINE <new generator> = <restricted word>

<new generator> ::= {< acceptable generator > which is not a
<generator>}

<contain switch instruction> ::= <contain switch verb> <comment>

<contain switch verb> ::= CONTAIN | *CONTAIN | NOCONTAIN

Editing Instructions:

<remove instruction> ::= <remove trivial relations instruction> |
<remove relations instruction> | < remove generators
instruction>

<remove trivial relations instruction> ::= REMOVE | REMOVE
TRIVIAL <comment>

Editing Instructions (continued):

<remove relations instruction> ::= <remove verb> RELATIONS
<relation list>

<remove generators instruction> ::= <remove verb> GENERATORS
<generator list>

<remove verb> ::= REMOVE | DELETE

<alter instruction> ::= ALTER <alter information> <relation key>
<alter information> TO <word>

<alter information> ::= <empty> | <alter information> <key> |
<alter information> {<comment> not containing a <key>
or "TO"}

<key> ::= <relation key> | <size key>

<relation key> ::= RELATION <relation number>

<size key> ::= FIRST <integer>

<replace instruction> ::= REPLACE <non-trivial word> <with
delimiter> <restricted word> <in delimiter> <relation
identifier>

<non-trivial word> ::= {<restricted word> which is not
equivalent to _1_}

<with delimiter>† ::= , | ← | WITH

<in delimiter>‡ ::= , | ← | IN

---

†"WITH" may be used as a <with delimiter> only if it is not a <generator>

‡"IN" may be used as a <in delimiter> only if it is not a <generator>.

Editing Instructions (continued):

<relation identifier> ::= {<comment > not containing "ALL" , "RELATION" or <relation number>} <replace in part>

<replace in part> ::= <all relations part> | <specified relations part>

<all relations part> ::= ALL <comment> | <no input>

<specified relations part> ::= <relation list> | RELATIONS <relation list>

<add instruction> ::= ADD GENERATORS <define generator list> | ADD RELATION <word>

<shift instruction> ::= SHIFT <relation designator> <shift direction> <shift size> <comment>

<shift direction> ::= RIGHT | LEFT

<shift size> ::= <integer>

<invert instruction> ::= INVERT <relation designator> <comment>

Chain Instructions:

<follow instruction> ::= FOLLOW {<restricted word> in <generator>s of the top of the chain}†

<build chain instruction> ::= BUILD {<comment> not containing "TO"} <top of chain part>

<top of chain part> ::= <empty> | TO <presentation part>

---

†See Follow Instruction, Chapter III

Chain Instructions (continued):

<trace instruction> ::= <trace verb> <comment>

<trace verb> ::= TRACE | NOTRACE

Switch Control Instructions:

<detail switch instruction> ::= <detail switch verb> <comment>

<detail switch verb> ::= DETAIL | NODETAIL

<type switch instruction> ::= <type switch verb> <comment>

<type switch verb> ::= SUPPRESS | RESUME

Disk Save File Control Instructions:

<disk file name instruction> ::= <disk name verb> <comment>

<disk name verb> ::= STANDARD | NONSTANDARD

<new disk file instruction> ::= FINDFILE <comment>

<disk summary instruction> ::= DISKSUMMARY <comment>

Debugging Instructions:

<cleanup instruction> ::= CLEANUP <comment>

<dump instruction> ::= DUMP <comment>

Miscellaneous Instructions:

<abelianize instruction> ::= ABELIANIZE <relation list>

<time instruction> ::= TIME <comment>

Miscellaneous Instructions (continued):

<suspend instruction> ::= SUSPEND <suspension time> <comment>

<suspension time> ::= <positive integer>

< quit instruction> ::= QUIT <comment>

Appendix B

Sample Program Run - Teletype Output

```
??EXECUTE GROUP←
 7:GROUP/GROUP=03 BOJ 1028 FROM 01/04
GROUP PRESENTATION PROGRAM - VERSION 9.3
:  * SAMPLE RUN FOR TECHNICAL SUMMARY REPORT←
:  DUMP (TO ILLUSTRATE INITIAL STATE OF PROGRAM)←
:  USE 65 PRINTER COLUMNS←
:  NONSTANDARD DISK SAVE FILE NAMES←
:  PRESENTATION -- GROUP OF A SIMPLE LINK←
PLEASE ENTER A DISK SAVE FILE NAME  SAMPLE/RUN←
DISK SAVE FILE IS SAMPLE/RUN
ENTER GENERATORS:  A1 A2 A3 A4 B1 B2←
RELATION 1.  A2 A3 A1-1 A3-1←
RELATION 2.  A3 A2 A4-1 A2-1←
RELATION 3.  B1 A4 B1-1 A2-1←
RELATION 4.  A1 B1 A3-1 B1-1←
RELATION 5.  B2 A1 B1-1 A1-1←
RELATION 6.  A2 B1 A2-1 A2-1←
RELATION 7.  ←
```

```
PRESENTATION SAMPLE/RUN-1
GENERATORS  A1, A2, A3, A4, B1, B2
RELATION
     1.  A2 A3 A1 -1 A3 -1
     2.  A3 A2 A4 -1 A2 -1
     3.  B1 A4 B1 -1 A2 -1
     4.  A1 B1 A3 -1 B1 -1
     5.  B2 A1 B1 -1 A1 -1
     6.  A2 B1 A2 -1 A2 -1

DO YOU WANT TO MAKE CORRECTIONS?  YES←
TYPE RELATION NUMBER, "=", AND CORRECT RELATION

RELATION 6 = A2 B1 A2-1 B2-1←
RELATION 6.  A2 B1 A2 -1 B2 -1

RELATION ←
```

```
: * NOW ABELIANIZE AND CHECK HOMOLOGY←
: ABELIANIZE←

PRESENTATION SAMPLE/RUN-3
GENERATORS A1, A2, A3, A4, B1, B2
RELATION
    1. A1 -1 A2
    2. A3 A4 -1
    3. A2 -1 A4
    4. A1 A3 -1
    5. B1 -1 B2
    6. B1 B2 -1

: * NOTE THAT A1 = A2 = A3 = A4 AND B1 = B2←
: * HENCE HOMOLOGY IS FREE ABELIAN ON TWO←
: * GENERATORS, WHICH IS CORRECT←
: RECALL 1←
REMINDER: NO GENERATORS ARE PRESERVED
PRESENTATION SAMPLE/RUN-1
: PRESERVE A1 B1←
PRESERVED A1 B1
```

```
: SIMPLIFY←

   SOLVE RELATION 1 FOR A2
   A2 = A3 A1 A3 -1
   YIELDING PRESENTATION SAMPLE/RUN-5

   SOLVE RELATION 4 FOR A3
   A3 = B1 -1 A1 B1
   YIELDING PRESENTATION SAMPLE/RUN-7

   SOLVE RELATION 2 FOR A4
   A4 = B1 -1 A1 B1 A1 -1 B1 -1 A1 B1 A1 B1 -1 A1 -1 B1
   YIELDING PRESENTATION SAMPLE/RUN-9

   SOLVE RELATION 5 FOR B2
   B2 = A1 B1 A1 -1
   YIELDING PRESENTATION SAMPLE/RUN-11

PRESENTATION SAMPLE/RUN-11
GENERATORS  A1, B1
RELATION
    1.  1
    2.  1
    3.  B1 A1 B1 A1 -1 B1 -1 A1 B1 A1 B1 -1 A1 -1 B1 -1 A1 B1 A1 -1
          B1 -1 A1 -1
    4.  1
    5.  1
    6.  B1 -1 A1 B1 A1 B1 -1 A1 -1 B1 A1 B1 A1 -1 B1 -1 A1 -1 B1 A1
          B1 -1 A1 -1
```

```
:  *CONTAIN←
   THESE RELATIONS WERE FOUND REDUNDANT:  6

PRESENTATION SAMPLE/RUN-12
GENERATORS  A1, B1
RELATION
     1.  1
     2.  1
     3.  B1 A1 B1 A1 -1 B1 -1 A1 B1 A1 B1 -1 A1 -1 B1 -1 A1 B1 A1 -1
            B1 -1 A1 -1
     4.  1
     5.  1
     6.  1


:  BUILD CHAIN←
TOP OF CHAIN IS PRESENTATION SAMPLE/RUN-1
:  FOLLOW B1 A2-1 B1-1 A3-1←
IS THE WORD CIRCULAR?  NO←
EQUIVALENT WORD IS  A1 B1 A1 -1 B1 -1 A1 -1 B1 -1 A1 -1 B1
DO YOU WISH CONTAINMENT OR ABELIANIZATION?  NO←
:  DUMP (A MORE GENERAL EXAMPLE THAN THE INITIAL STATE)←
.  QUIT←
CPU TIME IS    1 MIN 31 SEC
I/O TIME IS    0 MIN 46 SEC
 GROUP/GROUP: 3 EOJ 1039
```

Appendix C

Sample Program Run - Printer Output

```
                    * SAMPLE RUN FOR TECHNICAL SUMMARY REPORT
! DUMP (TO ILLUSTRATE INITIAL STATE OF PROGRAM)

                        DUMP TAKEN AT USER REQUEST

                        CURRENT DISK SAVE FILE IS ???/???
                        CURRENT PRESENTATION NAME IS ???/???-0

                        SWITCH SETTINGS
                             CONTAINSWITCH      FALSE
                             DETAIL             TRUE
                             TYPEENABLE         TRUE
                             TRACECHAIN         FALSE
                             CHAIN IS BUILT     FALSE
                             STANDARDNAMES      TRUE

                        0 GENERATORS

                        NO RELATIONS

                        FIRST 10 OF 3000 SYLLABLES FOLLOW

                    0.  ! ! ! ! ! ! ! ! ! !
```

NOT REPRODUCIBLE

```
                TOP OF CHAIN
                    ENTRY ???=0 - PRESENTATION STORAGE
                END OF CHAIN

* PRESENTATION -- GROUP OF A SIMPLE LINK

    PRESENTATION SAMPLE/RUN=1
    GENERATORS  A1, A2, A3, A4, B1, B2
    RELATION

        1.  A2 A3 A1^-1 A3^-1

        2.  A3 A2 A4^-1 A2^-1

        3.  B1 A4 B1^-1 A2^-1

        4.  A1 B1 A3^-1 A1^-1

        5.  B2 A1 B1^-1 A1^-1

        6.  A2 B1 A2^-1 B2^-1


                    * RUN ABELIANIZE AND CHECK HOMOLOGY
* ABELIANIZE

    PRESENTATION SAMPLE/RUN=3
    GENERATORS  A1, A2, A3, A4, B1, B2
    RELATION

        1.  A1^-1 A2

        2.  A3 A4^-1
```

3. $A2^{-1}\ A4$

4. $A1\ A3^{-1}$

5. $B1^{-1}\ B2$

6. $B1\ B2^{-1}$

* NOTE THAT A1 = A2 = A3 = A4 AND B1 = B2
* HENCE HOMOLOGY IS FREE ABELIAN ON TWO
* GENERATORS, WHICH IS CORRECT

: RECALL 1

PRESENTATION SAMPLE/RUN=1

: PRESERVE A1 B1

: SIMPLIFY

SOLVE RELATION 1 FOR A2

$A2 = A3\ A1\ A3^{-1}$

PRESENTATION SAMPLE/RUN=5

GENERATORS A1, A3, A4, B1, B2

RELATION

1. 1

2. $A3\ A1\ A3^{-1}\ A4^{-1}\ A3\ A1^{-1}$

3. $B1\ A4\ B1^{-1}\ A3\ A1^{-1}\ A3^{-1}$

4. $A1\ B1\ A3^{-1}\ B1^{-1}$

5. $B2\ A1\ B1^{-1}\ A1^{-1}$

6. $A3\ A1\ A3^{-1}\ B1\ A3\ A1^{-1}\ A3^{-1}\ B2^{-1}$

SOLVE RELATION 4 FOR A3

$$A3 = B1^{-1}\ A1\ B1$$

PRESENTATION SAMPLE/RUN=7
GENERATORS A1, A4, B1, B2
RELATION

1. 1

2. $B1^{-1}\ A1\ B1\ A1\ B1^{-1}\ A1^{-1}\ B1\ A4^{-1}\ B1^{-1}\ A1\ B1\ A1^{-1}$

3. $B1^{2}\ A4\ B1^{-2}\ A1\ B1\ A1^{-1}\ B1^{-1}\ A1^{-1}$

4. 1

5. $B2\ A1\ B1^{-1}\ A1^{-1}$

6. $B1^{-1}\ A1\ B1\ A1\ B1^{-1}\ A1^{-1}\ B1\ A1\ B1\ A1^{-1}\ B1^{-1}\ A1^{-1}\ B1\ B2^{-1}$

SOLVE RELATION 2 FOR A4

$$A4 = B1^{-1}\ A1\ B1\ A1^{-1}\ B1^{-1}\ A1\ B1\ A1\ B1^{-1}\ A1^{-1}\ B1$$

PRESENTATION SAMPLE/RUN=9
GENERATORS A1, B1, B2
RELATION

1. 1
2. 1

3. $B1\ A1\ B1\ A1^{-1}\ B1^{-1}\ A1\ B1\ A1\ B1^{-1}\ A1^{-1}\ B1^{-1}\ A1\ B1\ A1^{-1}\ B1^{-1}\ A1^{-1}$

4. 1

5. $B2\ A1\ B1^{-1}\ A1^{-1}$

6. $B1^{-1}\ A1\ B1\ A1\ B1^{-1}\ A1^{-1}\ B1\ A1\ B1\ A1^{-1}\ B1^{-1}\ A1^{-1}\ B1\ B2^{-1}$

SOLVE RELATION 5 FOR B2

$B2 = A1\ B1\ A1^{-1}$

PRESENTATION SAMPLE/RUN=11

GENERATORS A1, B1

RELATION

1. 1
2. 1

3. $B1\ A1\ B1\ A1^{-1}\ B1^{-1}\ A1\ B1\ A1\ B1^{-1}\ A1^{-1}\ B1^{-1}\ A1\ B1\ A1^{-1}\ B1^{-1}\ A1^{-1}$

4. 1
5. 1

6. $B1^{-1}\ A1\ B1\ A1\ B1^{-1}\ A1^{-1}\ B1\ A1\ B1\ A1^{-1}\ B1^{-1}\ A1^{-1}\ B1\ A1\ B1^{-1}\ A1^{-1}$

```
: *CONTAIN
  RELATION 3 IS CONTAINED IN RELATION 6

                    A1 B1 A1 B1^-1 A1^-1 B1 A1 B1 A1^-1

          REPLACED BY  B1 A1 B1 A1^-1 B1^-1 A1 B1
  THESE RELATIONS WERE FOUND REDUNDANT:  6

  PRESENTATION SAMPLE/RUN=12
  GENERATORS  A1, B1
  RELATION
      1.  1
      2.  1

      3.  B1 A1 B1 A1^-1 B1^-1 A1 B1 A1 B1^-1 A1^-1 B1^-1 A1 B1

             A1^-1 B1^-1 A1^-1
      4.  1
      5.  1
      6.  1


: BUILD CHAIN

: FOLLOW  B1 A2^-1 B1^-1 A3^-1
        NON-CIRCULAR WORD


  EQUIVALENT WORD IS  A1 B1 A1^-1 B1^-1 A1^-1 B1^-1 A1^-1 B1
```

: DUMP (A MORE GENERAL EXAMPLE THAN THE INITIAL STATE)

```
DUMP TAKEN AT USER REQUEST

CURRENT DISK SAVE FILE IS SAMPLE/RUN
CURRENT PRESENTATION NAME IS SAMPLE/RUN=12

SWITCH SETTINGS
     CONTAINSWITCH      FALSE
     DETAIL             TRUE
     TYPEENABLE         TRUE
     TRACECHAIN         FALSE
     CHAIN IS BUILT     TRUE
     STANDARDNAMES      FALSE

2 GENERATORS   A1, B1
RELATION     BEGIN     LENGTH
    1          -1         -1
    2          -1         -1
    3           0         16
    4          -1         -1
    5          -1         -1
    6          -1         -1
```

FIRST 96 OF 3000 SYLLABLES FOLLOW

0. $B1\ A1\ B1\ A1^{-1}\ B1^{-1}\ A1\ B1\ A1\ B1^{-1}\ A1^{-1}\ B1^{-1}\ A1\ B1\ A1^{-1}\ B1^{-1}\ A1^{-1}\ B1\ ::\ B1^{-1}\ A3^{-1}$

20. $A1\ B1\ A1^{-1}\ B1^{-1}\ A1^{-1}\ B1^{-1}\ A1^{-1}\ B1\ ::\ A4\ B1^{-1}\ A1\ B1\ A1^{-1}\ B1^{-1}\ A1\ B1\ A1\ B1^{-1}\ A1^{-1}$

40. $B1\ ::\ B2^{-1}\ A1\ B1^{-1}\ A1^{-1}\ ::^{-1}\ B2\ B1\ ::^{0}\ A1^{-1}\ B1^{-1}\ A2\ B1\ A1\ B1^{-1}\ A1\ B1\ A1\ B1^{-1}$

60. $A1^{-1}\ B1\ A1\ B1\ A1^{-1}\ B1\ A1^{-1}\ B1^{-1}\ A1\ B1\ A1\ B1^{-1}\ A1\ B1\ A1\ B1^{-1}\ A1^{-1}\ B1\ A1\ B1$

80. $A1^{-1}\ B1^{0}\ A1^{0}\ B1^{0}\ A1^{0}\ B1^{0}\ B1^{-1}\ A1\ B1\ A2^{0}\ B1^{-1}\ A1^{-1}\ B1\ A1\ B1^{-1}\ A1^{-1}$

TOP OF CHAIN

ENTRY SAMPLE/RUN=1 - PRESENTATION STORAGE

```
        ENTRY SAMPLE/RUN-4 - REPLACEMENT ENTRY
        ENTRY SAMPLE/RUN-5 - PRESENTATION STORAGE
        ENTRY SAMPLE/RUN-6 - REPLACEMENT ENTRY
        ENTRY SAMPLE/RUN-7 - PRESENTATION STORAGE
        ENTRY SAMPLE/RUN-8 - REPLACEMENT ENTRY
        ENTRY SAMPLE/RUN-9 - PRESENTATION STORAGE
        ENTRY SAMPLE/RUN-10 - REPLACEMENT ENTRY
        ENTRY SAMPLE/RUN-11 - PRESENTATION STORAGE
        ENTRY SAMPLE/RUN-12 - PRESENTATION STORAGE
END OF CHAIN
```

INDEX

< > indicates the location of the formal syntax of this term.

AR 70-31 Unclassified

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) Mathematics Research Center, University of Wisconsin, Madison, Wis. 53706	2a. REPORT SECURITY CLASSIFICATION Unclassified 2b. GROUP None

3. REPORT TITLE

A COMPUTER PROGRAM FOR INTERACTIVE MANIPULATION OF FINITE GROUP PRESENTATIONS

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

Summary Report: no specific reporting period.

5. AUTHOR(S) (First name, middle initial, last name)

F. D. Crary

6. REPORT DATE February 1970	7a. TOTAL NO. OF PAGES 124	7b. NO. OF REFS 4
8a. CONTRACT OR GRANT NO. Contract No. DA-31-124-ARO-D-462 b. PROJECT NO. None c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) 1045 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report) None	

10. DISTRIBUTION STATEMENT

Distribution of this document is unlimited.

11. SUPPLEMENTARY NOTES None	12. SPONSORING MILITARY ACTIVITY Army Research Office-Durham, N.C.

13. ABSTRACT

An interactive computer program for manipulating finite group presentations is described. A brief mathematical background is provided along with operating instructions and notes on the programming. An illustrative example is included

DD FORM 1 NOV 1473

Unclassified

Security Classification

AGO 5696A

Mathematics Research Center

A COMPUTER PROGRAM FOR INTERACTIVE MANIPULATION OF FINITE GROUP PRESENTATIONS

F. D. Crary

MRC Report No. 1045 AD_____

Contract No. : DA-31-124-ARO-D-462

February 1970

124 pp.

UNCLASSIFIED

Group Theory and Generalizations
Computing machines

An interactive computer program for manipulating finite group presentations is described. A brief mathematical background is provided along with operating instructions and notes on the programming. An illustrative example is included.

Mathematics Research Center

A COMPUTER PROGRAM FOR INTERACTIVE MANIPULATION OF FINITE GROUP PRESENTATIONS

F. D. Crary

MRC Report No. 1045 AD_____

Contract No.: DA-31-124-ARO-D-462

February 1970

124 pp.

UNCLASSIFIED

Group Theory and Generalizations
Computing machines

An interactive computer program for manipulating finite group presentations is described. A brief mathematical background is provided along with operating instructions and notes on the programming. An illustrative example is included.

Mathematics Research Center

A COMPUTER PROGRAM FOR INTER - ACTIVE MANIPULATION OF FINITE GROUP PRESENTATIONS

F. D. Crary

MRC Report No. 1045 AD_____

Contract No. : DA-31-124-ARO-D-462

February 1970

124 pp.

UNCLASSIFIED

Group Theory and Generalizations
Computing machines

An interactive computer program for manipulating finite group presentations is described. A brief mathematical background is provided along with operating instructions and notes on the programming. An illustrative example is included.

Mathematics Research Center

A COMPUTER PROGRAM FOR INTERACTIVE MANIPULATION OF FINITE GROUP PRESENTATIONS

F. D. Crary

MRC Report No. 1045 AD_____

Contract No.: DA-31-124-ARO-D-462

February 1970

124 pp.

UNCLASSIFIED

Group Theory and Generalizations
Computing machines

An interactive computer program for manipulating finite group presentations is described. A brief mathematical background is provided along with operating instructions and notes on the programming. An illustrative example is included.

Mathematics Research Center

A COMPUTER PROGRAM FOR INTERACTIVE MANIPULATION OF FINITE GROUP PRESENTATIONS

F. D. Crary

MRC Report No. 1045 AD_____

Contract No. : DA-31-124-ARO-D-462

February 1970

124 pp.

UNCLASSIFIED

Group Theory and Generalizations
Computing machines

An interactive computer program for manipulating finite group presentations is described. A brief mathematical background is provided along with operating instructions and notes on the programming. An illustrative example is included.

Mathematics Research Center

A COMPUTER PROGRAM FOR INTERACTIVE MANIPULATION OF FINITE GROUP PRESENTATIONS

F. D. Crary

MRC Report No. 1045 AD_____

Contract No.: DA-31-124-ARO-D-462

February 1970

124 pp.

UNCLASSIFIED

Group Theory and Generalizations
Computing machines

An interactive computer program for manipulating finite group presentations is described. A brief mathematical background is provided along with operating instructions and notes on the programming. An illustrative example is included.

Mathematics Research Center

A COMPUTER PROGRAM FOR INTER - ACTIVE MANIPULATION OF FINITE GROUP PRESENTATIONS

F. D. Crary

MRC Report No. 1045 AD_____

Contract No. : DA-31-124-ARO-D-462

February 1970

124 pp.

UNCLASSIFIED

Group Theory and Generalizations
Computing machines

An interactive computer program for manipulating finite group presentations is described. A brief mathematical background is provided along with operating instructions and notes on the programming. An illustrative example is included.

Mathematics Research Center

A COMPUTER PROGRAM FOR INTERACTIVE MANIPULATION OF FINITE GROUP PRESENTATIONS

F. D. Crary

MRC Report No. 1045 AD_____

Contract No.: DA-31-124-ARO-D-462

February 1970

124 pp.

UNCLASSIFIED

Group Theory and Generalizations
Computing machines

An interactive computer program for manipulating finite group presentations is described. A brief mathematical background is provided along with operating instructions and notes on the programming. An illustrative example is included.