

COMPUTATIONAL TECHNIQUES APPLIED TO GROUP PRESENTATIONS

Kevin Rutherford

A Thesis Submitted for the Degree of PhD
at the
University of St Andrews



1989

Full metadata for this item is available in
St Andrews Research Repository
at:

<http://research-repository.st-andrews.ac.uk/>

Please use this identifier to cite or link to this item:

<http://hdl.handle.net/10023/13432>

This item is protected by original copyright

Thesis
submitted for the degree of
Doctor of Philosophy

Computational Techniques
applied to
Group Presentations

by

Kevin Rutherford



ProQuest Number: 10167175

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



ProQuest 10167175

Published by ProQuest LLC (2017). Copyright of the Dissertation is held by the Author.

All rights reserved.

This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Thesis
submitted for the degree of
Doctor of Philosophy

Computational Techniques
applied to
Group Presentations

by

Kevin Rutherford



Tu
A936

DECLARATIONS

I, Kevin Rutherford, hereby certify that this thesis has been composed by myself, that it is a record of my own work, and that it has not been accepted in partial or complete fulfilment of any other degree or professional qualification.

Signed:

Date: 31.1.89

I was admitted to the Faculty of Science of the University of St. Andrews under Ordinance General no. 12 in April 1982, and as a candidate for the degree of Ph.D. in February 1983.

Signed:

Date: 31.1.89

In submitting this thesis to the University of St. Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and abstract will be published, and that a copy of the work may be made and supplied to any *bona fide* library or research worker.

CERTIFICATION

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate to the Degree of Ph.D.

Signed:

Date: 2/2/89

Title: Computational Techniques applied to Group Presentations
Author: K.Rutherford
Faculty: Department of Mathematical Sciences
Supervisor: Dr E.F.Robertson
Date: 31-1-89
Pages: 300 (approx.) (53,000 words)

Abstract:

Designs for a collection of re-usable software modules are developed. The modules are implemented in C and expressed in a tool-kit for the Unix operating system. Each tool is an expert in some aspect of the manipulation by computer of group presentations. The granularity of the tool-kit has been chosen so that common usages of the Todd-Coxeter and Reidemeister-Schreier methods can be expressed in various ways using any tool composition language (eg. shell scripts), and running as a collection of co-operating processes. Data file formats for the interchange of group-theoretic information between processes are described. The tools are tested on well-known examples, and are used to prove a long-standing conjecture. Use of the tools as the basis for a rule-based "expert system" is discussed.

TABLE OF CONTENTS

0. INTRODUCTION

0-1. Rationale

0-2. Summary

1. TOOL-SET DESIGN

1-0. Introduction

1-0.1. - Primitive Tool Architecture

1-1. Orderings and Canonical Forms

1-1.1. - Motivations

1-1.2. - Word Length

1-1.2.1. - Rationale

1-1.2.2. - Design

1-1.3. - The canon() Function

1-1.3.1. - Generator Processing

1-1.3.2. - Relator Pre-Processing

1-1.3.3. - Relator Processing

1-1.3.4. - Relator Post-Processing

1-1.3.5. - Remarks

1-2. Tietze Transformations

- 1-2.1. - Motivations and Requirements
 - 1-2.1.1. - Motivations
 - 1-2.1.2. - Requirements
- 1-2.2. - High-Level Design
 - 1-2.2.1. - General Architecture
 - 1-2.2.2. - Length Functions
 - 1-2.2.3. - Tietze Transformations
- 1-2.3. - Detailed Design
 - 1-2.3.1. - User Interface
 - 1-2.3.1.1. - Language Structure
 - 1-2.3.1.2. - Simple Commands
 - 1-2.3.1.3. - Composite Commands
 - 1-2.3.2. - Run-Time Variables
 - 1-2.3.2.1. - Read-Only Variables
 - 1-2.3.2.2. - Read-Write Variables
 - 1-2.3.3. - Atomic Commands
- 1-2.4. - Examples

1-3. Cosets and Enumeration

- 1-3.1. - Requirements and Rationale
 - 1-3.1.1. - Coset-Id Equivalence
 - 1-3.1.2. - Coset-Id Sufficiency
 - 1-3.1.3. - Coset-Id Definition
 - 1-3.1.4. - Subgroup Definition
 - 1-3.1.5. - Coset Enumeration
- 1-3.2. - High-Level Design
 - 1-3.2.1. - General Architecture
 - 1-3.2.1.1. - Inputs
 - 1-3.2.1.2. - Outputs
 - 1-3.2.1.3. - Modules
 - 1-3.2.2. - Data Structures
 - 1-3.2.3. - Coset-id Definition
 - 1-3.2.4. - Coset-id Sufficiency
 - 1-3.2.5. - Coincidence Processing
- 1-3.3. - Low-Level Design
 - 1-3.3.1. - Data Structures
 - 1-3.3.2. - Coset-id Definition
 - 1-3.3.3. - Coset-id Sufficiency
 - 1-3.3.4. - Coincidence Processing

1-4. Subgroups of Small Index

1-5. The Reidemeister-Schreier Process

- 1-5.1. - Rationale
- 1-5.2. - Design
 - 1-5.2.1. - Architecture
 - 1-5.2.2. - Constructing a Spanning Tree
 - 1-5.2.3. - An Example

1-6. Other Primitive Tools

2. *USING_THE_TOOLS*

2-0. Introduction

2-1. Tests of Individual Tools

- 2-1.1. Introduction
- 2-1.2. Definition Strategies in Coset Enumeration
 - 2-1.2.1. - Introduction
 - 2-1.2.2. - Definitions Restriction
 - 2-1.2.3. - Memory Restriction
 - 2-1.2.4. - Pass Components
 - 2-1.2.5. - Equivalence Restriction
 - 2-1.2.6. - Conclusions
- 2-1.3. General Methods in 'tt'
 - 2-1.3.1. - Types of Meta-Command
 - 2-1.3.2. - Report Writers
 - 2-1.3.3. - General-Purpose Reduction Commands
- 2-1.4. Tests of Transformation Meta-Command Schemas
 - 2-1.4.1. - Introduction
 - 2-1.4.2. - Easy Examples
 - 2-1.4.3. - Tests of Meta-Command Schemas
 - 2-1.4.4. - Conclusions

2-2. Tests of the Tool-Kit

- 2-2.1. Introduction
 - 2-2.2. Groups involving Fibonacci and Lucas Numbers
 - 2-2.2.1. - Conjecture
 - 2-2.3. Proofs and Derivations using 'tt'
 - 2-2.3.1. - Tracing in 'tt'
 - 2-2.3.2. - An Example: $H(n)$
 - 2-2.3.3. - A Derivation Tool
 - 2-2.4. Choice of Spanning Tree
 - 2-2.5. Optimising Coset Enumeration
 - 2-2.5.1. - Basic Coset Enumeration
 - 2-2.5.2. - Optimising the Reidemeister-Schreier Method
 - 2-2.5.3. - Enumerating in a Related Group
 - 2-2.5.4. - Enumerating over Related Subgroups
 - 2-2.6. Presenting a Subgroup
 - 2-2.6.1. - A Reidemeister-Schreier Process
 - 2-2.6.2. - Todd-Coxeter and Reidemeister Schreier
 - 2-2.6.3. - Presenting the Commutator Subgroup
 - 2-2.6.4. - Subgroups Containing the Commutator Subgroup
 - 2-2.6.5. - Presentations of Normal Subgroups
 - 2-2.6.6. - Subgroups Containing Normal Subgroups
 - 2-2.6.7. - Shell Scripts which Present Subgroups
- 2-3. Using the Tool-Kit
- 2-3.1. Introduction
 - 2-3.2. The Group-Theory Shell
 - 2-3.2.1. - Rationale
 - 2-3.2.2. - Implementation Notes
 - 2-3.2.3. - Examples
 - 2-3.2.4. - Discussion
 - 2-3.3. Goal-Directed Problem Solving
 - 2-3.3.1. - Rationale
 - 2-3.3.2. - Design
 - 2-3.3.3. - Implementation Notes
 - 2-3.3.3.1. - The gp.fg Algorithm
 - 2-3.3.4. - Discussion
 - 2-3.4. Rule-Based Problem Solving
 - 2-3.4.1. - Rationale
 - 2-3.4.2. - Design
 - 2-3.4.2.1. - The 'gp' Database
 - 2-3.4.2.2. - The 'gp' Executive
 - 2-3.4.2.3. - The 'gp' Browser
 - 2-3.4.3. - Implementation Notes
 - 2-3.4.4. - Examples
 - 2-3.4.5. - Discussion

3. *PROOF_OF_A_CONJECTURE*

3-0. Introduction

3-1. A Subgroup of $Y(n)$

3-1.1. - LEMMA

3-1.2. - LEMMA

3-1.3. - COROLLARY

3-2. Some Useful Relations

3-2.1. - LEMMA

3-2.2. - LEMMA

3-2.3. - LEMMA

3-3. The Isomorphism Theorem

3-3.1. - LEMMA

3-3.2. - LEMMA

3-3.3. - THEOREM

4. *CONCLUSIONS*

4-1. Further Tools

4-2. Open Questions

4-3. Summary

APPENDICES

A. DATA_FILE_FORMATS

A-0. Introduction

A-1. P-File

A-1.1. - Examples

A-2. C-File

B. STANDARD_MEMORY_MODELS

B-0. Introduction

B-1. Simple Data Types

B-1.1. - Generator-id

B-1.2. - Coset-id

B-2. Words in a Free Group

B-2.1. - Motivations

B-2.2. - Design

B-2.3. - C-Language Binding

B-2.4. - Implementation Notes

B-3. Coset-id Multiplication Table

B-3.1. - Motivations

B-3.2. - Design

B-3.3. - C-Language Binding

B-3.4. - Implementation Notes

B-4. Coset Table Spanning Trees

B-4.1. - Design

B-4.2. - C-Language Binding

C. *THE_PRIMITIVE_TOOLS*

C-0. Introduction

- C-0.1. - Implementation Details
- C-0.2. - Tool-Set Installation

C-1. Manual Pages for Primitive Tools

C-2. Manual Pages for Library Functions

D. *PRIMITIVE_TOOL_MODELS*

D-0. Introduction

D-1. Primitive Tool Models

E. *REFERENCES*

Acknowledgements

This research was made possible by SERC Grant no. GR/D/03567, which was used by the Pure mathematics Department of the University of St. Andrews to purchase a Cifer 9000 personal microcomputer, running System III Unix*.

I wish to thank my supervisor, Dr Edmund Robertson, for many helpful suggestions and for much support and guidance during the many years of this project. Parts of this thesis have been proof-read by Peter Greenwood, of Computaquest Ltd, by my supervisor and by my wife. I owe them my thanks for tolerance in the face of bad grammar and poorly-expressed ideas.

Finally, my greatest debt I owe to my wife, who has never known married life without the presence of the research documented herein. I wish here to record my deep gratitude for the patience and support she has shown me during the last six years.

Kevin Rutherford.

(Rode Heath, 31-1-89)

* UNIX is a trademark of AT&T Bell Laboratories.

CHAPTER 0

INTRODUCTION

A rationale for the research work described in this thesis is presented. A philosophy for working on group presentations with computers is developed, as an alternative to the existing methodologies embodied in programs such as TC, TTRANS and CAYLEY. The contents of the remainder of this thesis are summarised.

0-1. *Rationale*

The aim of the research presented in this thesis has been to design software which handles group presentations, using an architecture and approach different to that embodied in the popular TC [15], TTRANS [21] and CAYLEY [13,14] programs.

The principal drawback of the existing programs is their lack of flexibility. In TC and TTRANS this is mainly because their user interfaces provide no constructive operators and few meta-level facilities (such as TTRANS' "AU" command). Furthermore, in order to extend the basic problem-solving capabilities of any of these programs, the user has to alter or extend the program itself. There is no easy way to interface external methods, or to combine the programs together to solve new problems.

Below are a few natural questions which cannot easily be answered by the software described above, even though it contains very nearly all of the necessary functionality:

1. In many situations we may wish to test whether two groups are isomorphic, using coset enumeration. For example:
 - ⊕ given a group G and two sets A, B of subgroup generators with A a subset of B , show that the coset tables of A in G and B in G are isomorphic (in the sense of [42]), and thereby show that the two sets generate the same subgroup;
 - ⊕ given two presentations P and Q , in which P presents a factor group of Q , use coset enumeration to show that the two groups are

isomorphic.

However, the interesting cases are usually those in which one of the enumerations completes while the other doesn't. Is it possible to use the coset definitions from the successful enumeration to "seed" the unsuccessful enumeration? That is, is there a coset table spanning tree (or equivalently, a set of Schreier generators for the subgroup) which can act as a skeleton, helping the second enumeration to succeed?

2. Find a presentation for the derived subgroup of a given group G , without finding a set of generating commutators (this case is actually solvable using the extant software, although few users are aware of the tricks required).
3. What is the effect of the choice of coset table spanning tree on the structure of the subgroup presentation generated by the Reidemeister-Schreier algorithm?
4. Are two coset tables isomorphic?
5. Leech [27] describes a method of coset enumeration in which one enumerates over the shorter relators and then uses the longer relators to force coincidences, repeating the process until termination. Are there cases in which the standard HLT method fails where Leech's approach succeeds?

The disadvantages of the monolithic architecture of the above programs can be summarised as restricting their users' ability to solve difficult problems in new ways. The software's design and implementation lack some essential

qualities:

Modularity

One cannot invoke any of the provided functionalities without loading all the others as well. This is wasteful of machine resources, and tends to obscure the fact that many of the functions present are quite logically distinct from each other.

Configurability

The monolithic architecture and overwhelming size of each of the programs is a discouragement to the addition of configurability to existing functions (eg. the provision of different coset defining strategies during Todd-Coxeter enumeration).

Extensibility

In order to extend the functionalities provided, one has to program in FORTRAN. This tends to fudge the distinction between consumer and supplier: almost everyone who uses the software is also required to be an expert in its internal functions, data structures etc. This can affect the way problems are tackled by the user.

Flexibility

It is extremely difficult to use this software in ways which were not foreseen by the original architects.

The thesis of this research is that these difficulties can be overcome by

decoupling the functionalities of these programs into a collection of special-purpose tools. Each tool knows how to perform a single task, and can be treated by the user as an expert "black box". A typical primitive tool might be one which constructs a set of Schreier generators for a subgroup of some group, given a coset table which relates the two groups.

High-level solutions to problems such as those described above can then be implemented using any tool composition language, such as shell scripts. The power of the tool-kit is expected to rest on the freedom of the user to overcome first-order problems (such as the failure of TC to enumerate some example successfully) by the construction of higher-order methods (such as a script which searches for a successful definition strategy). The above primitive tool, which constructs a set of Schreier generators, might be used with other tools which enumerate cosets, perform Reidemeister rewriting, simplify group presentations etc., in a high-level script which calculates presentations for subgroups of a given group. The action of such a script will often be subjective and non-deterministic, making decisions on the basis of high-level user directives or its own internal heuristic knowledge.

The benefits of a modular architecture are well-known, and in this case can be summarised as follows:

Modularity

A certain clarity is gained by forcing small modules to *communicate* data, instead of just sharing variables.

Configurability

If each function of the system is physically (and psychologically) isolated from the others, there are fewer barriers to providing the configurability missing from the existing software. Options can be added to any module without interfering with other parts of the system.

Extensibility

Modules can be added to the set without increasing the overall conceptual complexity of the tool-kit. The users of some parts may be the suppliers of others, reducing the knowledge needed to use the whole system.

Flexibility

If the modules are sufficiently well designed, they can be plugged together in the combination most appropriate to the problem at hand. No assumptions are made by the programmer about the run-time context in which his module will be used. Consequently, the set of solvable problems is enlarged.

However, this paradigm will not be tested if we face larger problems by rebuilding the tools so that they can crunch bigger numbers. Although this may appear to mean that some problems now "go out of range", the problems are better solved by more intelligent scripts. It may thus be possible to augment the basic tool-kit by perpetuating successful problem-solving heuristics.

0-2. *Summary*

The central theme of this research is the flexibility of group-theoretic software. In this Chapter a rationale has been presented in which the ability to solve problems is directly related to the granularity of the available libraries and tool-sets.

Chapter 1 describes the design of a tool-set whose granularity seems well-suited to solving certain types of problem, and to extension by the provision of related tools or libraries of methods. The designs are presented independently of any implementation of the tool-set.

Examples of the use of this tool-set are found in Chapter 2, in which it is shown how these tools can be used to solve a wide range of problems. Individual tools, such as *tc* and *tt*, are shown to give good performance on standard test cases. This is seen to occur because they offer the user the means to combine low-level functions into high-level methods. Possible future developments are also explored, building on the adaptability of the tool-set in order to design an intelligent, rule-based system which can automatically solve simple problems involving group presentations.

Chapter 3 proves a conjecture of Campbell and Robertson. The proof is a generalisation of a proof for certain special cases, which was found by the above tool-kit during re-examination of well-known test cases for Tietze transformation software.

Chapter 4 presents some conclusions drawn from this work.

The Appendices describe aspects of a particular implementation of the tool-

set. The implementation of the primitive tools comprising the group theory tool-set rests upon a library of data file support, i/o, memory management and operating system independence. This library, and the data structure and external file interfaces to it, are defined in Appendices A and B.

Appendix C contains a specification for each member of the group theory tool-set, in the form of Unix-style manual pages. Manual pages for a C-language binding to the tool-set's support library are also given here.

CHAPTER 1

TOOL-SET DESIGN

Designs for each of the primitive group-theoretic tools are described here in detail. Each tool relies on standard file formats and data structures, so that the group-theoretic software can be packaged as functions which are operating-system independent. The designs are described independently of their implementations.

1-0. *Introduction*

Each of the tools described in this Chapter embodies an algorithm or method which has been found to be useful when working with group presentations. Each of the Sections below describes the design of one of these tools, independently of any specific implementation. Appendix D summarises these designs pictorially. Details of one particular implementation of these tools can be found in Appendices B and C.

1-0.1 *Primitive_Tool_Architecture*

The principal design aim in this collection of tools has been to isolate the group-theoretic software into reusable modules. These modules should be essentially independent of the host operating system, of the tool into which they are linked, and of the run-time context in which they are invoked. The modules therefore depend on a library of functions which perform i/o, allocate and free memory, manage data structures and output trace statements. Each module operates on structured data objects of certain types.

The modules are collected together into tools. Each tool orchestrates the actions of a collection of modules on a collection of data objects. The tools communicate with the run-time environment using functions which can translate the structured objects into files.

The library of support functions plays an important role in the design of the primitive tools, and in their integration to form a working tool-kit. By providing standard abstract data types and operating system independence, the library makes the tool-kit more flexible:

software portability -

porting a few functions effectively ports a large number of tools;

data portability -

sites using the library can exchange data files easily, knowing that the library will ensure correct interpretation of their contents;

reusability -

new algorithms or improved techniques, implemented as library modules and interfacing with the standard memory models, can be exchanged with little porting effort;

ease of integration -

new functions are easy to slot into old situations if they interface using the common models.

The usefulness of the tool-set hinges on the granularity chosen by the tool-builder. The particular set of tools presented here is fairly fine-grained, in that most tools contain only one module. For instance, coset enumeration over a non-trivial subgroup has been designed as two tools, instead of as one tool containing two modules. Similarly the Reidemeister-Schreier method of presenting a subgroup splits naturally into two tools, which post-process the results of the coset enumeration tools. Chapter 2 describes a few ways in which these tools may be combined to solve problems involving group presenta-

1-1. *Orderings_and_Canonical_Forms*

This Section describes the primitive tool *canon*, which is used to alter the order and form of generators and relators in a group presentation $P = \langle X \mid R \rangle$. All of the functionality of the tool is embodied within the library function *canon()*, which is employed in various primitive tools (most notably *tt*).

1-1.1 *Motivations*

A canonical form for group presentations is defined in [20] and used in [20,21]. The ordering appears to be quite a sensible one for applications such as coset enumeration and Tietze transformations, because:

Coset Enumeration:

- ⊕ keeping the relators in ascending order of length might cause critical coincidences to occur sooner [15];
- ⊕ free reduction of the presentation reduces the number of redundant cosets defined.

Tietze Transformations:

- ⊕ the canonical form and ordering allows duplicate relators to be easily detected and removed;
- ⊕ keeping the relators in ascending order of length permits a most effective use of substring searching, using short relators to perform eliminations;

However, several other properties of the form of a group presentation may also be important:

- ⊕ the group generators might be best listed in ascending or descending frequency order, so that Todd-Coxeter defining strategies (eg. fill method G, Section 1-3.2.3) have the most effect, or so that construction of a set of Schreier generators can introduce trivial redundancies into the Reidemeister relators;
- ⊕ the relators might be best written as the inverses of their canonical forms, for instance to allow Todd-Coxeter defining strategies to fill the relator tables in "reverse order";
- ⊕ it is sometimes advantageous to rotate some of the relators so that long common strings are shifted to the front of two or more words. This can speed up the Todd-Coxeter algorithm [15];
- ⊕ the notion of relator length, as used in [20,21] to define the ordering inductively, could be replaced by other notions. Candidate "length functions" include ones which ignore powers of generators, or which apply weights [41] to certain generators.

As always, the most flexible approach here would be to provide an interactive tool which allowed the user to re-order the generators and relators in any way he chose. However, since our aim is the provision of tools which can be used by other programs automatically, we implement a simple function which embodies some of the above criteria as user-selectable algorithms.

1-1.2 *Word_Length*

This Section describes the various ways we use to calculate the "length" of a word in a group.

1-1.2.1 *Rationale*

Let G be a group, with presentation $P = \langle X \mid R \rangle$; let X^+ be the inverse closure of X , and let $F(X)$ be the free group generated by X . Let $w = x_1^{a_1} x_2^{a_2} \dots x_n^{a_n}$ be a word in the generators of G and their inverses, such that $a_i > 0$ and $x_i \neq x_{i+1}$ for all $1 \leq i \leq n$. Let $wt : X^+ \rightarrow Z$ be an arbitrary integer-valued function on the group generators. We define the following three functions:

$$len : F(X) \rightarrow Z \quad len(w) = \sum_{i=1}^n a_i$$

$$blen : F(X) \rightarrow Z \quad blen(w) = n$$

$$wlen : F(X) \rightarrow Z \quad wlen(w) = \sum_{i=1}^n a_i \cdot wt(x_i)$$

Remarks:

- ⊕ $len(w)$ is called the *physical* length of w ;
- ⊕ $blen(w)$ is the *base* length of w , and is an *effective* length;
- ⊕ $wlen(w)$ is the *weighted* length of w with respect to wt , and is also an *effective* length;
- ⊕ the model of weighted lengths was presented in [41], in a somewhat more restricted form than that shown here;

⊕ let $f(w)$ be the free reduction of w , that is, with all terms $x.x^{-1}$ removed. Then $len(f(w)) \leq len(w)$ and $blen(f(w)) \leq blen(w)$. For example, suppose that $X = \{x, y\}$ and wt is defined as

$$\begin{aligned} wt(x) &= -3 & wt(y) &= 10 \\ wt(x^{-1}) &= -3 & wt(y^{-1}) &= -10 \end{aligned}$$

Then for $w = yxx^{-1}y$ we have:

	w	$f(w)$
len	4	2
blen	4	1
wlen	14	20

⊕ if we express w as the product of two words in G , $w = u.v$, then clearly

$$\begin{aligned} len(w) &= len(u) + len(v) \\ wlen(w) &= wlen(u) + wlen(v) \end{aligned}$$

However, if $u = x_1 \dots x_i$ and $v = x_i \dots x_n$ then

$$blen(w) = blen(u) + blen(v) - 1$$

Otherwise

$$blen(w) = blen(u) + blen(v)$$

⊕ none of the effective length functions described here is the same as the "length" used in [20,21,41], which is the un-exponentiated length of a relator.

1-1.2.2 Design

During the operation of a primitive tool (which is the only time during which a presentation's canonical form may be examined or altered by software) we provide support for the notion of a *current* length function. If the length of

a word in a group is measured with respect to the current length function, the resulting integer is the current *effective* length of the word.

The global variable *lengthFunction* will always contain a token which describes the current length function, taking one of the symbolic values *PHYSICAL*, *BASE* or *WEIGHTED*. The function *elength()* calculates the effective length of a word, with respect to the current length function.

A C-language binding and implementation of this design is described in the *length(3)* manual page, Appendix C.

1-1.3 *The_canon()_Function*

The function *canon()* applies to a Presn structure consisting of a list of named generators and a list of words in the free group on those generators. Any or all of the words may also be relators, and therefore may be replaced by any cyclic permutation of the constituent generators. In what follows, we present a variety of "canonical forms" for a group presentation $P = \langle X \mid R \rangle$. For most P , many of these forms will be identical. The descriptions will be couched in terms of group relators, although at any time the data structures may actually contain words which cannot be cyclically permuted.

Each of the forms is best described as a mechanism, rather than as a static form. The mechanisms all fall into a general pattern, as follows:

- a. every relator is freely and cyclically reduced. Any which are then trivial are removed from the list;
- b. the free group generators are processed, usually by sorting;

- c. the relators are pre-processed;
- d. the relators are processed;
- e. the relators are post-processed;

Each of steps b-e may be "turned off". Consequently, every form is freely and cyclically reduced, but anything else depends upon user selection. The remainder of this Section describes the various actions which may be selected within each of the above phases b-e. A C-language binding for the *canon()* function is given in manual page *canon(3)*, while *canon(1)* is a user guide for the *canon* tool. Both are listed in Appendix C.

1-1.3.1 *Generator_Processing*

This phase simply sorts the generators, according to exactly one of the following selections:

- l the generators are not sorted;
- r the order of the generators is randomised;
- a the generators are sorted into ascending alphabetical order according to their names in the namelist (see Appendix B-2). The result depends upon the collating sequence used in the host environment. The implementation described in *canon(3)* uses the standard ASCII collating sequence;
- f the generators are sorted into ascending order of frequency in the current list of words.

1-1.3.2 *Relator_Pre-Processing*

This phase prepares the presentation, ready for relator processing, by the application of none, some or all of the following:

- g the order of the group generators is reversed;
- r each relator is cyclically permuted until it represents the minimal word under the collating sequence defined by the current order of the group generators. For instance, if the group generators are currently

$$x, a, b, c$$

then the form x^2abc is "less than" the form bcx^2a ;

- R each relator is cyclically permuted by a random amount, which is likely to be different for each relator;
- b if the current length function is *blen* each relator is rotated (if necessary) until its effective length is minimal. For instance, the relator

$$xabcx$$

has effective length 5, whereas the form

$$abcx^2$$

has effective length 4.

1-1.3.3 *Relator_Processing*

This phase traditionally represents the principal "canonicalisation" of the group presentation. The relators are re-organised into one of the following forms:

r the relators are ordered randomly. The actual form of each relator is unchanged;

m the relators are sorted and permuted so as to maximise the occurrence of common strings at the beginnings of relators. The relators with the longest common substring are brought to the head of the list and permuted until each has the common string at its front. They are then sorted into increasing order of effective length, with equal length relators further sorted using the current ordering on the generators. If two sets of relators feature common substrings of equal length, the set with the "least" relator (by effective length and generator collation) is treated first. The remaining relators are then treated similarly. For instance, using the free length as our length function, and given the ordering

$$x, y, a, b, c$$

on a set of group generators, then the relators

$$yabc^2, a^2bcb, xabcay, x^3bcab$$

have the following sorted form:

$$abcba, abc^2y, abcayx, bcabx^3$$

l the relators are sorted into order of increasing effective length. Any relator which is found to be identical to another is removed. Relators of equal effective length are sorted into ascending order under the collating sequence derived from the current generator ordering;

1-1.3.4 *Relator_Post-Processing*

This phase applies none, some or all of the following functions to the form output by the previous phase:

- r the order of the relators is reversed;
- i every relator is inverted;
- g the order of the generators is reversed.

1-1.3.5 *Remarks*

- ⊕ relator ordering *m* is clearly only one way to organise such a matching form. For instance, other treatments might order relators pairwise, instead of concentrating on matching sets as above. The choice seems arbitrary, and therefore other treatments should be implemented at some stage;
- ⊕ perhaps the decision to remove identical relators should be optional, but it is difficult to imagine why;
- ⊕ the post-processing option which reverses the generator ordering could be achieved by piping two instances of *canon* together, as can many other esoteric "canonical forms";
- ⊕ an alternative implementation of the *canon* primitive tool is shown in Appendix D-1, as a shell script which specialises *tt* into a filter.

1-2. *Tietze Transformations*

This Section describes the design of the primitive tool *tt*, which allows the "editing" of a group presentation *P* using Tietze and Nielsen transformations.

1-2.1 *Motivations_and_Requirements*

1-2.1.1 *Motivations*

The motivations which have driven the design of *tt* all stem from the desire to improve the program TTRANS [21,41], both in terms of the results it can produce and in its ability to behave well as an interactive tool. Specific motivations arise from specific aspects of the design of TTRANS:

- ⊕ the *editing* commands of TTRANS, those which actually alter the current presentation, chatter to the user about what they are doing. This is not necessarily a bad thing, especially if the operation takes a long time, but some degree of configurability in the diagnostic sub-system might significantly improve the user's view of the system's intelligence;
- ⊕ the actions of TTRANS are always repeatable. Although this is usually desirable, the injection of a little randomness may occasionally improve results;
- ⊕ the length function *wlen* may be too powerful to be useful in *tt*, unless the set of weights is restricted somehow.

TTRANS, in the form currently available on the Vaxen at St.Andrews, provides a few, pre-programmed, meta-level commands. These use the other commands in various ways to achieve automatic reduction of the current presentation.

- ⊕ the goals of these meta-commands are perhaps restrictive. A better system might permit presentations other than the "shortest" to be sought;
- ⊕ a system which had no built-in meta-commands, but instead had a simple way to construct them interactively, would perhaps be more flexible.

1-2.1.2 *Requirements*

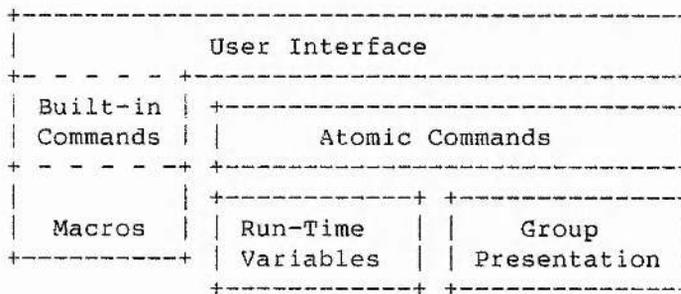
- ⊕ *tt* must integrate well with the environment, in this case the primitive tool-set, by using file formats, memory models and software which are common with the other tools;
- ⊕ in the main, most applications could possibly be tackled using only one or two basic meta-command designs. The user must be able to provide these in some stable form, for perpetual re-use;
- ⊕ the commands which actually transform the presentation must be sufficiently low-level that the user can combine them easily in different ways; the means of combining commands must be natural, arising from an analysis of the meta-command(s) provided by TTRANS [21];
- ⊕ the notion of word length must be generalised from simple use of the free length, as in TTRANS, to permit the user to direct the "shortening" of a presentation down a variety of paths.

1-2.2 *High-Level Design*

1-2.2.1 *General_Architecture*

tt is modelled on the Unix tool *ed(1)*, which is a simple line editor. A file is copied into memory, edited under interactive instruction from the user, and the resulting version is written out to file again (if the user so wishes). In this case, the file is a P-file and the memory model is a Presn.

The following schematic summarises the architecture of *tt*:

1-2.2.2 *Length_Functions*

The length functions supported within the primitive tool-set are fully described in Section 1-1.2. However, in *tt* we have decided to restrict the provision of weights for *wlen*, so as to be more manageable. Practical experience suggests that only two weight values are required at any time, especially if the user has the facility to alter them during the course of any operation.

Consequently, in *tt* we introduce the notion of *tags*. A generator and its inverse are always both tagged or both untagged. Whenever the *wlen* function is in use, all tagged generators have a particular weight, while all untagged generators have a different particular weight. The two weights are chosen by the user.

The user's choice of which generators to tag can also be used to alter the criteria for eliminating redundant generators.

1-2.2.3 *Tietze Transformations*

This Section describes the design of the atomic commands which provide the functionality of the Tietze transformations.

X+ Adding a Redundant Generator

This Tietze transformation is not supported directly in *tt*, but can be modelled using the 'newgen' and 'addrel' commands.

X- Removing a Redundant Generator

The function *eg()* lies at the heart of every method provided by *tt* for the removal of generators. *eg()* takes a generator $x \in X$ and a relation $x = w$ as parameters. x is then replaced everywhere in P by the word w . Finally, x is removed from X .

tt provides the means by which the user can apply *eg()*, under various circumstances, to the presentation P by the notion of "eliminator selection". An *eliminator* is a pair, consisting of a generator and a relator, such that the generator or its inverse occurs exactly once in the relator. In any presentation there are often a great many eliminators, and any one of them may be the "right" one to apply in order to achieve the desired transformation $X-$. The eliminator selection method applies a set of criteria in turn to the set of eliminators, until no further criteria remain. At that stage, if exactly one eliminator has passed every test it will be

used to supply the parameters to `eg()`. Criteria are chosen from the following set:

- a keep only those eliminators involving a generator with minimal frequency in the relators of P ;
- A keep only those eliminators involving a generator with maximal frequency in the relators of P ;
- b keep only those eliminators involving a generator which occurs in the fewest relators of P ;
- B keep only those eliminators involving a generator which occurs in the most relators of P ;
- c keep only those eliminators involving a relator with the least free length of those in P ;
- C keep only those eliminators involving a relator with the greatest free length of those in P ;
- d keep only those eliminators which would increase the length of P the least (in a very naive sense). That is, keep the eliminators for which the frequency of the generator multiplied by the length of the relator is minimal;
- D keep only those eliminators which would increase the length of P the most (in a very naive sense). That is, keep the eliminators for which the frequency of the generator multiplied by the length of the relator is maximal;

- e keep only those eliminators involving a relator with the least effective length of those in P , according to the current length function;
- E keep only those eliminators involving a relator with the greatest effective length of those in P , according to the current length function;
- t keep only those eliminators involving a tagged generator;
- T keep only those eliminators involving an untagged generator;
- l keep only the first eliminator;
- ? keep one eliminator at random;

See the descriptions of the 'c', 'g' and 'G' atomic commands for the use of these elimination selection criteria.

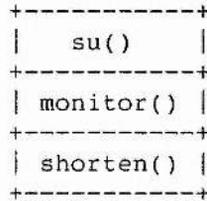
R+ Adding a Redundant Relator

The 'addrel' command allows the user to add a relator to the current presentation, but *tt* cannot validate whether this relator can be derived from the existing relators. Alternatively, the 'su' command (see next item, R-) allows both R+ and R- to be performed together in a simple, decidable function.

R- Removing a Redundant Relator

As before, this transformation is not easily achieved except in the context of a slightly less atomic action called "String Replacement". In

general terms, String Replacement is analogous to the "substring searching" of TTRANS, but so many details are different in this version that it is worth covering all of the ground again. The method has the following architecture:



where the functions have the following objectives:

shorten(rels, x)

x is a *monitor point*, *rels* is a list of relators. *x* is known to be some point in a relator *mon* (which is not in the list *rels*), ie. *x* is a pointer to a Letter in a doubly-linked circular list of Letters. *shorten()* takes each relator *rel* on the list *rels* and compares it with *mon*, by first looking for points which contain the same generator as at *x* and then looking for a matching word either side of this point (this is exactly the same mechanism as that described in [21]). We now have the following relators:

```

mon: axbv
rel: axbw

```

where *v* and *w* are arbitrary words. If conditions are right the following Tietze transformations are now performed:

R+ add the relator $v^{-1}w$

R- remove relator *rel*

In practice, the relator *rel* in the current Presn structure is simply rewritten so as to have the same net effect as the two transformations (this is the *substitution* step). *shorten()* now continues, trying to alter *rel* further. The remaining relators in *rels* are then examined in the same way.

monitor(rels, r)

r is a *monitor* relator, *rels* is a list of relators. The *monitor()* function selects a few points on *r*, passing them to *shorten()* as monitor points. Two points have particular interest to *monitor()*:

m1: the first letter of the relator (in its current cyclic permutation);

m2: its "antipode", halfway along the word.

As explained in [21], use of these two points is all that is necessary when searching for string matches which are over half the length of the monitor relator, and when the free length is the only length function in use.

su()

This function organises the use of *monitor()*, selecting relators in turn to serve as monitors. In every case, the list of relators to be passed to *monitor()* is the list of relators which immediately follow

the monitor in the current Presn.

The actions of these functions can be configured by the user, by means of key-letter flags. The complete list of configuration flags is:

- x test mode: everything is performed normally, except that the substitution step is not carried out. This mode can be useful with tracing enabled, to understand what substitutions might occur without detrimentally affecting the current presentation;
- < the substitution step is allowed whenever the effective length of *v* (above) is strictly less than that of *axb*;
- = the substitution step is allowed whenever the effective length of *v* (above) is equal to that of *axb*. Note that the '=' and '<' flags are independent of each other. Each may be used independently of the other, or they may be used together to mean "less than or equal";
- ? if an equal-length match is found, decide whether to actually carry out the substitution by "tossing a coin";
- 1 allows the use of monitor point *m1*;
- 2 allows the use of monitor point *m2*;
- a selects every point as a monitor point. This option is useful when length functions other than the free length are in use;
- o *monitor()* offsets the selection of monitor points by a random amount

before starting to call *shorten()*;

r the point at which *shorten()* begins to look for matches in *mon* is offset by a random number of letters;

l the relators in the current Presn are used in reverse order as monitor relators. This option does not affect the fact that the list of *rels* passed to *monitor()* consists of those relators which follow the monitor relator; the complete reversal procedure can be achieved using *canon* to reverse the relators prior to calling *su*;

c if this flag is specified, *su()* does not give up as soon as a monitor relator alters the Presn, but continues until every relator has been used as a monitor.

The implementation of substring searching in TTRANS is equivalent to the flags "<l2c", together with a loop around the *su* command itself.

1-2.3 Detailed_Design

1-2.3.1 User_Interface

The user interface of *tt* has been designed so as to be completely independent of the actual set of commands offered by *tt* itself. It acts as a mediator, invoking the services of *tt* on the user's behalf, and providing input structuring mechanisms such as composite commands and macro commands. As such, it presents the user with a command language interpreter, whose lowest-level operations are provided by a library of presentation-specific functions.

1-2.3.1.1 *Language_Structure*

The language interpreter reads user commands from standard input, until end-of-file is reached (this is usually achieved by the user typing ^D at the keyboard, or by the end of an input script file being reached). The command language of *tt* is defined by the following BNF syntax (see Appendix A for an explanation of the meta-syntax used here):

```

script ::= <cmds> EOF
cmds   ::= <cmd> [<sep> <cmd>]*
cmd    ::= [INTEGER] <instr>
instr  ::= <ident> [<arg>]* | <block>
block  ::= { <cmds> } | <sbra> <cmds> <sket>
sbra   ::= [
sket   ::= ]
ident  ::= <letter> [ <letter> | <digit>]*
arg    ::= <string> | <word> | $ <word>
string ::= " [<char> | <white>] "
word   ::= <char>*
char   ::= <letter> | <digit> | <sbra> | <sket>
        | !|^|&|*|(|)|{|}|:|'|"|,|.|<|>|?|_|+|-|=|~|`
white  ::= SPACE | TAB
sep    ::= NEWLINE | ;

```

Input is structured into *commands*, each of which may be *simple* or *composite*. Each command (of either type) has an *exit status*, which reports "success" (1, *true*) or "failure" (0, *false*) to the *tt* executive.

Each command may be preceded by an integer which indicates the number of times the command is to be carried out. However, the repeat count is overridden as soon as the command fails, causing immediate termination. A repeat count of 0 indicates that the command is to be executed until it fails. An unspecified repeat count defaults to 1. The exit status of a repeated command is *true* if the command succeeded at all. Thus, if a command with repeat count 6 succeeds only twice, the exit status is still 1.

1-2.3.1.2 *Simple_Commands*

Every simple command has the syntax

```
[ <count> ] <cmd> [ <arg> ]*
```

where each <arg> is a text string, separated from the next <arg> by any amount of white space. An <arg> may be replaced by the value of a variable if it begins with a '\$' symbol. In order to allow white space to form part of an <arg>'s value, double quotes may be placed around any text, turning it into a single string. Within a quoted string, the back-slash character acts as an escape. For instance,

```
"The value of \"filename\" is \"fred.p\"\\n"
```

is a valid string (whose last character is a new-line). Simple commands come in the following varieties:

Atomic Commands

These are the commands supplied by *tt* itself. They perform tasks related to the actual job in hand, which in this case is the editing of a group presentation. Each atomic command is called (by the user interface executive) as a function with two arguments. The first is a parameter count; the second is a list of the text strings which are the parameters themselves. A C-language binding for such a function is therefore of the form

```
int cmd(argc, argv)
int argc;
char **argv;
```

Each must be implemented so as to return an exit status: 1 for success, 0

for failure. However, the meanings of these status values in each case are purely arbitrary.

Macro Commands

The user interface provides the means by which the user may extend the set of simple commands. A *macro* is a simple command which has been defined by the user. It is governed by exactly the same rules as any other simple command, in that it has an exit status and can be invoked with a repeat count. The exit status of a macro command is defined to be the exit status of the final command in its replacement text.

A macro may take arguments, just as any other command. They are specified in the definition of the macro by \$1, \$2, etc. (unlike other simple commands, a macro is restricted to 9 arguments by the macro processor). Parameter values are substituted into the replacement text before the text is executed, so the content of the macro cannot change during its execution.

Built-In Commands

The user interface itself provides some commands, which are:

```
define [name text]
```

defines the macro command *name*. When *name* is invoked as a simple command, its replacement *text* will be inserted into the input stream as if typed by the user. If the macro *name* already exists, its value will be replaced by *text*. If *text* contains any of the symbols '\$1'

through '\$9', these are replaced by their corresponding actual parameter values when the macro is interpreted.

If the parameters to 'define' are not given, the current set of macro definitions is printed out.

undefine name

removes *name* from the list of macro commands;

set var value

sets the value of the run-time variable *var* to *value*;

source file

switches the interpreter's input file to *file*. Commands are read from this file until end-of-file is reached, at which point input switches back to the previous stream. *source* commands can be nested upto 20 files deep, allowing the possibly of nested files of macro definitions etc.

Each of these commands always exits with *true* status, except in the case that the *file* argument to *source* cannot be opened for reading.

As far as the user is concerned, each of these command types behaves in exactly the same way, and each can be used anywhere as a simple command.

1-2.3.1.3 *Composite_Commands*

The BNF syntax given above provides two kinds of composite command:

OR-block Syntax: { cmd ; ... }

An OR-block succeeds (exits with status *true*) if *any* of the commands in the block succeeds;

AND-block Syntax: [cmd ; ...]

An AND-block succeeds only if *every* command in the block succeeds.

In each case, every command in the block is executed before the exit status for the block is computed.

Repeat counts apply to composite commands exactly as to simple commands, using the exit status of the block as defined above.

1-2.3.2 *Run-Time_Variables*

In *tt*, the run-time variables form a set of atomic data entities. Each carries a single value, which is always a text string (although the string may represent an integer or anything else).

The user signifies a variable as a parameter to a command by the usage '\$name'. Just prior to the execution of the command, each variable is replaced in the command's parameter list by the value of the named variable. The syntax '\$name' will be ignored if it occurs anywhere other than as a (complete) command parameter.

There are two types of variable in *tt*, corresponding to two different usages.

1-2.3.2.1 *Read-Only_Variables*

Variables of this type provide a way to extract information about the current presentation, without introducing hundreds of special commands. The full set of read-only variables in *tt* is:

minlen the free length of the physically shortest relator;
maxlen the free length of the physically longest relator;
totlen the total free length of the Presn;
etotlen the current total effective length of all the relators;
nrels the current number of relators;
ngens the current number of generators.

Their values are re-calculated each time they are required. The set is by no means exhaustive, but is nevertheless fairly representative of the kind of data required by the user during the more common algorithms.

1-2.3.2.2 *Read-Write_Variables*

Read-write variables offer the user a simple way to configure the actions of the atomic commands. The full set of read-write variables in *tt* is:

filename contains the name of the P-file which is currently being edited.
The atomic commands *e* and *w* use the value of this variable when they are given no argument;

`lenalg` contains the "name" of the current length function to be used within `tt`. Legal values for this variable are restricted to:

- `free` the free length of words is used whenever they or their lengths are compared;
- `base` the function `blen` is used as the current length function;
- `tagged` the `wlen` length function is used. Generators which have been tagged (see the 'tag' atomic command) have the weight specified by the run-time variable `$tag`, while untagged generators (see the 'untag' atomic command) have weight `$notag`;

The value of this variable affects the working of the 'canon' and 'su' atomic commands, as described elsewhere;

`tracelevel` the current global trace level. This value overrides the environment variable `$TRACE` (see `trace(3)` for further details);

`tracefile` the name of the current output file for trace statements. The value overrides the environment variable `$TRACEFILE`;

`tag` the current weight value for tagged generators and their inverses. Until altered by the user, this variable defaults to weight 10;

`notag` the current weight value for untagged generators and their inverses. Until altered by the user, this variable defaults to weight 1;

1-2.3.3 *Atomic_Commands*

The following is the complete list of atomic commands provided by *tt* at present:

?

displays a help screen to the user, listing this set of atomic commands;

addgen name

creates a new generator, adding the given *name* to the current Namelist. This operation is not a Tietze transformation, although it can be used in conjunction with *addrel* to form a macro command which is a Tietze transformation (see Section 1-2.4 for an example);

addrel word

adds the word *word* to the current list of relators. This is not necessarily a Tietze transformation, and *tt* will not validate the new relator;

c [crit]

uses "eliminator selection" to remove a redundant generator from the current presentation. The selection criteria are represented by key-letters in the first parameter to 'c', as defined in Section 1-2.2.3. If no criteria are given, this command reports the current number of eliminators. 'c' fails if the criteria select other than one eliminator;

`canon [args]`

canonicalises the presentation. The arguments behave exactly as for the `canon(1)` primitive tool, except that `'-l'` need not be specified because of the `$lenalg` run-time variable;

`e [file]`

begins a new edit. The current Presn is discarded, a new P-file is opened and loaded into a Presn memory model (by the `readPresn(3)` function). If the argument is present, it is taken to be the name of a suitable P-file. Otherwise, the value of `$filename` is used;

`echo [arg]*`

the arguments, if any, are evaluated and written to standard output. A new-line character is then written. This command is useful within macros, to provide commentary on the current activity;

`g [gen]*`

removes the named generators from *P*, if possible. Each named generator is used, in turn, to create a set of eliminators; one of these is then selected, using criterion "1", and is used to remove the generator. If any generator cannot be eliminated, either because it doesn't exist or occurs more than once in any relator, the command fails immediately. If no generators are named, criterion "1" is applied to the complete set of possible eliminators, failing if there are no eliminators at all;

`G len`

uses relators whose free length is less than *len* to eliminate

generators. Each relator of the appropriate length is taken, in turn, and forms a set of eliminators. Criterion "1" is applied to these and a generator is removed. The command fails if no generator was removed at all;

`invert gen [gen]*`

replaces the named generators by their inverses throughout the presentation. This is a Nielsen transformation;

`p`

prints the current presentation in standard P-file syntax. However, tagged generators are highlighted wherever they occur;

`print format [arg]*`

prints the string *format*, replacing occurrences of the '\$' symbol by the remaining arguments, in turn. Note that *format* is output literally: no new-line is appended to the output. So multiple *print* statements can be used to build a single line of output;

`rename gen1 gen2`

changes the name of generator *gen1* to be the string *gen2* in the `Namelist`;

`su [flags]`

attempts to reduce the effective length of one or more relators, using the string replacement algorithm outlined above. If the *flags* are not supplied, the default action is "<12". *su* succeeds if and only if the presentation was altered. In practice, this can mean

that `su` succeeds with no net effect, when two substitutions cancel each other out, but this is rare;

`tag [gen]*`

sets a tag flag against the named generators. These generators will now appear high-lighted in the display produced by `p`, and will have weight `$tag` under the `WEIGHTED` length function. They may also be affected by the selection criteria under the `'c'` command;

`untag gen*`

removes the tag flag from the named generators. These generators now revert to weight `$notag` under the `WEIGHTED` length function. They may also be affected by the selection criteria under the `'c'` command;

`w`

writes the current presentation to `file`, in P-file syntax. If `file` is not specified, the value of `$filename` is used.

1-2.4 Examples

```
define SU "12 su; echo Total length = $totlen"
```

defines a high-level version of the `su` command, which runs it 12 times and then prints out the length of the resulting presentation. Note that this macro will always succeed, because `echo` always succeeds. Note also that `"$totlen"` is not a valid macro argument marker, so is passed into the replacement text un-altered.

```
define SU "$1 su; echo Total length = $totlen"
```

is a slightly more general version, which takes the repeat count for *su* as a parameter.

```
define newgen "addgen $1; addrel \"\$1 = \$2\""
```

defines a function which is a Tietze transformation, defining a new generator and adding a relator which contains it to the presentation. So the usage

```
newgen "f" "abcx-1"
```

causes generator *f* to be added to the Namelist, and appends the relation $f = abcx-1$ to the presentation.

1-3. *Cosets_and_Enumeration*

This Section describes the primitive tool *tc*, which houses a number of functions dealing with the cosets of some subgroup of a given group. The principal use of *tc* is the enumeration of the cosets of the subgroup, essentially using the Todd-Coxeter method [45].

1-3.1 *Requirements_and_Rationale*

We can place the following requirements on the tool *tc*:

- ⊕ it must be possible to enumerate the cosets of a subgroup for which one has no (convenient) set of generating words;
- ⊕ it must be possible to "re-start" an enumeration using the partial results from a previous attempt;
- ⊕ the tool must integrate well with the rest of the tool-set, both in terms of physical data interchange and in its general philosophy;
- ⊕ *tc* must allow the user to *fully* specify the definition strategy which is to be used within the Todd-Coxeter enumeration method;
- ⊕ *tc* must promote the distinction between the cosets of a subgroup (which are defined by the group and subgroup under examination) and the objects which are used to denote them during the running of a process. Several of these latter (*coset-ids*) can often turn out to denote the same actual coset;
- ⊕ if the implementation is to be extended (by the addition of new defini-

tion strategies, for instance), its architecture must clearly separate and support the variety of activities for which the tool may be used;

- ⊕ the tool must have a sufficiently low-level interface that varieties of user-model can be built around it. Consequently, the tool must not be interactive.

The design of the primitive tool *tc* will be heavily dependant upon a set of library functions which manipulate coset-ids and their related data structures. Most of this Section consists of a design description of these functions, which fall into four general categories:

Coset-id equivalence

Some coset-ids refer to the same actual coset. An equivalence relation on coset-ids is defined, and is implemented as a transversal function;

Coset-id sufficiency

At any time, the current collection of coset-ids may not be sufficient to name every coset (as far as we know). Sufficiency conditions are stated, and implemented as a Boolean output value from the above transversal function;

Coset-id definitions

If one's ultimate aim is the enumeration of a set of cosets, one must introduce enough coset-ids to name every coset. A range of definition

strategies is provided;

Subgroup definition

The subgroup whose cosets are to be enumerated can be defined existentially, by the effect it has on the action of generators on cosets. A technique for implementing this effect is described.

The primitive tool *tc* co-ordinates these functions in order to enumerate the cosets of a subgroup of some group:

Coset enumeration

The Todd-Coxeter method of enumerating cosets is expressed in terms of the above four mechanisms. The high-level design of *tc* is a realisation of this model, and is seen to meet the requirements stated above.

Before venturing into a high-level design description of *tc* and the library functions which support it, we give brief definitions of the above five operations on cosets and coset-ids.

Let G be a finitely-presented group with generating set X and presentation $P = \langle X \mid R \rangle$. Denote by X^+ the inverse closure of X . Let H be a subgroup of finite index in G and let C be the set of cosets of H in G (hereafter referred to simply as *cosets* for clarity).

Let $t : N \rightarrow C$ be any finite partial function such that $t(1) = H$ and $t(0)$ is undefined. Let T be the inverse image of t on C . Call the elements of T *coset-ids*. T is finite.

Let '.' signify the multiplication operation in G . Let $f : T \times X^+ \rightarrow T \cup \{0\}$ be any multiplication function for coset-ids which satisfies

$$f(i, g) = 0 \quad \text{or} \\ j, \quad \text{for some } j \in T \text{ such that } t(i).g = t(j)$$

for all $i \in T$ and $g \in X^+$, subject to the *consistency condition* that

$$f(i, g) = j \in T \text{ iff } f(j, g^{-1}) = i.$$

j is called the *image* of i under the action of g .

Let $w_1 = g_1.w_2$ and $w_2 = g_2 \dots g_n$ be words in G , and suppose that $f(i, g_1) = m \in T \cup \{0\}$. We can (recursively) define an action f' of words on coset-ids by extension of f :

$$\begin{aligned} f'(i, w) &= 0 && \text{when } m = 0 \\ &= m && \text{when } w_2 = 1 \\ &= f'(m, w_2) && \text{otherwise} \end{aligned}$$

A coset-id $i \in T$ is said to be *I-closed with respect to X* (or just *I-closed*) when the image $f(i, g) \neq 0$ for all $g \in X^+$.

A coset-id $i \in T$ is said to be *closed with respect to a relator $r \in R$* when the image $f'(i, r) \neq 0$. A coset-id i is *R-closed with respect to P* (or just *R-closed*) when i is closed with respect to every $r \in R$.

1-3.1.1 Coset-Id_Equivalence

Define an equivalence relation e on T by

$$i \sim j \text{ iff } t(i) = t(j).$$

The coset-ids i and j are said to be *coincident*.

Let U be any transversal of T/e , and let A be an equivalence class of T/e with representative $a \in U$. Then U has the following properties:

1. if there exist $b \in A$ and $g \in X^+$ such that $f(b, g) = j \neq 0$ then we can state that $f(a, g) = j$ and $f(j, g^{-1}) = a$;
2. if any $b \in A$ is I-closed then a is I-closed;
3. if any $b \in A$ is closed at some relator $r \in R$ then a is also closed at r ;
4. if any $b \in A$ is R-closed then a is R-closed.

1-3.1.2 Coset-Id_Sufficiency

t , restricted to U , is an injection. Thus if t is surjective, the cardinality of U equals the index $[G:H]$. The following statements are all equivalent:

1. t is surjective;
2. there exists no $g \in X^+$ and $i \in T$ such that $f(i, g) = 0$;
3. for every relator $r \in R$ and coset-id $i \in T$ we have $f'(i, r) = j$, for some coset-id j such that $t(i) = t(j)$.

1-3.1.3 Coset-Id_Definition

We can define a function f_2 , related to f by the "definition" of a coset-id n , as follows:

- a. Let $n \in N - T$;
- b. Define a new set of coset-ids $T_2 = T \cup \{n\}$;

c. Define a new function $t_2 : N \rightarrow C$, such that

$$\begin{aligned} t_2(i) &= t(i) && \text{for all } i \in T; \\ t_2(n) &= 0. \end{aligned}$$

d. Choose some $m \in T$ and $h \in X^+$ such that $f(m, h) = 0$;

e. Define a multiplication function $f_2 : T_2 \times X^+ \rightarrow T_2 \cup \{0\}$ by

$$\begin{aligned} f_2(m, h) &= n \\ f_2(n, h^{-1}) &= m && \text{(consistency condition)} \\ f_2(n, g) &= 0 && \text{whenever } g \neq h \\ f_2(i, g) &= f(i, g) && \text{otherwise} \end{aligned}$$

1-3.1.4 Subgroup_Definition

Suppose that i and j are coset-ids which lie in different equivalence classes of T / e , and that there exist $k \in T$ and $g \in X^+$ for which $f(k, g) = i$. We can extend the subgroup H by asserting that $f(k, g) = j$ also. For if $t(i) = Hx$ and $t(j) = Hy$ for some $x, y \in G$, we have asserted that $yx^{-1} \in H$, which previously was untrue.

Conversely, any function f which satisfies the definition given above existentially defines a subgroup H . In the particular situation in which $T = \{1\}$ and $f(1, g) = 0$ for all $g \in X^+$, H is the trivial subgroup $\langle 1 \rangle$. In this situation we have an "empty coset table".

1-3.1.5 Coset_Enumeration

Given a group presentation P , and functions t and f as defined above, we have an existential definition for a subgroup H of the group G presented by P . This information is the starting point for Todd-Coxeter coset enumeration as

implemented in $t\alpha$.

The aim of the process is to find a function f (and hence t and T) such that t is surjective. At all times, the function f represents our current state of knowledge. From it we can construct a partial multiplication table for the action of G on the cosets of H .

The aim is pursued by repeated application of the following steps:

- i. The set T of coset-ids is factored by the equivalence relation e , giving a transversal U . Equivalence classes are obtained by "tracing" the relations of P . That is, the value $j = f'(i, r)$ is found, for all $i \in T$ and $r \in R$. If $j \neq i$, we know that $j \sim i$ because $t(f'(i, r)) = t(i)$.
- ii. Replace the definitions of f and t by their restrictions to U . We now have $T = U$, by definition. Properties 1-4 of U imply no loss of information here.
- iii. If t is surjective, the cardinality of U equals the index of H in G and the multiplication table f denotes the action of X^+ on U . The process terminates.
- iv. If t is not surjective we must try a different f . Define a function f_n , by "defining" $n-1$ new coset-ids. Call this new function f , and use it to define t and T . These now form the input to a new instance of the process.

If the above procedure terminates, the cardinality of T is equal to the index $[G:H]$.

The Todd-Coxeter coset enumeration method is the repetition of these transversal-sufficiency-definition steps, starting with an existentially defined subgroup H of G . The choice of coset-id definitions in step iv may be such that the process does not terminate, even though the index $[G:H]$ is finite.

1-3.2 High-Level_Design

The following design points arise directly from the requirements:

- tc is a "one-pass" process, allowing no restarts, word editing or re-ordering, memory sizing etc.;
- since the user chooses the defining strategy used within tc to supplement the transversal function, we can make no predictions about the method he employs. We can restrict the maximum number of coset-ids to be finite, but we cannot guarantee that the user will specify a coset enumeration method which terminates for subgroups of finite index;
- in previous implementations of the Todd-Coxeter method [15,27,29,46], the implementation of the transversal function is distributed through the other mechanisms of the tool. We must not allow this to happen in tc : if the user is to be in complete control of the definition of new coset-ids he should not have to cope with the possibility that the transversal function might be applied during his calculations;
- we shall make no special effort to allow tc to deal with huge numbers of coset-ids. This choice is made partly for practical reasons (on a single-user personal computer, developing a large enumerator would pose

problems which have little to do with computational group theory), but mostly for philosophical reasons, as outlined in Chapter 0.

1-3.2.1 *General_Architecture*

We can now define the architecture of the *tc* tool.

1-3.2.1.1 *Inputs*

A *P-file*, which contains a presentation $P = \langle X \mid R \rangle$ for the group G ;

A *C-file*, which defines a set T of coset-ids and a multiplication table which defines the action of X^+ on T , as above. This file defines a subgroup H of G existentially.

1-3.2.1.2 *Outputs*

An *exit status*, which may take one of the following values:

SUFFICIENT_COSET_IDS (= 0) the output C-file defines a set T of coset-ids which are sufficient to name all of the cosets of H in G ;

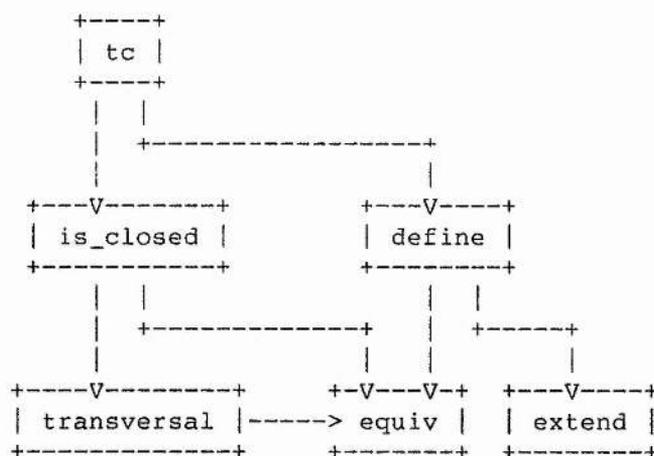
INSUFFICIENT_COSET_IDS (= 1) the output C-file defines a set T of coset-ids which are insufficient to name all of the cosets of H in G (to the best of *tc*'s knowledge);

FATAL_ERROR (= 2) a drastic error has occurred within *tc* or one of its support libraries. Any output C-file is likely to be meaningless.

A *C-file*, which defines a set T of coset-ids and a multiplication table of the action of X^+ on T . If the exit status of tc is SUFFICIENT_COSET_IDS, the coset-ids are in one-to-one correspondence with the cosets of H in G . The correspondence guarantees that $t(1) = H$.

1-3.2.1.3 Modules

The high-level structure of the tc tool can be depicted as follows:



where

tc

acts as the supervisor for the Todd-Coxeter coset enumeration process, creating a supply and demand situation. New cosets are defined only when they are necessary to further the process. For instance, if it happened that the coset-ids defined within the input C-file were sufficient, no new coset-ids need be supplied, because there is no demand;

is_closed

determines whether or not the current set T of coset-ids is sufficient, according to the second sufficiency criterion listed above;

transversal

applies a partial transversal function to the set T , removing those coset-ids which have been shown to lie in equivalence classes for which T already has a representative in U . The above architecture clearly shows that this activity takes place only at the request of *is_closed*, and never during the action of *define*;

define

implements a user-defined definition strategy for extending T . This is a *service*, not a "phase";

extend

adds a new coset-id to the domain T of f , as defined above;

equiv

notes that two coset-ids lie in the same equivalence class of T / e . This module simply acts as a note-book to help *transversal* find equivalence classes more quickly.

1-3.2.2 *Data_Structures*

The tool *tc* uses objects of the following types in order to communicate static information between the modules:

Table

defines the set T of coset-ids and contains information for memory management libraries. See Appendix B for details;

Coset

houses all known information about a particular coset-id i , including the action of X^+ on i under f . See Appendix B for details;

Presn

houses the set of words R , the relators of P which define G . See Appendix B for details;

Control

houses the user's selection of definition strategy. This module communicates between *tc* and *define*.

In addition, a private data structure is used by *equiv()* and *transversal()* to house the list of coset-id equivalence classes.

1-3.2.3 *Coset-id_Definition*

The role of a coset-id defining strategy is to extend the current list of coset-ids, according to demand. Each strategy has several parameters, but each also relies upon other functions which know about the nuts and bolts of coset table management etc. The definitions service within *tc* has the structure:

fies the condition with relator r ;

IR Inverse Relators:

Find a relator $r = g_1 \dots g_n \in R$ and a coset-id $k \in T$ such that $f'(k, r^{-1}) = 0$. Then there exists a greatest i (and $w = g_{i+1} \dots g_n$) such that $f'(k, w^{-1}) \neq 0$ but $f(f'(k, w^{-1}), g_i^{-1}) = 0$. Define a new coset-id m such that $f(f'(k, w^{-1}), g_i^{-1}) = m$. Re-examine the product $f'(k, r^{-1})$ until no further definitions can be made in this way. Continue until no coset-id k satisfies the condition with relator r ;

C Cosets:

Find a coset-id k and a relator $r = g_1 \dots g_n \in R$ such that $f'(k, r) = 0$. Then there exists a least i (and $w = g_1 \dots g_{i-1}$) such that $f'(k, w) \neq 0$ but $f(f'(k, w), g_i) = 0$. Define a new coset-id m such that $f(f'(k, w), g_i) = m$. Re-examine the product $f'(k, r)$ until no further definitions can be made in this way. Continue until no relator $r \in R$ satisfies the condition with k ;

IC Inverse Cosets:

Find a coset-id k and a relator $r = g_1 \dots g_n \in R$ such that $f'(k, r^{-1}) = 0$. Then there exists a greatest i (and $w = g_i \dots g_n$) such that $f'(k, w^{-1}) \neq 0$ but $f(f'(k, w^{-1}), g_i^{-1}) = 0$. Define a new coset-id m such that $f(f'(k, w^{-1}), g_i^{-1}) = m$. Re-examine the product $f'(k, r^{-1})$ until no further definitions can be made in this way. Continue until no relator

$r \in R$ satisfies the condition with k ;

I Images:

Find a coset-id $k \in T$ and group generator $g \in X^+$ such that $f(k, g) = 0$. Define a new coset-id m such that $f(k, g) = m$. Continue until no $g \in X^+$ satisfies the condition with this $k \in T$;

G Generators:

Find a group generator $g \in X^+$ and a coset-id $k \in T$ such that $f(k, g) = 0$. Define a new coset-id m such that $f(k, g) = m$. Continue until no coset-id $k \in T$ satisfies the condition with this $g \in X^+$;

U User input:

Allow the user to specify the values $k \in T$ and $g \in X^+$ for which he wishes the definition $m = f(k, g)$ to be made.

Some obvious remarks about these filling methods:

- ⊕ the filling methods IR and IC can be achieved externally to *tc*, using a version of *canon* which puts each relator into the inverse of its canonical form. Consequently they are not implemented in *tc*;
- ⊕ the order in which relators are examined in strategy C is likewise the purview of *canon*, applied to the input P-file prior to passing it to *tc*;
- ⊕ neither method G nor method I can terminate, as defined above, because each definition $m = f(k, g)$ will permit the subsequent definition $n = f(m, g)$;

- ⊕ the order in which coset-ids are examined in fill method C could be determined by the user, or dynamically determined using heuristics within *tc* (see [35] for some suggestions as to sensible heuristics which might apply here). This has not been tried yet - coset-ids are examined in the order in which they occur in the coset table;
- ⊕ the relators in method R, and the generators and their inverses in method G, could also be ordered interactively or dynamically;
- ⊕ the U method could be implemented as an interactive module, or as an application of the *extend* tool, reading from a P-file. Neither of these has been implemented in the current tool-set's version of *tc*.

At the start of the process the filling method is specified in the enumeration procedure's Control structure (in *tc*, this structure is initialised according to parameters from the command line).

In order to make the filling method work effectively, and to place limits on the amount of time and space used by the definition strategy, several parameters are provided in Control for the user to set:

maxCosets:

the maximum number of Cosets which may be defined at any one time. This is the physical size of the set of Coset structures;

maxDefs:

the maximum number of definitions which can be made during one invocation

of *define()*;

maxCoincs:

the maximum number of coincidences which may be noted during the working of *define()* and its subsidiary functions;

maxPasses:

the maximum number of passes which may occur during one request to *define()*.

1-3.2.4 *Coset-id_Sufficiency*

As we have stated above, the subgroup H is defined existentially via the definitions of T and f in the form of a C-file. The primitive tool-set provides several ways to produce this C-file, but an obvious approach (the technique recommended by Todd and Coxeter in [45] and used in almost every implementation of their method) is to use the modules of the *tc* tool.

If the set R of words in the generators X^+ is in fact a set of generating words for the subgroup H of G , the technique can be expressed as an alternative condition to coset-id sufficiency. The following statements are equivalent:

1. the set T of coset-ids and the multiplication function f existentially define the subgroup H ;
2. coset-id 1 is *closed* with respect to every $r \in R$.

Accordingly, *is_closed* has a parameter which, if set, causes it to check for "subgroup sufficiency" instead of "group sufficiency". This parameter is set by the user, and passed to *is_closed* by the module *tc*. Note that if we wish to enumerate the cosets of the normal closure of H in G , we should use "group sufficiency" when defining the C-file for H .

1-3.2.5 *Coincidence_Processing*

Our chosen approach in the design of software to produce the transversal U of T/e is to build up a data structure which links coset-ids together into equivalence classes. The recognition that two coset-ids are *coincident* is the constructor for this structure, and is embodied in the function *equiv()*. The application of the structure to the set of coset-ids is implemented as *transversal()*, whose output is the set U of non-equivalent coset-ids.

The data structure exists statically, so as to allow *equiv()* to augment it at any time and to permit *transversal()* to apply it at any time as a partial equivalence relation. However its contents, whereabouts and structure are invisible to the other modules discussed above. The data structure is created by *equiv()* when it is invoked for the first time.

1-3.3 *Low-Level_Design*

The module *tc*, which acts as supervisor for the Todd-Coxeter method, pulling the other modules together for the user, is very simple. Here we give two simple-minded implementations, which take no account of fatal error conditions etc, to illustrate the module structure. In an imperative pseudo-language we have:

```

tc(C, P)
{
  WHILE NOT is_closed(C, P) DO
    define(C, P)
  DONE
  RETURN C
}

```

In an applicative language we might have:

```

tc C P = C          when (is_closed C P)
      tc (define C P) P otherwise;

```

(In both cases P represents a presentation for G and C represents a coset table which existentially defines a subgroup of G).

1-3.3.1 *Data Structures*

All objects of the four types listed above are allocated dynamically, using library routines defined in Appendix C. The Control type, which is not defined in Appendix B, consists of the following fields:

Control

Field	Type	Usage
maxCosets	32-bit integer	the maximum number of coset-ids which may be defined at any one time.
maxDefs	32-bit integer	

		the maximum number of defini-
		tions which can be made during
		one invocation of <i>define</i>
maxCoincs	32-bit integer	the maximum number of coin-
		cidences which may be noted
		during the working of <i>define</i>
maxPasses	32-bit integer	the maximum number of passes
		which may occur during one
		request to <i>define</i>
fillers	character string	the sequence of filling
		methods to be used in each
		pass

1-3.3.2 Coset-id_Definition

New cosets are actually defined using the module *extend()*, which returns the coset-id of the newly-allocated Coset structure. This function hides the details of memory management, free-list scanning etc. from the programmer supplying the definition strategy. Thus the function call *extend(i, g)* returns a

new coset-id m , defined such that $f(i, g) = m$ and $f(m, g^{-1}) = i$. Both of these facts are recorded, in the *image* fields of the Coset structures for the coset-ids i and m respectively. Counters for each of the limiting quantities found in the Control structure are zeroed by `define()` and incremented by `extend()` or the functions which house the definition strategies. If any counter reaches the user-specified maximum indicated in Control, `define()` terminates.

The filling methods themselves may have side-effects, in that they may discover facts which might be of use to the modules *is_closed* and/or *transversal*, as follows:

- a. methods R or C may discover that a coset-id is closed with respect to all of the relators $r \in R$. This fact is noted in the *flags* field of the coset-id's Coset data structure;
- b. method I may discover that a coset-id is I-closed. This fact is noted in the *flags* field of the coset-id's Coset data structure;
- c. methods R or C may discover that two coset-ids are coincident. This is always the result of discovering a relator $r \in R$ and a coset-id $i \in T$ such that $f'(i, r) = j$ and $f'(i, r) = k$, with $j \in T$ and $k \in T$, but $j \neq k$. This fact is noted by calling the module *equiv()* to record that j and k are coincident.

The request for more cosets succeeds (ie. `define()` returns "true") if any coset was defined or any side-effect was noted. The caller's request has been satisfied, at least in part.

1-3.3.3 Coset-id_Sufficiency

The approach taken by the module `is_closed()` is quite straight-forward. We use the second sufficiency condition (Section 1-3.1.2), in that every coset-id $i \in T$ is checked for R -closure with respect to every relator $r \in R$. Coset-ids are checked in "ascending order", using the natural ordering " $<$ " on N .

The process of checking a coset-id for closure can have two side-effects:

1. if the coset-id does turn out to be closed in this way, a flag CLOSED is set on the coset-id's associated Coset structure. A pointer is maintained to the first coset-id which doesn't close when applied to some relator. This is the starting-point for the next call to `is_closed()`;
2. a pair of coset-ids may be found to be coincident. This fact is noted via `equiv()`.

If every coset-id closes at every relator, and no coincidences have been noted, `is_closed()` returns "true", otherwise "false". The outer levels of the calling process (usually `tc` or `tcs`) can then decide how to use this information.

Within `tc`, `is_closed()` is the only function which calls `transversal()`. `transversal()` is called immediately on entry to `is_closed()`, to mop up any coincidences which may have been found by any other part of the `tc` process, and then after each coset-id has been applied to all relators.

It can be argued here that coincidences found while checking a coset-id should be applied to the coset-ids immediately, instead of waiting until this coset-

id has been applied to every relator. However, by strictly controlling the times of potential collapse we obtain some benefits:

- ⊕ the implementation of coset-id checking is now easy. Very little book-keeping is required because the set T of coset-ids cannot shift under our feet;
- ⊕ the property of coset-id closure is more clearly defined for the user. We allow the coset-id to demonstrate its closure or non-closure, and only then do we examine the side effects of the process, namely that some coset-ids may be redundant;
- ⊕ the overall organisation of the Todd-Coxeter algorithm is clarified. Should we wish, at some later date, to alter the time at which the transversal is applied to T , the advantages and disadvantages of any approach can be easily measured.

This approach appears to have only one disadvantage, which is that closure may be demonstrated for some coset-ids which turn out to be redundant. Consequently, some unnecessary checking has taken place. When there are particularly long relators this may be inefficient. The problem can be alleviated somewhat by recognising the COINC flag within the scanning portion of *is_closed()*, so that coset-ids which are already known to be redundant are skipped. In practice, however, it appears that *transversal()* itself usually takes longer than the extra checking which would have occurred, so the problem is not a serious one.

1-3.3.4 *Coincidence_Processing*

When *equiv()* is called upon to note that some pair of coset-ids are equivalent, it adds this information to the static data structure which represents the equivalence relation. One or both of the coset-ids will be chosen as *redundant*, in the sense that they will not represent their equivalence class in the transversal *U*. The redundant coset-id(s) have a COINC flag set in their corresponding Coset structures, so that other modules within *tc* may choose to ignore these coset-ids.

The traditional data structure for the equivalence relation is a list of pairs of coset-ids, the *coincidence queue*. Furthermore, this list, as was suggested in [4] and implemented in TC, is often stored by corrupting the values in the *image* fields of Coset structures. Not only does this lead to cryptic overloading of data objects; in a model such as that above, in which the list can exist during and between all aspects of processing, the corruption might cause the definition of *f* to become incorrect.

The list organisation described by Leech [27,29] has been found to be convenient. Each coincident pair (a, b) of coset-ids is ordered so that $a > b$ according to the natural ordering on *N*. One organisation for the list is to order it, with $(a, b) > (c, d)$ iff $a > c$ or $a = c$ and $b > d$. (there is never any requirement for any pair to occur more than once in the list). Now it is easy to discover whether a coincidence is already known, by scanning the list. However, this is such a slow process that it is much quicker to keep the list unordered, and instead to alter *transversal()* so as to ignore cases of undefined coset-ids being referred to after their removal from *T*.

To counter the above difficulties, and to model the τ rationale more closely, we here adopt a different approach. The data structure consists of a list of equivalence classes, each of which is represented physically by a data structure containing the list of coset-ids which are currently known to fall into that class. The class has a nominated representative in U , and every coset-id in the class has its Coset structure point to the Class object.

This organisation counters the inefficiency of the list approach. Consider a request to *equiv()* to note that coset-ids a and b are now equivalent:

- ⊕ since each Coset points to a Class only if its coset-id is a member of that equivalence class, we have instant knowledge as to whether a coset-id is worth looking for in the structure;
- ⊕ if neither a nor b is currently a member of a Class, we create a new Class and elect either a or b as its representative;
- ⊕ if a is known to fall within a particular equivalence class, but b is hitherto "unclassified", we simply append b to the class containing a ;
- ⊕ if both a and b are already known to fall into the same class (a test which is a simple comparison of Class pointers), we can dismiss this coincidence;
- ⊕ if a and b are currently thought to lie in different equivalence classes, these classes must be merged. This can be achieved by chaining together their lists of coset-ids, or by having one Class refer to the other as being the holder of all its data. Neither operation requires any searching or list scanning.

1-4. *The_Reidemeister-Schreier_Process*

This Section describes the design of a couple of primitive tools which can be used together or separately to carry out a variety of tasks. Primarily, the tools are designed to be used to perform the Reidemeister-Schreier method for presenting a subgroup of a group given by its presentation.

1-4.1 *Rationale*

The Reidemeister-Schreier process for presenting a subgroup H of a group G with finite presentation $P = \langle X \mid R \rangle$ can be structured in terms of three processes executing in sequence:

- a. construct a coset table for H in G ;
- b. construct a Schreier transversal for the cosets of H , using the coset table above;
- c. rewrite a set of relators for G into a set of relators for H , using the Reidemeister method, to produce a finite presentation for H , on a set of Schreier generators determined by the Schreier transversal above.

In the primitive tool-set we choose to make this division concrete, by providing individual tools (*tc*, *st* and *rewrite*) for each step and allowing the user to plug them together to construct particular instances of the Reidemeister-Schreier method. The benefits of this modular approach are:

- many different Schreier transversals are constructible from any particular coset table. By isolating this step into a tool we can provide the user with choice, which may improve the chances of success of the whole

process;

- ⊕ the relators which are rewritten using the Schreier transversal need not be the same relators as were used in the Todd-Coxeter method to create the subgroup's coset table. In fact, they need not even present the same group. In this case we can obtain interesting combinations of tools which present subgroups which have no convenient set of generators, for instance;
- ⊕ the coset table may not have arisen from the Todd-Coxeter process. If the primitive tool-set can be expanded to provide several mechanisms for the construction of coset tables, our Reidemeister-Schreier method should not suddenly find itself pushed to one side because of its incompatibility with newer tools;
- ⊕ Schreier transversals, which are equivalent to coset table spanning trees, may be useful in their own right. For instance, it may be possible to "seed" a coset enumeration with a "similar" subgroup's spanning tree, possibly as a way of starting the Todd-Coxeter process off in the right direction.

Many of these ideas will be explored in Chapter 2. The remainder of this Section describes the primitive tool *st*, in which much of the user's choice is expressed.

1-4.2 Design

In order for Schreier transversals to be useful in (at least) the situations outlined above we restrict the scope of the tool *st* so that it constructs a transversal and then stops. We leave it to the tools of Section 1-6 to pick up this data and use it to seed coset tables, define Schreier generators etc. This, in turn, implies a type of data interchange file format to mediate information between the tools. A P-file usage is defined below for this purpose.

1-4.2.1 Architecture

In architecture then, *st* is a filter, reading in a C-file and writing out a P-file, constructed according to instructions received on the command-line. A P-file is also used as input, to help guide the algorithms (see below) and to provide a namelist for the output P-file. The version of *st* considered here will be fairly simple, although more elaborate tools could easily be constructed along similar lines.

The ultimate objective of *st* is therefore to build a Presn structure, which will then be output as a P-file. The contents of the P-file will be a set of maximal words in a Schreier transversal for the cosets of H , from which the complete transversal can be "read off" using the Schreier property that every left sub-word must also be a member of the transversal. The Presn structure itself will in turn be "read off" a representation of a minimal spanning tree for the coset table given. The algorithms described below therefore construct coset table spanning trees, using the input Table (read from a C-file) as a Schreier coset graph.

1-4.2.2 *Constructing_a_Spanning_Tree*

The starting-point will always be the coset with coset-id 1. This is assumed always to represent the subgroup H itself. Thereafter *st* presents the user with two graph-traversing methods, both of which are defined recursively. Each constructs a Tree structure, as defined in Appendix B-4. The root of the Tree represents the start of every word in the tree, and is labelled with the identity (0).

Let i be a coset-id for which a node already exists in the Tree, and suppose that the method is now visiting this node with a view to specifying the definition of further coset-ids. First, the flag DEFINED is set on the coset-id's data structure, so that no further such visits may occur. What happens next depends on the choice of method:

D *Depth-First Traversal*

For each generator or inverse $g \in X^+$, if the coset-id $j = f(i, g)$ is DEFINED, continue. However, if j has not been visited yet, visit j now. When every $g \in X^+$ has been considered, the visit to i is complete.

B *Breadth-First Traversal*

For each generator or inverse $g \in X^+$, if the coset-id $j = f(i, g)$ is DEFINED or MARKED, continue. Otherwise set the MARKED flag against coset-id j . Now, for every generator or inverse $g \in X^+$ such that $j = f(i, g)$ is MARKED but not DEFINED, visit j . When every $g \in X^+$ has been considered, unset coset-id i 's MARKED flag. The visit to i is now complete.

Some remarks:

- in both methods, the DEFINED flag on i also prevents the use of fixed points (generators for which $f(i, g) = i$) in the tree;
- in either method, the order in which the generators are tried at each node visit is critical. The user of *st* can choose to visit them in the order defined by the coset table, or in the order defined by the Namelist of the controlling P-file. See Section 1-1.3.1 (and *canon(1)*) for orderings which can be so applied.

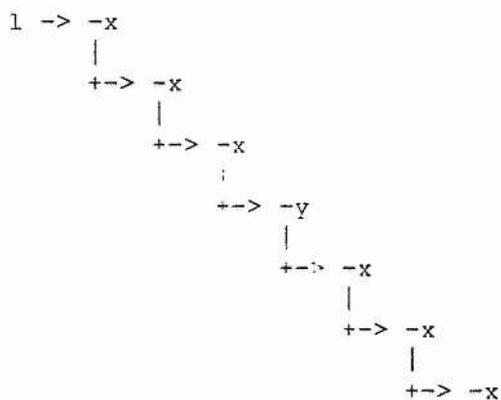
1-4.2.3 *An Example*

As an example, consider a coset table for the trivial subgroup of the quaternion group $Q = \langle x, y \mid xyx = y, yxy = x \rangle$:

	-x	x	-y	y
1:	2	8	4	7
2:	6	1	3	5
3:	4	7	8	2
4:	5	3	6	1
5:	7	4	2	8
6:	8	2	7	4
7:	3	5	1	6
8:	1	6	5	3

(In what follows we use the ordering $-x, x, -y, y$ on the group generators.)

Method D produces the tree

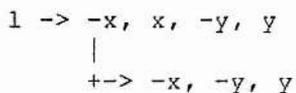


The corresponding P-file contains

```

# Schreier transversal
<x, y | x-3y-1x-3>
    
```

Conversely, method B produces the tree



which has the following associated P-file:

```

# Schreier transversal
<x, y | x-2, x-1y-1, x-1y, x, y-1, y>
    
```

The manual page st(1) describes the current instance of this tool from the user's point of view.

CHAPTER 2

USING THE TOOLS

The flexibility of the tool-set designed in Chapter 1 is tested. Compositions of the tools which solve a variety of problems are demonstrated. Interactive environments which interface them to the user are developed. The major tools *tc* and *tt* are shown to be powerful precisely because of their modular structures.

2-0. Introduction

The strength of the designs presented in Chapter 1 rests on flexibility. The major tools, *tc* and *tt*, each provide many approaches to any problem, and are structured so as to be easily extensible in certain general directions. Furthermore, the tool-kit as a whole has been given a fine-grained structure, with the aim that the user is not restricted to solving problems of the types anticipated by the tool-builder.

In this Chapter, both of these kinds of flexibility are tested. In addition interactive, goal-directed and heuristic environments are presented. Behind their interfaces, the tools are used to tackle problems in "standard" ways, either under user control or autonomously, using knowledge about the tools and their problem domain.

Specific tool output, where included in this Chapter, is obtained from the particular implementation of the tool-set which is described in Appendices B and C. These tools were implemented in 'C' and run in the Unix System III operating system. Shell scripts included here are written for the Unix Bourne shell *sh(1)*.

2-1. *Tests_of_Individual_Tools*

2-1.1 *Introduction*

The purpose of this Section is to demonstrate that the individual tools designed in Chapter 1 do their intended jobs. For the most part this involves an exploration of the flexibility of *tc* and *tt*, and demonstrations that this flexibility allows the tool user to solve more problems.

Each of the tool-kit tools has been tested on several hundreds of examples, very often in conjunction with other members of the tool-kit. For instance:

- ◊ *tc* has been tested on every example from [15], and on many (usually small) examples with full tracing enabled. The results were found to be correct in every case. A selection of these tests, on three "pathological" examples from [15], is included in Section 2-1.2.
- ◊ *tt* was tested extensively while developing the command meta-schemas described in Section 2-1.4.
- ◊ several prototype designs for *st* were tested on hundreds of examples before arriving at the structure described in Chapter 1. These examples were in turn generated using *tc*, *rewrite* and *tt*.
- ◊ many of the heuristics implemented in the prototype *gp* tool make use of the smaller tools in the set, such as *def* and *dindex*, which have therefore been tested in that context.

After the tool designs had been shown to work, attention was focussed on the subgroups $H(n)$ of the groups $Y(n)$, defined in [11,12,21]. Various

combinations of methods in *tc*, *st* and *tt* were essayed on these examples before the presentations given in Sections 2-1.4 and 2-2.2 were produced. The results of many of these tests can be found in this Chapter, and many of the compositions used are described in Section 2-2.6.

2-1.2 *Definition_Strategies_in_Coset_Enumeration*

2-1.2.1 *Introduction*

The principal benefits of the particular implementation of *tc* chosen in Chapter 1 can be seen to be:

- the software architecture. This isolates memory management, coset table primitives and control structures into black boxes, clarifying the role of the tool and the actions it performs;
- the introduction of selectable Definition Strategies, with clearly-defined parameters and combining rules.

This Section concentrates on the latter, Definition Strategies. The results given here are by no means exhaustive. They are designed to give a flavour of what is possible with *tc*. As a result, many interesting questions remain unexplored, particularly questions of the relationship of *tc* Definition Strategies with other primitive tools.

We concentrate on three of the "pathological" examples taken from [15]: E_1 of order 1, defined by the (canonicalised) P-file

$$\langle a, b, c \mid a^2c^{-1}a^{-1}c, ab^2a^{-1}b^{-1}, bc^2b^{-1}c^{-1} \rangle$$

(2, 5, 7; 2) of order 1, defined by

$$\langle a, b \mid a^2, b^5, [a^{-1}, b^{-1}]^2, (ab)^7 \rangle$$

and $PSL_2(11)$ of order 660, defined by

$$\langle a, b \mid a^2, (ab)^3, b^{11}, (ab^4ab^{-5})^2 \rangle$$

Each will be enumerated over the trivial subgroup. The presentation of each has been canonicalised (by 'canon -ir -rl').

2-1.2.2 *Definitions_Restriction*

The `-d` flag to `tc` allows the user to restrict a Definition strategy so that it will terminate after defining a certain number of new coset-ids. Varying the `-d` flag, while leaving the other `tc` parameters set to their default values, we have:

Example E_1 :

<code>-d</code>	Maximum Coset-ids	Total Definitions
1000	1001	1000
200	600	600
100	599	600
49	550	588
20	539	560
10	539	550
1	538	541

Example $(2,5,7;2)$:

<code>-d</code>	Maximum Coset-ids	Total Definitions
1000	1001	1000
200	318	400
100	260	300
50	219	250
25	227	250
20	242	260

Example $PSL_2(11)$:

-d	Maximum Coset-ids	Total Definitions
1000	1644	2006
200	900	1216
100	753	1010
50	695	901
40	686	889

In general, lowering `-d` increases the chances of an enumeration succeeding within a particular memory size, but the effect is by no means monotonic.

2-1.2.3 *Memory_Restriction*

The `-m` flag to `tc` allows the user to restrict the maximum number of coset-ids which may be known at any one time. The default value is 10000. Successively restricting this value while leaving the other `tc` parameters set to their default values, we have:

Example E_1 :

-m	Maximum Coset-ids	Total Definitions
10000	1001	1000
1000	1000	999
700	700	699
550	550	595
538	538	578

Example $(2,5,7;2)$:

-m	Maximum Coset-ids	Total Definitions
10000	1001	1000
1000	1000	999
500	500	777
250	250	383
200	200	301

Example $PSL_2(11)$:

-m	Maximum Coset-ids	Total Definitions
10000	1644	2006
1000	1000	1718
680	680	892
670	670	1031
665	-	-
660	660	1063

(Note that *tc* failed to find the order of this group for the case $-m665$).

In general, restricting memory space also reduces the total number of coset-ids which are defined, which in turn reduces the time taken by the process. These effects are not monotonic.

2-1.2.4 *Pass_Components*

Repeating the previous exercise, with each pass consisting of the strategies "CI" (instead of "C", as above), we have:

Example E_1 :

-m	Maximum Coset-ids	Total Definitions
10000	1001	1000
1000	1000	999
700	700	752
690	690	740
683	683	733

Example (2,5,7;2):

-m	Maximum Coset-ids	Total Definitions
10000	1001	1000
1000	1000	999
500	500	779
250	250	389
200	-	-
196	196	307

Note that *tc* fails to find the order of this group with "-m200 -fCI", but succeeds if the memory space is further reduced to 196 coset-ids!

Example $PSL_2(11)$:

-m	Maximum Coset-ids	Total Definitions
10000	1655	3000
1000	1000	2119
800	800	1533
700	700	1282
680	680	1233
670	-	-

Passes involving "CI" strategies usually define more redundant coset-ids than "C" passes, but this can mean that more coincidences are found sooner. For comparison, here is the trace output from two of these examples:

```
$ cat .trc
stats = tc_stats close_stats coinc_stats defn_stats;
info = tc_info;
```

```
info : 3;
stats : 6;
$ TRACE=6
```

```
$ tc -tC -m200 -fC -P 2572.p
start coinc left npass Cdefs Gdefs Idefs Rdefs ndefs qpush
  1      0      1      19    199      0      0      0    199      9
 200    83    117    11     83      0      0      0     83      0
 200    19    181     5     19      0      0      0     19      0
 200   199     1     1      0      0      0      0      0      0
  1      0      1
```

```
Total cosets defined: 301
Index = 1
```

```
$ tc -tC -m196 -fCI -P 2572.p
start coinc left npass Cdefs Gdefs Idefs Rdefs ndefs qpush
  1      0      1     17    167      0     28      0    195     15
 196    84    112     9     73      0     11      0     84      1
 196    28    168     5     24      0      4      0     28      0
 196   195     1     1      0      0      0      0      0      0
  1      0      1
```

```
Total cosets defined: 307
Index = 1
```

2-1.2.5 Equivalence_Restriction

The `-c` flag to `tc` allows the user to restrict a Definition Strategy so that it terminates when a certain number of "coincidences" (coset-id equivalences) have been noted. `tc` can thus be forced to check coset-ids sooner than is normal (the default value for this flag is "`-c 1000`"). Varying the `-c` flag, while leaving the other definition parameters set to their default values, produces:

Example E_1 :

No coincidences are noted during the default definition strategy with

pass type "C", so the `-c` flag has no effect on this enumeration. The effect of this flag are instead demonstrated using pass type "IC":

<code>-c</code>	Maximum Coset-ids	Total Definitions
1000	1001	1000
16	1855	1997
5	1409	1433
4	1182	1291
3	1354	1372
2	938	949
1	1141	1151

In this case, restricting the number of noted equivalences lowers the chances of an enumeration succeeding. Note also that the number of equivalences noted during the first request for coset-id definitions is 16, but that restricting the definition strategy to 16 coset-ids means that several redundant coset-ids are not now defined. These particular coset-ids clearly have a large effect on the Todd-Coxeter process in this case.

Example $PSL_2(11)$:

<code>-c</code>	Maximum Coset-ids	Total Definitions
1000	1644	2006
10	1516	1880
5	1025	1406
4	1381	1457
3	927	1163
2	1140	1280
1	734	954

Example $(2,5,7;2)$:

-c	Maximum Coset-ids	Total Definitions
1000	1001	1000
10	533	615
5	333	378
4	300	319
3	358	385
2	329	360
1	340	361

In general, lowering `-c` increases the chances of an enumeration succeeding within a particular memory size, but the effect is by no means monotonic.

Here is trace output from three of these enumerations:

```
$ cat .trc
stats = tc_stats close_stats coinc_stats defn_stats;
info = tc_info;

info : 3;
stats : 6;
$ TRACE=6

$ tc -tC -c1000 -P psl2(11).p
start coinc left npass Cdefs Gdefs Idefs Rdefs ndefs qpush
  1      0     1     76  1000     0     0     0  1000     9
1001   357   644    99  1000     0     0     0  1000    15
1644   990   654     6     6     0     0     0     6     0
  660     0   660
```

```
Total cosets defined: 2006
Index = 660
```

```
$ tc -tC -c5 -P psl2(11).p
start coinc left npass Cdefs Gdefs Idefs Rdefs ndefs qpush
  1      0     1     42   560     0     0     0   560     5
 561   114   447    51   578     0     0     0   578     5
1025   294   731    21   227     0     0     0   227     5
 958   311   647    11    40     0     0     0    40     5
 687    28   659     3     1     0     0     0     1     0
 660     0   660
```

```
Total cosets defined: 1406
Index = 660
```

```

$ tc -tC -c1 -P psl2(11).p
start coinc left npass Cdefs Gdefs Idefs Rdefs ndefs qpush
  1      0     1     10    151     0     0     0     151     1
 152    15    137    26    279     0     0     0     279     1
 416    35    381    35    353     0     0     0     353     1
 734    93    641     2     12     0     0     0     12     1
 653    64    589     6     43     0     0     0     43     1
 632    34    598    11     52     0     0     0     52     1
 650    29    621    10     35     0     0     0     35     1
 656    19    637    11     20     0     0     0     20     1
 657     6    651     8     9      0     0     0     9      0
 660     0    660

```

```

Total cosets defined: 954
Index = 660

```

2-1.2.6 Conclusions

Each of the user parameters to the *tc* definition strategies can be used to reduce the amount of memory space required for any particular enumeration. Although these effects do not occur monotonically, or predictably, it may be expected that the flexibility built into *tc* has increased its usefulness, especially within the tool-kit as a whole. For comparison, here we list the "best" results obtained above, alongside those obtained in [15] for the same enumerations. (Note that the statistics given in [15] for the look-ahead method were the "best" results obtained with various parameter settings).

Example E_1 :

Algorithm	Maximum Coset-ids	Total Definitions
Felsch	588	588
Lookahead	695	758
HLT	1649	1705
<i>tc</i> -d10	539	550
<i>tc</i> -m538	538	578
<i>tc</i> -m683 -fCI	683	733
<i>tc</i> -c2 -fIC	938	949

Example $(2,5,7;2)$:

Algorithm	Maximum Coset-ids	Total Definitions
Felsch	254	257
Lookahead	224	227
HLT	344	362
tc -d50	219	250
tc -m200	200	301
tc -m196 -fCI	196	307
tc -c4	300	319

Example $PSL_2(11)$:

Algorithm	Maximum Coset-ids	Total Definitions
Felsch	1066	1118
Lookahead	661	824
HLT	1188	1495
tc -d40	686	889
tc -m660	660	1063
tc -m680 -fCI	680	1233
tc -c1	734	954

The total number of coset-id definitions required in each case is a measure of the "time taken" by the method used. In many cases, therefore, *tc*'s definition strategies are "slower" than the best look-ahead application of TC in [15].

2-1.3 *General_Methods_in_'tt'*

This Section explores the constructors of *tt*'s command language. Meta-level commands are developed which solve a variety of problems in different ways.

2-1.3.1 *Types_of_Meta-Command*

The primitive tool *tt* has been designed (after the model of TTRANS [21]) as an interactive editor for group presentations. It provides basic facilities which act on a single presentation (in this respect, *addrel* is a somewhat dubious inclusion), together with mechanisms which allow the user to construct meta-commands.

In TTRANS one meta-command, AU, is supplied. If the user's goal is not the same as that of AU, only laborious hands-on interaction can achieve that goal. This situation has been alleviated in *tt* by the introduction of three types of meta-level feature:

- ◊ constructors for meta-commands, namely repeat counts, macros, strings, semi-colons, AND-blocks and OR-blocks;
- ◊ run-time variables which allow the user to configure the actions of system primitives (such as effective length function, global trace level etc.);
- ◊ run-time variables which allow the system to communicate transient data to the user (such as current number of group generators).

These features permit the construction of meta-commands with a wide range of goals:

- ◊ report-writers, which output the current state of the edit in styles appropriate to the user's taste or the kinds of interaction required;
- ◊ pattern-matchers, which might report common substrings or redundant generators;
- ◊ reduction commands, which attempt to remove as much redundancy from the presentation as possible. They might be designed to minimise the number of generators, the number of relators or the total length of the presentation;
- ◊ beautifiers, reduction commands which might prefer redundancy if the result contains high indices or a proliferation of conjugates;
- ◊ word derivations, which could provide input to a proof-writing process.

Meta-commands of some of these types will be examined in this Section.

2-1.3.2 *Report_Writers*

The following meta-command might be used within a reduction command, to give a running update on the "size" of the current presentation as it gradually falls:

```
define size "print \"$$ totlen=$ (effective $)\n\"
           $ngens $nrels $totlen $etotlen"
```

This could be replaced by the following version, which writes a more readable (ie. verbose) report. The screen is cleared each time (the macro CLEAR has to be locally defined), so that the user sees the effect of a form being continu-

ously updated:

```
define s1 "print \"Generators: $\n\" $ngens"
define s2 "print \"Relators:  $\n\n\" $nrels"
define s3 "print \"Total free length: $\n\n\" $totlen"
define s4 "print \"Current length function: $\n\" $lenalg"
define s5 "print \"Total effective length:  $\n\" $setotlen"
define line "print \"-----\n\""
define size "CLEAR; line; s1; s2; s3; s4; s5; line"
```

Other report-writers merge more closely with the task at hand. For instance, the meta-command `SL` performs substring replacements, using longer relators first:

```
define nl "print \"\n\""
define tot "print \"$ \" $totlen"
define SL "0[su <121; canon; tot]; nl"
```

Here, the total length of the presentation is printed each time it is reduced by the `su` command. When the length can be reduced no further, a `<NEWLINE>` is printed.

2-1.3.3 *General-Purpose_Reduction_Commands*

Experiments with `tt` seem to show that most useful reduction commands have the general structure

```
init; loop; tidy
```

with the components having the following purposes:

init canonicalises the presentation before work begins, and removes any obvious redundancy (such as generators which are either trivial or equal to another generator);

loop systematically eliminates redundant generators and shortens relators, usually in alternation;

tidy squeezes any last drops of redundancy out of the presentation, usually by random or equal-length string substitutions.

Among the more useful of the commands of this general form, two constructions for the *loop* component seem quite sensible:

a. `init; n [0 [eg; ss]; rc]; tidy`

b. `init; n { 0 [eg; ss]; rc }; tidy`

(for some small value of *n*) where the purposes of these *loop* components are:

eg attempts to use the *c*, *g* or *G* commands to eliminate one or more redundant generators from the presentation;

ss uses the *su* command in some form, attempting to reduce the effective length of the presentation;

rc makes an attempt to disturb the presentation in some way (without making it significantly larger). This command is only called after either *eg* or *ss* has failed.

2-1.3.3.1 *Remarks:*

- ⊕ the basic idea behind both of these "algorithm schemas" is to eliminate redundant generators from the presentation. Between eliminations, some redundancy in the relators can be removed also. If either of these activities fails, the presentation is tweaked (by *rc*) before they are tried again;
- ⊕ *rc* must be programmed so that it succeeds if it has touched the presentation in any way. This is usually achieved with commands enclosed in an OR-block;
- ⊕ if the inner loop fails immediately (either *eg* or *ss* fails at the first attempt) schema (a) above will move to the *tidy* phase, regardless of the attempts of *rc*. Thus schema (b) is likely to be more robust;
- ⊕ *rc* usually embodies a little randomness. It may happen that the random numbers conspire to cause *rc* to fail, even though there was something useful to be done. In this case again, schema (b) is more likely to be able to try once more;
- ⊕ in many cases, *rc* is programmed to consist mainly of equal-length string substitutions, usually repeated several times. These can cancel themselves out, even within a single call to *rc*, so that *rc* succeeds without altering the presentation. This is a waste of time, but otherwise isn't serious.

The next Section presents examples of the use of these meta-command schemas, which seem to bear out the above remarks.

2-1.4 Tests_of_Transformation_Meta-Command_Schemas

2-1.4.1 Introduction

This Section contains the results of a few tests of *tt* meta-commands on well-known examples. In particular, the two types of schema introduced in the previous Section are compared.

In the tests, the examples used will be the following:

Subgroup	Index	References	Prognosis
H1	26	[16,20,21]	intermediate
H2	64	[16,20,21]	intermediate
H3	12	[16,20,21]	intermediate
H4	12	[16,20,21]	intermediate
H5	42	[16,20,21]	very easy
H6	78	[16,20,21]	easy
H(8)	19	[11,12,21]	very difficult
H(-8)	13	[11,12,21]	difficult
G	18	[2,9]	intermediate

2-1.4.2 Easy_Examples

Consider the examples $H_1 - H_6$, used in [21] to test TTRANS. These were input to *tt*, using essentially the same Reidemeister-Schreier presentation for each subgroup as was input to TTRANS. The following (very primitive) series of *tt* commands was then applied:

```
define SU "0[su <12c; canon]"

canon
G 2
SU
0{4c dabc?; SU}
60{c dabc?; su =12o?; su <12oc}
```

The results were:

	ngens	nrels	elen
H1	2	4	14
H2	2	4	6
H3	2	7	50
H4	2	6	36
H5	2	3	4
H6	2	3	4

(where *elen* gives the total length of the presentation's relators, ignoring any exponent on each relator).

These results compare favourably even with the hand-derived results of [21] (note, for instance, that 2-generator presentations were always found by *tt*). Comparison of the actual output presentations (in the cases quoted in [21]) is revealing:

Output from *TTRANS*:

$$H_1 = \langle a, b \mid a^4, (ab^{-1})^3, abab^{-2}a^{-1}b^{-2}, (ab^{-2})^3, (a^2b^{-3})^3, [a^{-1}, b^{-1}]^4, (ab^2)^6 \rangle$$

$$H_2 = \langle a, b \mid a^4, b^4, (ab)^2, (ab^{-1})^4, (a^2b^2)^4 \rangle$$

Output from *tt*:

$$H_1 = \langle a, b \mid b^3, a^4, (ab^{-1})^3, [bab, a^2] \rangle$$

$$H_2 = \langle a, b \mid a^4, b^4, (ab)^2, (ab^{-1})^4 \rangle$$

In each case, the order of both generators is found by *tt*. These are quite easy examples, but the results are not atypical. See Section 2-2.2 for results of *tt* application to the other examples in [21], H(8) and H(-8).

2-1.4.3 *Tests_of_Meta-Command_Schemas*

We now test variants of the meta-command schemas (a) and (b) introduced in the previous Section. For this we use examples H_5 , H_3 , G and H(8) (in increasing order of difficulty). The meta-commands to be tested are a1 - a7, where

```
define S1 "0 [ su <12c; canon ]"
define S2 "0 [ su <12; canon ]"

define Q1 "5 { su =12oc?; canon; S1 }"
define Q2 "5 [ su =12oc?; canon; S1 ]"
define Q3 "5 { su =loc; canon; S1; su =2oc; canon; S1; su =12oc?; canon; S1 }"

define egl "[ c $1; c $1; c $1; c $1 ]"

define init1 "canon; 0 [ G 2; S1 ]"
define init2 "canon; G 2; 0 [ su <12lc; canon ]"

define tidy1 "S1; Q1"
define tidy2 "5 Q2"

define a1 "init1; 3 [ 0 [ c dabc?; S1 ] ]; tidy1"
define a2 "init1; 3 { 0 [ c dabc?; S1 ]; Q1 }; tidy1"
define a3 "init1; 3 { 0 [ 4c dabc?; S1 ]; Q1 }; tidy1"
define a4 "init2; 5 [ 0 [ egl 1 ]; Q2 ]; tidy2"
define a5 "init2; 5 { 0 [ egl 1; S2 ]; Q2 }; tidy2"
define a6 "init2; 5 { 0 [ egl d?; S1 ]; Q3 }; tidy2"
define a7 "init2; 5 { 0 [ egl dl; S1 ]; Q3 }; tidy2"
```

Each of these "algorithms" was run 10 times on each test group, in order to "average out" the effects of randomness. In the tables of results, the columns have the following meanings:

- i. the average number of generators left after the 10 trials;
- ii. the total free length of the shortest presentation found which had the minimum number of generators. This is, in some sense, the "best" presentation found for this example;

- iii. the number of trials which obtained this "best" result. This figure provides a measure of the "reliability" of the method;
- iv. the average free length of all the presentations produced in the 10 trials.

The results were:

Example H_5 :

	i	ii	iii	iv
a1	11.0	87	1	90
a2	8.1	65	1	74
a3	2.1	19	9	19
a4	2.1	19	2	55
a5	3.0	116	10	116
a6	2.4	19	5	21
a7	2.0	19	10	19

Example H_3 :

	i	ii	iii	iv
a1	17.5	149	2	162
a2	7.1	64	1	135
a3	2.0	85	1	162
a4	2.0	53	1	81
a5	2.0	121	1	166
a6	2.0	67	1	104
a7	2.0	247	5	250

Example G:

	i	ii	iii	iv
a1	17.0	145	1	148
a2	7.4	78	1	180
a3	3.0	82	3	198
a4	3.0	312	1	593
a5	3.0	328	1	390
a6	3.0	60	1	173
a7	3.0	66	8	87

This group requires at least three generators, and most of the meta-

commands have achieved this.

Example H(8):

	i	ii	iii	iv
a1	7.6	29	1	1064
a2	4.2	2663	1	1644
a3	2.7	45	1	1507
a4	2.0	117	1	131
a5	2.0	131	10	131
a6	3.7	2457	1	781
a7	2.0	1936	10	1936

Here *a1* finds a remarkable presentation for $H(8)$ (quoted elsewhere).

2-1.4.4 Conclusions

- ⊕ *a1* performs very poorly in every case, giving up before eliminating all of the redundant generators. This was predicted (Section 2-1.3.3) for a meta-command of schema type (a);
 - ⊕ the differences between the results of *a2* and *a3* (particularly in cases H_5 and $H(8)$) are surprising. In every example, more generators are eliminated if they are attempted in groups of 4. The most plausible explanation is that the elimination of four generators produces a large number of common substrings in the remaining relators, which are then very suitable for string matching and replacement;
 - ⊕ the results of *a4* and *a5* show two effects. Firstly *S2*, which gives up after performing one transformation with each monitor relator, performs worse than *Q2*, which is more robust.
- Secondly, since *a4* has no *ss* component, generators may be eliminated in fairly large batches. In fact, they will be eliminated all at once if

possible. This reinforces the above deduction regarding a_2 and a_3 , and suggests that substring replacement is best used only as a last resort;

◊ a_7 performs much more consistently than a_6 , but a_6 found a better result in almost every case ($H(8)$ being the exception). The differences must be a result of the choice of which generators to eliminate. Randomness in the eg component provides better results, but only in the long term;

◊ the results of a_3 and a_6 are fairly closely matched, showing that the exact form of the *tidy* component is not important.

Further experimentation is required, especially with more "difficult" examples. However, these results seem to show that tt 's flexibility (especially the inclusion of randomness) produces good results.

2-2. *Tests_of_the_Tool-Kit*

2-2.1 *Introduction*

This Section describes tests which have been carried out on compositions of the tools designed in Chapter 1. In the main, the granularity chosen in Chapter 1, especially in the design of the Reidemeister-Schreier method, was found to greatly improve the user's ability to present any subgroup "nicely".

In the sub-sections below which deal with compositions of tools, the approach taken is to describe the resulting application independently of any composition language. A graphical representation is given, in order not to obscure the application's design with implementation details. Appendix D describes the diagrams used below.

Given particular implementations of the primitive tools, there are several possible means of composing them to solve problems:

- i. diagrams. A mouse-driven, "box-fitting" shell might allow problems to be solved pictorially, eliminating the requirement for the user to know any form of composition language;
- ii. shell scripts or JCL scripts. Although the most practical solution, these also tend to be the most obscure, in terms of the relationship between the problem and its expression as a script;
- iii. functions written in an (interpreted) applicative language. This solution is definitely the most desirable: problems are solved by applying functions to group presentations. In such an environment, the tools to

and `tt` might usefully be decoupled into smaller functions. The applicative language itself could then be used to re-combine them into definition strategies and transformation meta-schemas. This possibility would provide greater flexibility and homogeneity than does the design of Chapter 1.

Examples of simplified shell scripts (written for `sh(1)`) for many of the composition diagrams in these sub-sections are also appended.

2-2.2 *Groups_involving_Fibonacci_and_Lucas_Numbers*

The groups $Y(n)$, mentioned in [11,12,21], are known to provide difficult examples for the TTRANS program [21]. They were therefore natural candidates for tests of *tt*. The results of those tests are presented here. Furthermore, the tests led to work which was to prove a conjecture from [11]. The proof is shown in Chapter 3.

The groups $Y(n)$ are defined in Chapter 3. Let $H(n) = \langle a, b^2 \rangle$ be a subgroup of $Y(n)$, and put $x = a$, $y = b^2$. We used a breadth-first coset table spanning tree in the production of a presentation of $H(n)$ by the Reidemeister-Schreier method. This produces a starting presentation which is essentially the same as that used in [11].

Now, consider the *tt* algorithm

```
define shorten "0[su <12c; canon]"
define squeeze "shorten; 5[su =12oc?; canon; shorten]"
canon
0[G 2; 0[su <121; canon]]
3[0[4c dabc?; shorten]; 5 squeeze]
5 squeeze
```

The algorithm has random features in it, so we ran it 5 times on each of the groups $H(n)$, where $-10 \leq n < 10$ and $n \neq 0 \pmod{3}$. Taking the best results for each group, we have:

n	ngens	nrels	elen	len
-10	2	3	65	65
-8	2	3	50	50
-7	2	3	30	43
-5	2	2	24	24
-4	2	3	24	31
-2	2	2	15	15
-1	2	2	17	17
1	2	2	15	15
2	2	3	16	19
4	2	2	19	19
5	2	2	21	21
7	2	3	43	43
8	2	3	28	43

Using the *blen* length function (see Section 1-1.2) on $H(-8)$ improves this to:

n	ngens	nrels	elen	len
-8	2	2	29	29

These results compare favourably with those obtained by TTRANS for the difficult examples of $H(8)$ and $H(-8)$:

n	TTRANS				tt			
-8	2	14	807	-	2	2	29	29
8	4	22	881	-	2	3	28	43

The actual presentations found by the above *tt* algorithms are all expressed in terms of the given generators x and y for $H(n)$.

The relators of the best presentations ever found for each n are listed below:

- 8 $yxyx^2 = xy$ $y^8 x^{-2} y^5 xyx^{-1} y^{-1} x^{-1} yx^{-1}$
- 7 $yxyx^2 = xy$ $y^{-7} x^{-1} yx^{-1} y^{-1} x^{-1} y^{-9} x^{-1}$ y^{14}
- 5 $yxyx^2 = xy$ $x^2 yxy^5 xyx^{-1} y^3 x^{-1} y$
- 4 $yxyx^2 = xy$ $y^{-4} x^{-1} yx^{-1} y^{-1} x^{-1} y^{-6} x^{-1}$ y^8
- 2 $yxyx^2 = xy$ $y^2 x^{-1} yx^{-1} y^{-1} x^{-2}$
- 1 $yxyx^2 = xy$ $y^{-1} x^{-1} yx^{-1} y^{-1} x^{-1} y^{-3} x^{-1}$
- 1 $yxyx^2 = xy$ $yx^{-1} yx^{-1} y^{-1} x^{-1} y^{-1} x^{-1}$
- 2 $yxyx^2 = xy$ $y^2 x^{-1} yx^{-1} y^{-1} x^{-2}$ y^4
- 4 $yxyx^2 = xy$ $y^4 x^{-1} yx^{-1} y^{-1} x^{-1} y^2 x^{-1}$
- 5 $yxyx^2 = xy$ $y^5 x^{-1} yx^{-1} y^{-1} x^{-1} y^3 x^{-1}$
- 7 $yxyx^2 = xy$ $y^7 x^{-1} yx^{-1} y^{-1} x^{-1} y^5 x^{-1}$ $y^6 xy^{-6} xy^2 x^2$
- 8 $yxyx^2 = xy$ $y^8 x^{-1} yx^{-1} y^{-1} x^{-1} y^6 x^{-1}$ y^{16}

2-2.2.1 Conjecture

The common form of these presentations prompts us to make two (related) conjectures about the groups $H(n)$:

1. The relation

$$yxyx^2 = xy \tag{2-1.1}$$

always holds in $H(n)$ whenever $n \not\equiv 0 \pmod{3}$.

2. The group $H(n)$, as defined above, has a presentation on generators $x = a$ and $y = b^2$ of $Y(n)$, with relations (2-1.1) and

$$y^n x^{-1} y x^{-1} y^{-1} x^{-1} y^{-n-2} x^{-1} = 1 \quad (2-1.2)$$

Furthermore, the order of x is $4n$, while the order of y is $2n$.

Conjecture 1 can be shown (Chapter 3) to be equivalent to the isomorphism conjecture of [11]. Simple tt algorithms have been found which prove this conjecture for the cases $n = \pm 1, \pm 2, \pm 4, \pm 5, \pm 7, \pm 8, \pm 10, \pm 11$. Chapter 3 contains a proof of the conjecture for all cases.

The stronger result, Conjecture 2 above, remains unproved.

2-2.3 Proofs_and_Derivations_using_'tt'

The examples in the previous Section show that one can use *tt* to prove conjectures by exhibiting relations in groups, to a certain extent. In this Section we take these ideas one stage further, using *tt* and the *trace(3)* library to derive proofs for relations.

Let $P = \langle X \mid R \rangle$ be the group presentation which is currently being edited within *tt*.

2-2.3.1 Tracing_in_'tt'

In order to provide for word derivations in *tt*, it was found to be necessary to be able to keep track of each relator of P by means simpler than by pattern-matching. An array *relIds* of Rel pointers was introduced into *tt*, which is initialised at start-up so as to contain pointers to the relators of P . Thereafter the array is not altered. Indexes into this array provide absolute relator identifiers for the duration of a *tt* session.

The most useful trace subjects for derivations are *ttElim* and *ttSu*. Consider the following simple example, with various levels of trace output enabled.

```
tracelevel = 0
```

```
* p
<a, b, c | abac, bcba, cabca>
* g; p
<b, c | bbc-lbc, bcbcbc-l>
* su; p
<b, c | bbc-lbc, c>
*
```

tracelevel = 3

```
* g
Eliminating generator: a
* su
*
```

tracelevel = 6

```
* g
Eliminating generator: a = b-lc-lb-l
* su
Replacing relator bc-lbcbc
      to get c
*
```

tracelevel = 9

```
* g
Eliminating generator: a = b-lc-lb-l
      changed abac -> cb-lc-lb-lc-lb-l
      changed cabca -> cb-2c-lb-l
* su
Replacing relator bc-lbcbc
      using bc-lbcb = 1
      to get c
*
```

tracelevel = 12

```
* g
Eliminating generator: a = b-lc-lb-l (relator [1])
      changed [0] abac -> cb-lc-lb-lc-lb-l
      changed [2] cabca -> cb-2c-lb-l
* su
Replacing relator [0] bc-lbcbc
      using [2] bc-lbcb = 1
      to get [0] c
*
```

Relator-ids are only produced at the highest level, as a last resort.

2-2.3.2 *An_Example:_H(n)*

In considering how to prove Conjecture 1 above, the first step was to find the shortest repeatable *tt* algorithm which produces the required word. In fact, Corollary 3-1.3 shows that any of a large family of relations would prove the conjecture, which eases the task somewhat.

If we define a *tt* macro *iv* by

```
define SU "0[su <12c; canon]"
define iv "{su =1c; canon; SU; su =2c; canon; SU}"
```

the following *tt* commands prove the conjecture in relatively few operations:

n	Command	Tracefile
4	canon; g k; iv; g j; iv	27K
5	canon; g m l k j; 3iv	60K
7	canon; g q p o n m; 3iv; g l k; 3iv	105K

(remembering that *rewrite* produces Schreier generators *a*, *b*, *c*, ...). Although the trace output was very large in each case, the simple rules listed in the next section allowed us to extract fairly short derivations for each case. In fact, the basic plan in each case turned out to be the same. This plan has been generalised in Chapter 3, in which Tietze transformations show Conjecture 1 to be true.

2-2.3.3 *A_Derivation_Tool*

The trace output produced by the *tt* commands above (indeed by any usage of *tt*) is extremely redundant when one wishes only to find the derivation of a single relation. The trace needs to be simplified before it becomes intelligible as a "proof". While simplifying the examples *H*(4), *H*(5) and *H*(7) by hand, the following heuristics became apparent:

- i. some relators will be unused in deriving the target word. Any substitutions or eliminations which involve these words can therefore be deleted. This can be done using a directed graph of the dependencies between Tietze transformations;
- ii. upto 90 per-cent of the equal-length substitutions will undo themselves later on. These pairs of transformations can be deleted;
- iii. inverse pairs of substitutions can be deleted even when they act on a relator which is altered in between, as long as the affected substrings don't overlap;
- iv. in fact, any consecutive pair of substitutions can be deleted if the presentation has not been altered by them;
- v. any pair of substitutions can be commuted in the trace if they involve four different relators;
- vi. elimination of a generator can be replaced by a series of (possibly lengthening) substitutions;
- vii. any application of rules (ii)-(v) should be followed by (i), in case some relator is now unused.

These heuristics should be fairly easy to implement, together with a reporter which produces a "proof" of the required relation. This has not yet been done.

2-2.4 *Choice_of_Spanning_Tree*

The primitive tool *st* permits the construction of a wide variety of coset table spanning trees. This Section demonstrates some of the effects of this choice on the Reidemeister-Schreier process, and on subsequent Tietze transformation methods for the resulting subgroup presentation.

Throughout this Section we shall use three of the examples introduced in Section 2-1.4: H_5 , G and $H(8)$. To the coset tables for each of these examples we applied two methods of constructing a spanning tree:

brs

a standard breadth-first traversal of the Schreier diagram, using the command '*st -b*'.

frs

a depth-first tree, with coset table columns ordered according to decreasing generator frequency, using the command '*st -d -c*', with a P-file containing the generator ordering.

For example, these methods produce the following P-files when applied to subgroup G :

brs

```
<x, y | x-ly-lx-4y-l, x-ly-lx-3y, x-ly-lx-2y-l, x-ly-lx-2y,
        x-ly-lx-ly-l, x-ly-lx-ly, x-ly-2, x-ly, x, y-l, y>
```

frs

$$\langle x, y \mid x^2yx^5yx^5, x^2yx^5yx^3yx^2 \rangle$$

These, together with the original group presentations, are passed to *rewrite*, producing Reidemeister presentations for the subgroups which have the following characteristics:

	brs			frs		
	ngens	nrels	elen	ngens	nrels	elen
H5	70	258	980	57	220	973
G	19	36	171	19	36	171
H(8)	20	38	460	20	38	460

(using column headings as before). Despite the similarity of size, the *brs* and *frs* methods produced very different presentations for examples G and H(8).

To these subgroup presentations we now apply algorithms a1 - a4 (defined in Section 2-1.4.3). Each algorithm is applied 10 times to each presentation, to counter the effects of their in-built randomness. In the results tables, columns have the same meanings as in Section 2-1.4. The results were:

Example H_5 :

	brs				frs			
	i	ii	iii	iv	i	ii	iii	iv
a1	11.0	87	1	90	17.0	106	8	107
a2	8.1	65	1	74	14.5	95	1	95
a3	2.1	19	9	19	2.3	19	7	19
a4	2.1	19	2	55	2.0	19	2	68

Here we find that presentations resulting from method *brs* allow more generators to be eliminated, and that consequently *brs* is more applicable to H_5 than is *frs*.

Example G:

	brs				frs			
	i	ii	iii	iv	i	ii	iii	iv
a1	17.0	145	1	148	3.5	242	1	410
a2	7.4	78	1	180	3.4	70	1	420
a3	3.0	82	3	198	3.3	290	1	492
a4	3.0	312	1	593	3.2	458	1	775

Here we find the reverse of H_5 . Method *frs* consistently allows *tt* to minimise the number of generators (this group requires three generators). However, the best presentations found using *brs* input are always "shorter".

Example H(8):

	brs				frs			
	i	ii	iii	iv	i	ii	iii	iv
a1	7.6	29	1	1064	6.3	29	1	1403
a2	4.2	2663	1	1644	3.4	92	1	1426
a3	2.7	45	1	1507	3.5	52	1	1409
a4	2.0	117	1	1555	2.0	108	1	1071

In general, *frs* seems to have worked well with H(8), and slightly better than has method *brs*.

These tables only scratch the surface; there are countless possibilities here which have not been explored. In particular, it is likely that some combinations of spanning tree and *tt* algorithm perform well in the majority of cases, and that these could be sought out and embodied in shell scripts or heuristics.

2-2.5 *Optimising_Coset_Enumeration*

In this Section we shall demonstrate some uses of the primitive tool *tc*, concentrating on ways of using related groups to help an enumeration succeed. Let G , G_1 and G_2 be groups, with presentations P , P_1 and P_2 respectively.

2-2.5.1 *Basic_Coset_Enumeration*

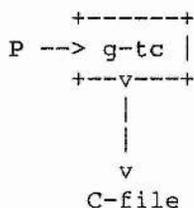
There are two different ways of using *tc* (the following names are simply for reference here - they do not relate to any implementation of *tc*):

g-tc The first, more natural way, is to force closure of a coset table, defining new coset-ids as necessary.

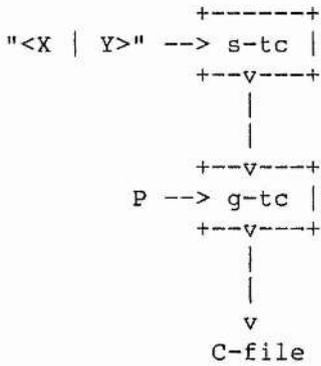
s-tc The second, restricted form, simply forces R-closure at coset-id 1 (see Section 1-3.1) for every word w in *tc*'s input P -file. This second form is designed specifically to produce a partial coset table which existentially defines a subgroup of G .

Consider the following three basic usages of *tc*:

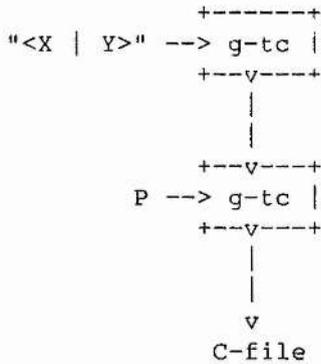
1. to enumerate the elements of G (ie. the cosets of $\langle 1 \rangle$ in G), use the *g-tc* form:



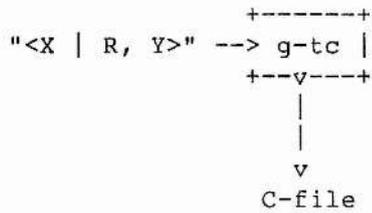
2. to enumerate the cosets of a subgroup H of G , generated by a set Y of words expressed in terms of the generators X of G :



3. to enumerate the cosets of the normal subgroup of G which is generated by the words Y :



In practice, this last is very unlikely to succeed. A more realistic solution to the same problem is:



(where $P = \langle X | R \rangle$).

Henceforth, the two forms of tc will not be distinguished, because everything that applies to subgroups also applies to normal subgroups.

2-2.5.2 *Optimising_the_Reidemeister-Schreier_Method*

Later in this Chapter (Section 2-2.6) we shall describe various ways of presenting a subgroup, using the Reidemeister-Schreier method. For now, assume that there exists a tool *rs*, which uses a coset table (C-file) to direct the rewriting of a group presentation (P-file) into a subgroup presentation (P-file). In most cases the same presentation *P* for *G* will be used both in *tc* and in *rs*. However, it may be that the only way to get *tc* to succeed is to use some specific definition strategy, which would benefit from a particular canonical form of *P*. The de-coupled process modules of the primitive tool-set provide easily for this eventuality:

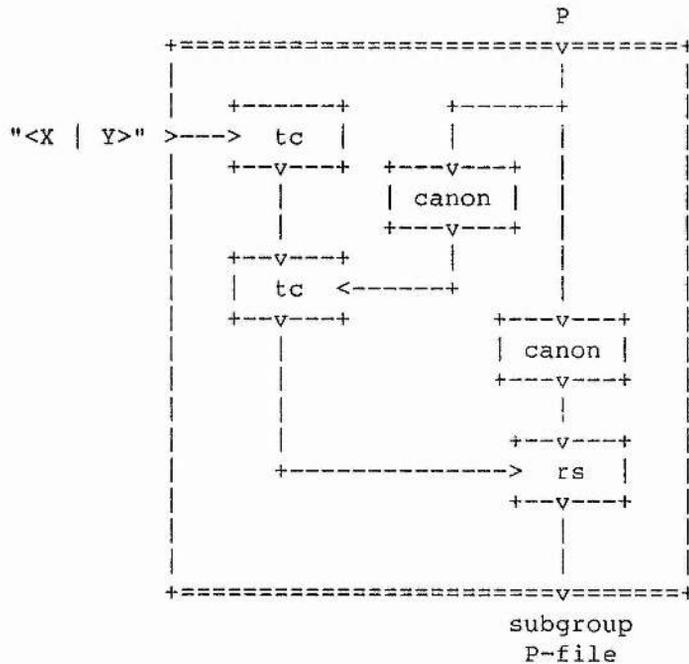


Figure 2-2.1. 'rsopt' - Optimised Reidemeister-Schreier

The two instances of *canon* can be used to "optimise" *P* in the style which is

most appropriate for each application.

2-2.5.3 *Enumerating_in_a_Related_Group*

Suppose that $P_1 = \langle X \mid R_1 \rangle$ and $P_2 = \langle X \mid R_2 \rangle$ present groups in which one expects that the coset tables of the subgroup $H = \langle X \mid Y \rangle$ are related. Suppose further that simple enumeration of the cosets of H in G_1 succeeds, while the enumeration of $[G_2:H]$ fails. A typical example of this situation arises when G_2 is a covering group for G_1 , so that $[G_2:H]$ is larger than $[G_1:H]$.

In some cases, it may be possible to "seed" the enumeration of $[G_2:H]$, using the coset-id definitions which were made during the successful enumeration. These definitions form a spanning tree for the coset table of H in G_1 . The primitive tool *extend* takes a tree of words (such as a spanning tree) and ensures that their products with a particular coset-id (usually coset-id 1, the subgroup H) are all defined:

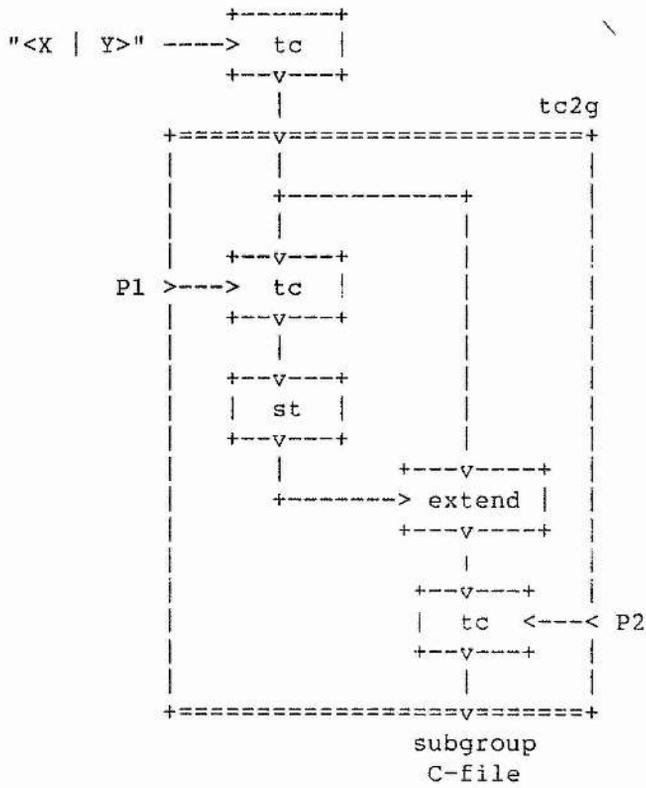


Figure 2-2.2. 'tc2g' - Related Groups

The enumeration of the cosets of H in G_2 now "starts" with a plausible set of coset-id definitions.

A shell script which implements this idea might be prototyped thus:

```

#!/bin/sh
#-----
# Module:      tc2g.sh
# Author:      K.Rutherford
# Version:     @(#) tc2g.sh 1.1 (89/01/29)
# Purpose:     Use a spanning tree to help in a similar coset
#              enumeration
#-----
# Stdin:       subgroup definition (unless -C)          [C-file]
# Stdout:      subgroup coset table                    [C-file]
# Bugs:
#             - No error-checking
#             - temporary files not removed
#             - poor support for primitive tool parameters
#-----
SGFILE=/usr/tmp/sg.$$
STFILE=/usr/tmp/st.$$
USAGE="$0 [tc_args] -P P-file P-file"
carg=""
input_sg="cat $SGFILE |"
tc_args=""

for arg in $*; do
  case "$arg" in
    -C)    carg="-C"
           input_sg=""
           ;;
    -P)    shift; P1="$1"
           shift; P2="$1"
           ;;
    *)    tc_args="$tc_args $arg" ;;
  esac
done

if [ "$carg" != "-C" ]; then
  cat > $SGFILE
fi

$input_sg tc -P $P1 | st > $STFILE
$input_sg extend $carg -P $STFILE | tc -P $P2

```

2-2.5.4 Enumerating_over_Related_Subgroups

A similar problem to that above is the situation in which the enumeration of $[G:H_1]$ succeeds, but the enumeration of $[G:H_2]$ fails. The tools *st* and *extend* can again be used to "seed" the definition strategy used in the second enumeration:

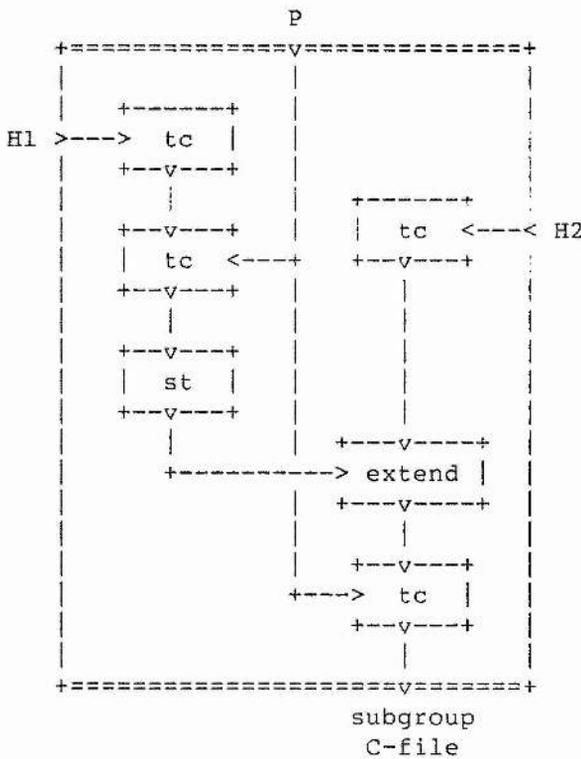


Figure 2-2.3. 'tc2sg' - Related Subgroups

A shell script which implements this idea might be prototyped thus:

```
#!/bin/sh
#-----
# Module:      tc2sg.sh
# Author:      K.Rutherford
# Version:     @(#) tc2sg.sh 1.1 (89/01/29)
# Purpose:     Use a spanning tree to help enumerate over a similar
#              subgroup.
#-----
# Stdin:       Subgroup definition                [C-file]
# Stdout:      Subgroup coset table                [C-file]
# Bugs:
#              - No error-checking
#              - temporary files not removed
#              - poor support for primitive tool parameters
#-----
TMP="/usr/tmp/tc2sg.$$"
USAGE="Usage: $0 P-file P-file"

sgl_file="$1"
gp_file="$2"
cat > $TMP.sg2

tc -C -s -P $sgl_file | tc -P $gp_file | st > $TMP.st
tc -C -s -P $TMP.sg2 | extend -P $TMP.st | tc -P $gp_file
```

2-2.6 *Presenting_a_Subgroup*

Applications of the Reidemeister-Schreier method of presenting subgroups are shown in this Section. The primitive tools are plugged together in various ways to present a variety of subgroups of a group. The approach taken in all applications is the standard Reidemeister-Schreier method:

- a. construct a set of Schreier generators for the subgroup;
- b. rewrite the relators of the super-group according to these generators and a coset table for the subgroup in the super-group.

The process models used here for the primitive tools are described in Appendix D. Shell script implementations of many of the applications depicted below can be found in this Section also, using an alternative architecture.

2-2.6.1 *A_Reidemeister-Schreier_Process*

First, here is a simple utility which implements the basic Reidemeister-Schreier algorithm for the construction of a subgroup presentation. The utility is a filter on a group presentation, using a coset table to drive the rewriting of the P-file into one which presents the subgroup represented by that coset table.

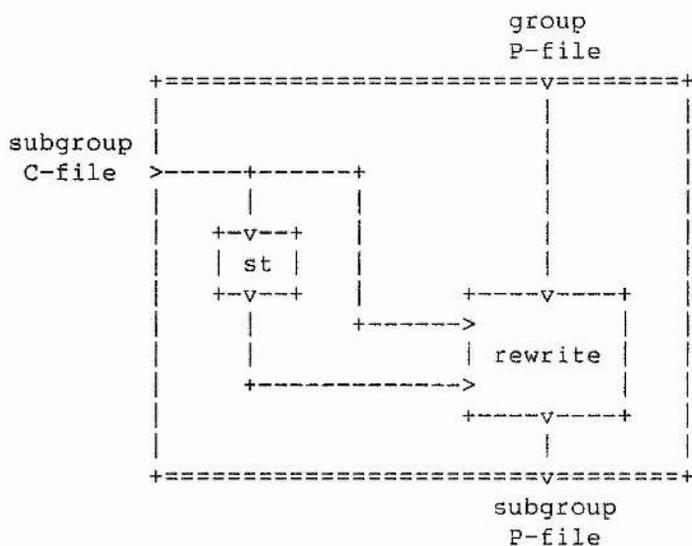


Figure 2-2.1. 'rs' - a Reidemeister-Schreier filter

Remarks:

- ⊕ note that, as far as *rs* is concerned, the coset table and group presentation are completely decoupled. They may have arisen independently of each other.
- ⊕ consequently the coset table may have arisen by means other than by the Todd-Coxeter method.

2-2.6.2 *Todd-Coxeter_and_Reidemeister_Schreier*

Now here is a simple composition which emulates the complete traditional Reidemeister-Schreier method:

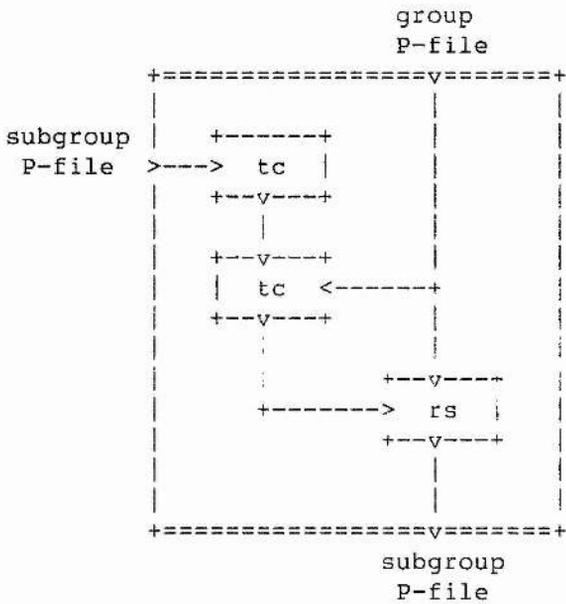


Figure 2-2.2. Using Subgroup Generators with 'rs'

2-2.6.4 Subgroups_Containing_the_Commutator_Subgroup

This approach can be generalised to present any subgroup of the commutator subgroup. First we generalise *dgrs* so that it takes the same parameters as *rs*:

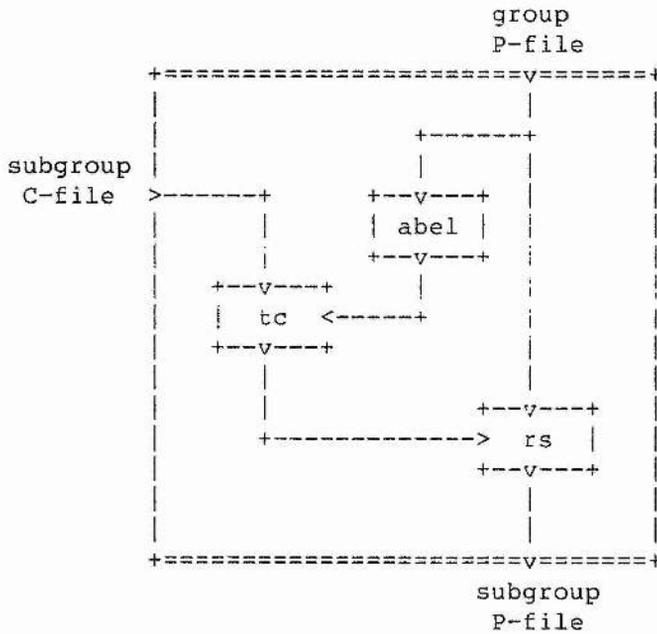


Figure 2-2.4. 'dgrs' - Generalised version

Now we use this as if it were *rs*, to present a subgroup containing the derived group of G :

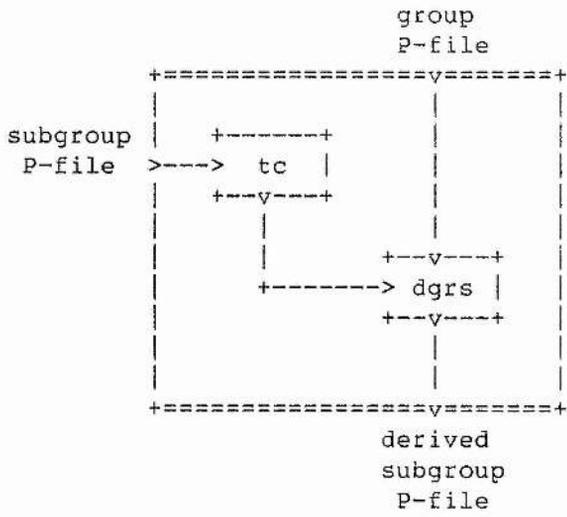


Figure 2-2.5. 'sdgrs' - Subgroups containing the Derived Group

2-2.6.5 Presentations_of_Normal_Subgroups

A similar technique allows us to present any normal subgroup of a group, by abstracting out *abel* from the above process. The following application presents any normal subgroup whose factoring relations are known:

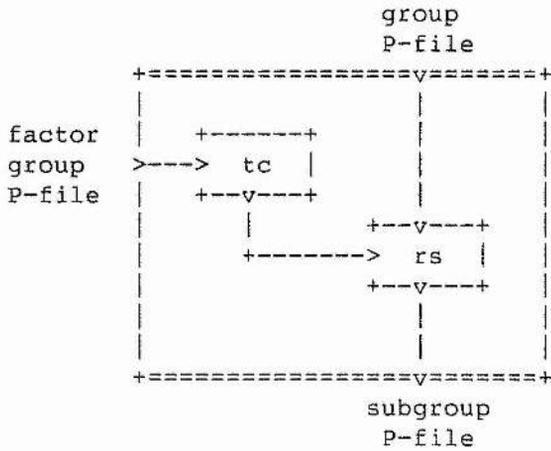


Figure 2-2.6. 'nrs' - Presentation of a Normal Subgroup

2-2.6.7 *Shell_Scripts_which_Present_Subgroups*

The approach adopted in the implementation of these shell scripts is different to that suggested above. This is simply to demonstrate a variety of approaches.

First, a Reidemeister-Schreier process:

```
#!/bin/sh
#-----
# Module:      rs.sh
# Author:      K.Rutherford
# Version:     @(#) rs.sh 1.2 (89/05/05)
# Purpose:     Simple Reidemeister-Schreier filter
#-----
# Stdin:       Group presentation           [P-file]
# Stdout:      Subgroup presentation       [P-file]
# Bugs:
#             - No error-checking
#             - temporary files not removed
#             - poor support for primitive tool parameters
#-----
TMP="/usr/tmp/rs.$$"
USAGE="Usage: $0 C-file < P-file"

sg_file="$1"

st < $sg_file > $TMP.st
rewrite -C $sg_file -T $TMP.st
```

Now we implement *nrs* and *snrs*, which produce a presentation for a subgroup of a factor group of a finitely presented group:

```

#!/bin/sh
#-----
# Module:      nrs.sh
# Author:      K.Rutherford
# Version:     @(#) nrs.sh 1.2 (89/05/05)
# Purpose:     Presents a subgroup containing a normal subgroup
#-----
# Stdin:       Group presentation           [P-file]
# Stdout:      Subgroup presentation       [P-file]
# Bugs:
#             - No error-checking
#             - temporary files not removed
#             - poor support for primitive tool parameters
#-----
TMP="/usr/tmp/nrs.$$"
USAGE="Usage: $0 [C-file] P-file"

case $# in
1)
    input_sg=""
    carg="-C"
    ng_file="$1"
    ;;
2)
    input_sg="cat $1 |"
    carg=""
    ng_file="$2"
    ;;
esac

eval $input_sg tc $carg -P $ng_file > $TMP.ct
rs $TMP.ct

```

```
#!/bin/sh
#-----
# Module:      snrs.sh
# Author:      K.Rutherford
# Version:     @(#) snrs.sh 1.2 (89/05/05)
# Purpose:     Presents a subgroup containing a normal subgroup
#-----
# Stdin:       Group presentation                [P-file]
# Stdout:      Subgroup presentation            [P-file]
# Bugs:
#             - No error-checking
#             - temporary files not removed
#             - poor support for primitive tool parameters
#-----
TMP="/usr/tmp/snrs.$$"
USAGE="Usage: S0 [P-file] P-file"

case $# in
1)
    nrs $1
    ;;
2)
    tc -C -s -P $1 > $TMP.ct
    nrs $TMP.ct $2
    ;;
esac
```

The remaining Reidemeister-Schreier methods depicted above are now implemented in terms of *nrs*:

```
#!/bin/sh
#-----
# Module:      tcrs.sh
# Author:      K.Rutherford
# Version:     @(#) tcrs.sh 1.1 (89/01/29)
# Purpose:     Presenting a subgroup from its generators
#-----
# Stdin:       Group presentation           [P-file]
# Stdout:      Subgroup presentation       [P-file]
# Bugs:
#             - No error-checking
#             - temporary files not removed
#             - poor support for primitive tool parameters
#-----
TMP=/usr/tmp/tcrs.$$
USAGE="$0 P-file"

cat > $TMP.gp
snrs $1 $TMP.gp < $TMP.gp
```

```
#!/bin/sh
#-----
# Module:      dgrs.sh
# Author:      K.Rutherford
# Version:     @(#) dgrs.sh 1.2 (89/05/05)
# Purpose:     Presents a subgroup containing the commutator subgroup
#-----
# Stdin:       Group presentation                [P-file]
# Stdout:      Presentation for derived group subgp [P-file]
# Bugs:
#             - No error-checking
#             - temporary files not removed
#             - poor support for primitive tool parameters
#-----
TMP="/usr/tmp/dgrs.$$"
USAGE="Usage: $0 [C-file]"

cat > $TMP.gp
abel < $TMP.gp > $TMP.ab
eval nrs $1 $TMP.ab < $TMP.gp
```

```
#!/bin/sh
#-----
# Module:      sdgrs.sh
# Author:      K.Rutherford
# Version:     @(#) sdgrs.sh 1.2 (89/05/05)
# Purpose:     Presents a subgroup containing the commutator subgroup
#-----
# Stdin:       Group presentation                [P-file]
# Stdout:      Presentation for derived group subgp  [P-file]
# Bugs:
#             - No error-checking
#             - temporary files not removed
#             - poor support for primitive tool parameters
#-----
TMP="/usr/tmp/sdgrs.$$"
USAGE="Usage: $0 P-file"

tc -C -s -P $1 > $TMP.ct
dgrs $TMP.ct
```

2-3. *Using_the_Tool-Kit*

2-3.1 *Introduction*

With the exception of *tt*, the tools designed in Chapter 1 are all non-interactive. Even *tt* can be used non-interactively, by means of a shell 'here' document (see also Appendix D).

This Section describes three applications which use the primitive tools as "black boxes". The applications know little or nothing about the mechanics of coset enumeration, canonicalisation etc. Their "expertise" lies in the use of the primitive tools to achieve certain results.

2-3.2 *The_Group-Theory_Shell*

This Section describes the tool *gsh*, which combines the primitive tools under a simple user interface. *gsh* provides a natural way of using the primitive tools when dealing with a single group presentation.

2-3.2.1 *Rationale*

The primitive tools were designed so as to require no user-interaction. In particular, *tc* differs from the earlier program TC [15] in that only coset enumeration is offered to the user, and only one attempt is carried out.

As a result, the tools are not easy to use. A complex of inter-communicating processes is required to perform the Reidemeister-Schreier method, for instance. Furthermore, in de-coupling the functions provided by TC into several smaller tools, the convenience of using TC has been destroyed. The naive application *gsh* is intended to restore the convenience of using TC, while still providing the power and flexibility of the primitive tool-kit.

A manual page describing *gsh* is given in Appendix C.

2-3.2.2 *Implementation_Notes*

gsh is implemented as a shell script for the Unix Bourne shell *sh(1)*. During the course of an interactive session with *gsh*, the following temporary files are maintained for the user:

Filename	Contents
gsh-g.\$\$	the current group's P-file
gsh-s.\$\$	the current set of subgroup generators, also held in a P-file
gsh-c.\$\$	the current subgroup's coset table, held in a C-file. This file does not always exist
gsh-r.\$\$	the current subgroup's Reidemeister-Schreier presentation. This file does not always exist

(where \$\$ is replaced by the shell's process-id, so that the filenames are unique even when several users are working with *gsh* in the same part of the filestore).

2-3.2.3 Examples

The following is a short interactive session with *gsh*, testing some of the effects of the *-c* argument to *tc* on the group $\text{Cam}(3)$ [7,15]:

```
$ gsh
gsh> group
Enter group P-file: <r, s | r2srsr-3s-1, s2rsrs-3r-1>
gsh> TRACE=6
gsh> enum
start coinc left npass Cdefs Gdefs Idefs Rdefs ndefs qpush
  1      0    1   133  1000    0    0    0  1000    0
 1001    9   992   144  1000    0    0    0  1000    6
 1992   26  1966   147  1000    0    0    0  1000   10
 2966  2846   120    2    0    0    0    0    0
  120    0   120
```

```
Total cosets defined: 3000
Index = 120
```

```

gsh> TC="--t -c8"
gsh> enum
start coinc left npass Cdefs Gdefs Idefs Rdefs ndefs qpush
  1      0    1   133  1000    0    0    0  1000    0
1001    9   992   144  1000    0    0    0  1000    6
1992   26  1966   121   824    0    0    0   824    8
2790   76  2714   101   695    0    0    0   695    8
3409  3289   120    2     0    0    0    0     0    0
  120    0   120

```

```

Total cosets defined: 3519
Index = 120

```

```

gsh> TC="--t -c6"
gsh> enum
start coinc left npass Cdefs Gdefs Idefs Rdefs ndefs qpush
  1      0    1   133  1000    0    0    0  1000    0
1001    9   992   110   764    0    0    0   764    6
1756   26  1730   137   950    0    0    0   950    6
2680   42  2638    49   317    0    0    0   317    6
2955  2835   120    2     0    0    0    0     0    0
  120    0   120

```

```

Total cosets defined: 3031
Index = 120

```

```

gsh> ^D
$

```

2-3.2.4 Discussion

Any simple shell script like *gsh* will prove ideal for solving single problems in isolation. *gsh* copes well with one group and its subgroups, allowing the user to cast around until coset enumeration succeeds or until a nicely-presented subgroup is found. In fact, this paradigm is encouraged by *gsh*, by its provision of run-time variables and by its concept of "current group" etc.

However, the naivety of *gsh* can also place limits on its usefulness, for instance:

- the restriction to working with one group at a time can be off-putting.

Special-purpose shell scripts are very often a better approach when deal-

ing with series of groups or subgroups;

⊕ if one recurses to a Reidemeister-Schreier presentation for a subgroup, hoping that *tt* will reduce it to something usable (or even recognisable), there is no way back to the parent group. This one-way regression means that the user is best advised to keep a "database" of P-files describing the groups he wishes to work on. This can become messy in the Unix file-store;

⊕ the more built-in commands *gsh* has (perhaps we could provide a command which performs *tt* on the subgroup presentation, or *extend* on the current coset table, or ...) the more built-in commands it still needs, as users notice gaps in its facilities or rough edges in its approach. In this respect, *gsh* is the antithesis of the primitive tool-set.

On the whole, it can be seen that *gsh* is a useful tool for solving small problems which remain fairly well-behaved. It was designed primarily as a test-bed for the primitive tools as they were developed. It proved useful in that respect, and incidentally showed that the usefulness of TC is not destroyed when the primitive algorithms are isolated from each other. However, as soon as a problem becomes difficult, or when usage of *gsh* becomes frequent, its model is seen to be weak.

2-3.3 *Goal-Directed_Problem_Solving*

This Section reports on a simple attempt to use the primitive tool-kit to solve problems automatically. The prototype application *gpg* represents a simple-minded approach to finding the order of a group, given a finite presentation for it.

2-3.3.1 *Rationale*

The calculation of the order of a group from its presentation is known to be a non-terminating computation [44]. However, problems of this type are continually solved, by the use of *ad hoc* and judgemental methods. For instance, when direct coset enumeration is unsuccessful, such problems are often solved by finding the order and index of some subgroup of the group in question.

This Section describes a prototype application for the tool-kit, which attempts to use coset enumeration of the group and its subgroups to discover the order of the group. The application has the following mission:

Given a finite group presentation P , attempt to discover the order of the abstract group G presented by P . Failing this, note as much information about G as can be obtained with the means to hand.

2-3.3.2 *Design*

Here we describe the basic algorithm of *gpg*. The approach consists of four stages, the last of which is merely an administrative phase; steps 1-3 carry out analysis of P . If, during the execution of any of these steps, the order of G becomes known, the application is deemed to have succeeded and will jump immediately to step 4.

1. Look for obvious signs that G is finite or infinite. This step consists of asking each of the following questions in turn:

1-a What is G 's derived index? If it is zero, G is infinite.

1-b Does G have a known infinite group as an "obvious" factor? That is, do the relations of G coincide with a subset of the relations of some known infinite group?

1-c Is G an "obvious" factor of any known finite group? That is, does any subset of the relations of G coincide with all of the relations of some known finite group?

The last two of these tests require *gpg* to be supported by a library of group presentations, together with some indication of the order of the presented group in each case.

2. Try to enumerate the cosets of $G/\langle 1 \rangle$, ie. attempt to find the order of G directly by application of the Todd-Coxeter method. This step consists of various applications of the *tc* tool, according to the following (vague) scheme:

2-a Use various of *tc*'s definition strategies, attempting to successfully enumerate the cosets within the memory limits of the local implementation of the tool-kit.

2-b Use *canon* to reorder the relators of P , attempting to suit the different definition strategies employed.

- 2-c Use Nielsen transformations on P (via tt) when it seems appropriate to do so.

This step is best supported by some sort of database which lists successful tc definition strategies, together with canonical forms which have been shown to integrate well with each.

- 3. Try to enumerate $G/\langle 1 \rangle$ indirectly, by looking for subgroups of small finite index. That is, attempt to enumerate the cosets in G of a large and varied collection of subgroup generators, using the single most appropriate tc definition strategy in each case. For each large subgroup H thus found, try the following:

- 3-a If H is generated by a single word w_1 , look for other words w_2 which don't reduce the index when added as extra subgroup generators. Each such w_2 then lies inside $\langle w_1 \rangle$ and therefore commutes with the original subgroup generator. The relation $[w_1, w_2] = 1$ can be added to P 's relations. Go back to step 2.

- 3-b Recurse (slightly) on subgroup H :

- i. Use the Reidemeister-Schreier method, with some effective tt reduction meta-schema, to present H .
- ii. Invoke steps 1-2 on the new presentation of H , but don't recurse through the whole process again.

- 4. Tidy up. Report the results to the application user and record all intermediate results and statistics.

Many of these stages are only feasible with the support of a database of some sort of information or expertise. However, a prototype version of *gpg* has been implemented, and is described below.

2-3.3.3 *Implementation_Notes*

The prototype implementation of the application *gpg* takes the form of two shell scripts, *gpg* and *gp.fg*:

gp.fg implements the central algorithm of this application (see next section).

As input, it expects to find a P-file in the current directory, in a file called *G*. As output it leaves a file *G.log* in the same directory, which lists everything discovered about the group presented in *G*.

gpg is intended simply to be a convenient user interface to *gp.fg*. It collects a group presentation from the user, together with a name for the group presented. It then runs *gp.fg* in the background, arranging for the resulting logfile to be mailed to the invoking user. In the meantime, *gp.fg* is detached from the terminal, allowing the user to continue with other work (possibly starting other jobs for *gp.fg*).

2-3.3.3.1 *The_gp.fg_Algorithm*

The algorithm embodied in *gp.fg* is implemented as a shell script which orchestrates the work of the "black boxes" in the primitive tool-kit. This means that the algorithm is easy to alter at the highest level, without necessitating detailed knowledge of the implementations of actual group-theoretic algorithms or their i/o mechanisms.

The following is a prototype design for the central algorithm of *gp.fg*. Note that all quantities are pure guesses: empirical evidence will direct later versions to have better bounds.

- i. Freely reduce the given presentation, and sort into canonical order.
- ii. Calculate the order of G 's derived factor. If the result is zero, report that G is infinite and halt.
- iii. Enumerate G over the trivial subgroup using Todd-Coxeter. If the order of G is found, halt. The Felsch method for Todd-Coxeter will be used here because, although it can take a long time, it usually defines the fewest cosets, and therefore is the most likely method to succeed, given "infinite" time.
- iv. Find some subgroups of small index, and record their generators. Here, "small" should be no greater than about 50, but this may be relaxed if there are no successes. The subgroups should have no more than three times the number of generators of G .
- v. Produce presentations for the small-index (ie. large) subgroups found earlier, using the Reidemeister-Schreier method. For each such subgroup, reduce the size of the presentation using Tietze transformations, and then try steps i - iii above. If the order of any is found, report the order of G and halt.
- vi. Report failure, and halt.

Note that this method does not recurse on the presentations of large sub-

groups. This would be very costly, and is not a technique frequently used in practice.

2-3.3.4 Discussion

In a few small tests, *gpg* found the order of the group given in the file *G*. In doing so, it provided raw data (timings, statistics) which might help improve its performance in future.

However, the design presented above is messy. It is difficult for software to obtain partial results from a logfile which contains trace output from *tc* or *tt*, but if the logfile were split into several smaller files then the local filestore would soon become cluttered with junk. These "usability constraints" could be overcome, however, by providing the user (and *gpg*) with a database of groups and presentations.

A more serious limitation in the design of *gpg* is that the designers (somewhat *ad hoc*) algorithm choices have been built into the application. The program itself has to be changed if new approaches need to be introduced, and this, by virtue of the somewhat artificial looping structure, could require that much of the program be re-designed. This defeats the point.

2-3.4 *Rule-Based_Problem_Solving*

A high-level application is designed, which uses the primitive tools of Chapter 1 as the building blocks for a suite of heuristics. A database of group presentations and domain expertise structures the users (and the programs) working environment.

2-3.4.1 *Rationale*

The motivation behind the tool *gsh* was to provide an interactive interface to the primitive tool-kit, featuring easy ways to solve certain types of simple problem. This approach, while useful, was seen in Section 2-3.2.4 to be inadequate for work involving difficult problems or several group presentations.

An investigation into the automation of the problem-solving process (*gpg*) showed that the flexibility and modularity of the tool-kit should be mirrored in the representation of domain knowledge and expertise. That is, both the data to be used by our software (groups, presentations) and the *software itself* (tasks, goals, heuristics) should be represented in a database.

These observations mirror the development of *Eurisko* [31,32,33] (and *RLL* [19]) from *AM* [30,34]. Although the application developed below does not "learn", it does allow domain expertise to be represented and structured usefully.

2-3.4.2 *Design*

The *gp* system comprises three major parts: a database, a simple database browsing program and a task executive. These are described below.

2-3.4.2.1 *The 'gp' Database*

The database is organised into *frames* [39], each of which represents a single entity or concept [30,31,32]. Each frame comprises a list of attribute-value pairs, or *slots*. A typical frame might represent a group, a task, a presentation or a heuristic. As in RLL, every attribute which might be possessed by a frame is itself represented by a frame.

2-3.4.2.2 *The 'gp' Executive*

The database and tool-kit are further complemented by a task executive tool, which maintains an *agenda* [30] of tasks. Tasks may be placed on the agenda by heuristics, or by the user of the browsing tool. In fact, the agenda is implemented as a database frame.

Each heuristic in the database knows how to make use of one or more of the primitive tools in order to increase the knowledge of the whole system, and in particular to further the objective(s) of any task which uses it. The objective of any heuristic is to supply a value for a particular slot of some target frame. Other slots may be filled as a side-effect of the heuristic's action. Each heuristic uses the primitive tools in ways which have been found (empirically) to produce results of the type required. A typical heuristic might use *tc* in a particular way to attempt to enumerate the elements of a group.

Each task in the database represents the job of filling some slot of some frame with its "correct" value. A typical task might attempt to compute the order of a group. Each task knows of a selection of heuristics which might

supply the required value.

In order to do its job, any heuristic may depend on values being known for certain slots in the target frame (or in related frames). These dependencies are expressed in a *DependsOn* slot in the frame for the heuristic. If any such information is not available, the executive will schedule the appropriate tasks, putting the heuristic to sleep in the meantime. This means that expertise which is added locally (eg. the introduction of a new heuristic which can test for the solubility of a group) can improve the effectiveness of the whole system.

2-3.4.2.3 *The 'gp' Browser*

The user interface to the database and its executive is the *gp* browser. It allows the user to examine the contents of any frame, and to alter the values on slots, to introduce new slots and to delete slots from the frame.

New frames can be created, by *specializing* or *exemplifying* an existing frame.

The user can add tasks to the agenda, by selecting a frame and invoking the *propose* command. The browser then presents a menu of slot types, each of which is known by the system to be associated with a task relevant to that frame. If the user selects a slot type, the appropriate task is placed on the agenda. This action also wakes the task executive, if necessary. The user now continues to work in the database, with his task proceeding in background.

2-3.4.3 *Implementation_Notes*

A prototype *gp* system has been implemented, and populated with a limited number of heuristics. The heuristics are implemented as shell scripts, using the primitive tools as black boxes.

One particular task in the database takes a group presentation and attempts to find the order of the group. Heuristics known to this task can

- ⊕ use *tc* in various ways, attempting direct enumeration of the group's elements;
- ⊕ use *def* to determine whether the group is infinite, by testing for positive deficiency;
- ⊕ use *dindex* to check for an infinite derived factor;
- ⊕ compute the order simply by multiplying the index and order of some known subgroup.

Dependencies on these heuristics can request sub-tasks which define, present and enumerate subgroups, using *tcs*, *abel* etc.

2-3.4.4 *Examples*

This sub-Section lists the contents of some frames in a small working database. First, the database root:

Name:	Anything, AnyConcept
Generalisations:	Anything
Specialisations:	Group, User, Heuristic, Slot, Task
ExampleOf:	Anything

The universe (according to this simple database) is divided into 5 types of concept. Three of these are shown below:

Name:	Group
Generalisations:	Anything
Specialisations:	SimpleGroup, SolubleGroup, NonSolubleGroup
Examples:	Trivial, A4, Q, SL(2,11), SL(2,13), SL(2,3)

Name:	Task
Generalisations:	Anything
Examples:	FindOrder, FillExampleOf, FindDindex, FindDGroup

Name:	Heuristic
Generalisations:	Anything
Examples:	Heur-1, Heur-2, Heur-5, Heur-20

The (unique) task which can find values for the *Order* slot of a *Group* is:

Name:	FindOrder
Description:	Attempt to find the order of a group
Generalisations:	Anything
ExampleOf:	Task
FillSlot:	Order
Heuristics:	Heur-1, Heur-2

The *Order* slot itself allows the browser to find that task:

Name:	Order
Description:	That attribute of a Group which contains its order
ExampleOf:	FillableSlot
RelevantTo:	Group
FillTask:	FindOrder

The *FindOrder* task's most trivial heuristic simply asserts that the group is infinite if it has zero derived index:

Name:	Heur-1
ExampleOf:	Heuristic
FillSlot:	Order
ForTask:	FindOrder
DependsOn:	DerivedIndex
Script:	IF currentGroup.DerivedIndex = 0 THEN assert currentGroup.Order = 0 (infinity) ENDIF

However, the *DependsOn* slot causes the derived index to be calculated. Once this is done, the information augments the database permanently and in a tidy, well-defined way (as the value of a slot of a frame). Even if the calculation does not allow this heuristic to succeed, sometime later it may be useful.

The following heuristic, for the same task, tries a little harder:

Name:	Heur-2
ExampleOf:	Heuristic
FillSlot:	Order
ForTask:	FindOrder
DependsOn:	Presentation
Script:	IF 'tc' finds order of group THEN assert currentGroup.Order = <index found by 'tc'> ENDIF

The dependency (that a *Presentation* must be given for the group) means that the user could specify this task for a concrete group, allowing the task *FindPresentation* to supply the information needed for coset enumeration.

Now consider the following group, defined in the database:

Name:	A4
ExampleOf:	Group
Presentation:	<S, T S3, T2, (ST)3>

and suppose that the user proposes the task *FindOrder* to the Agenda. *Heur-1* will be put to sleep, having proposed the task

FindDIndex(A4)

to compute A4's derived index. This task will eventually succeed (using the primitive tool *dindex*). Later *Heur-2* also succeeds, so the original task completes successfully. The frame for the group A4 now looks like this:

Name:	A4
ExampleOf:	Group
Presentation:	<S, T S3, T2, (ST)3>
DerivedIndex:	3
Order:	12

2-3.4.5 Discussion

Each refinement of the database contents, say by the addition of a new heuristic or the division of a type of task into two related types, improves the ability of the system to solve user problems. And each problem solved increases the knowledge in the system, which might be used to solve related problems. The system might therefore find uses in two areas:

1. to solve user problems, when the user either lacks sufficient techniques or sufficient on-line time. It is unlikely that the system can ever become intelligent enough to produce quick or elegant solutions to problems. However, sufficient refinement and specialisation of basic heuristics may allow most problems to be solvable by brute force.
2. as a repository of information relating to a particular research topic. Not only can the information be stored here for convenient retrieval, it can also be fed to the primitive tools directly. The results of tests are automatically saved in the right context, providing a sort of interactive notebook with group-theoretic calculator.

These applications may be realised with the implementation of a more robust and efficient implementation of *gp*.

CHAPTER 3

PROOF OF A CONJECTURE

A proof is given for a conjecture of Campbell and Robertson. The proof is a straightforward generalisation of word derivations performed by *tt* during the course of experiments for Chapter 2.

3-0. Introduction

In [11], Campbell and Robertson examine the two series of groups

$$X(n) = \langle a, b \mid (abab^{-1})^n = ba^{-1}bab^{-1}a, ab^2aba^2b = 1 \rangle$$

and

$$Y(n) = \langle a, b \mid (abab^{-1})^n = ba^{-1}bab^{-1}a, ab^2a^{-1}ba^2b^{-1} = 1 \rangle$$

The orders of the $X(n)$ are found. The orders of the $Y(n)$ are shown to be infinite when $n \equiv 0 \pmod{3}$, but otherwise $Y(n)$ is equal to $\bar{X}(n)$ for $-5 \leq n \leq 4$, where $\bar{X}(n)$ is the common homomorphic image of $X(n)$ and $Y(n)$:

$$\bar{X}(n) = \langle a, b \mid (abab^{-1})^n = ba^{-1}bab^{-1}a, ab^2aba^2b = 1, ab^2a^{-1}ba^2b^{-1} = 1 \rangle$$

The authors of that paper go on to conjecture that $Y(n) = \bar{X}(n)$ whenever $n \not\equiv 0 \pmod{3}$. This Chapter presents a proof of that conjecture.

The basic plan of the proof was set out by the primitive tool *tt*, during tests to find general-purpose algorithm macros which worked well for "difficult" examples. In fact, application of some *tt* macros to the groups $Y(-11)$ through $Y(11)$ prompted us to conjecture a much stronger result than is proved here. That conjecture gives a presentation for the subgroup $H(n)$ of $Y(n)$ (see Section 3-1). The conjecture is presented in Section 2-2.2.1, and remains unproven.

As presented here, the proof requires that $n \geq 6$. However, the proof for $1 \leq n \leq 5$ is just the same, but with the most general cases always collapsing to one or other of the special cases. Furthermore, in order to prove the result for $n < 0$, only minor modifications are necessary.

3-1. *A_Subgroup_of_Y(n)*

The group $Y(n)$ is generated by a and b subject to the relations

$$ab^2a^{-1}ba^2b^{-1} = 1 \quad (3-1.1)$$

and

$$(ab^{-1}ab)^n = bab^{-1}aba^{-1} \quad (3-1.2)$$

By analogy with [11] we define the subgroup $H(n) = \langle a, b^2 \rangle$ of $Y(n)$. We now use the Reidemeister-Schreier algorithm to determine a presentation for $H(n)$ in $Y(n)$.

Define cosets $1, 2, \dots, 2n+3$ by $1 = H(n)$ and

$$2i+1 = 2i.a \quad 1 \leq i \leq n+1$$

$$2i = (2i-1).b \quad 1 \leq i \leq n+1$$

No coincidences occur, so $H(n)$ has index $2n+3$ in $Y(n)$. Define a set of $2n+4$ Schreier generators for $H(n)$ by

$$1.a = w.1$$

$$2i.b = x_{2i-1}.2i-1 \quad 1 \leq i \leq n+1$$

$$(2i+1).a = x_{2i}.2i$$

$$1 \leq i \leq n+1$$

$$2n+3.b = z.2n+3$$

The Reidemeister rewriting process now yields the following relators for $H(n)$ from (3-1.1):

$$wx_1w^{-1}x_2 \tag{3-1.3.1}$$

$$x_3x_1w^2x_i^{-1} \tag{3-1.3.2}$$

$$x_{i+1}x_{i-1}x_{i-2}x_{i-1}^{-1} \tag{3-1.3.i}$$

$$z^2x_{2n+1}x_{2n}x_{2n+1}^{-1} \tag{3-1.3.2n+2}$$

$$x_{2n+2}x_{2n+1}x_{2n+2}^{-1}zx_{2n+2}z^{-1} \tag{3-1.3.2n+3}$$

(for $3 \leq i \leq 2n+1$). However, Reidemeister rewriting on (3-1.2) yields relators for $H(n)$ whose length increases with n . These turn out to be inconvenient for our purposes, so we adopt a slightly less direct approach. Define the words

$$h_1 = ab^{-1}ab$$

$$h_2 = ab^{-1}a^{-1}ba^{-1}b^{-1}$$

(so that (3-1.2) says that $h_1^nh_2 = 1$). The Todd-Coxeter process, as applied to

$Y(n)$ over $H(n)$, produces the following partial relator tables from (3-1.2):

h1	
1	4
2	5
3	2
i	$i+4$
$i+1$	$i-3$
$2n$	$2n+3$
$2n+2$	$2n+1$

(where i is even, and $4 \leq i \leq 2n-2$) and

h2	
1	6
2	8
3	4
4	10
5	2
i	$i+6$
$i+1$	$i-5$
$2n-2$	$2n+3$
$2n$	$2n+1$
$2n+2$	$2n-1$

(where i is even, and $6 \leq i \leq 2n-4$). Reidemeister rewriting now produces the following partial relators for $H(n)$ from these tables:

h1		R-S	(tidied)
1	4	$w x_1^{-1}$	-
3	2	$x_2 w$	$w x_1^{-1}$
i	i+4	x_{i+1}^{-1}	-
i+1	i-3	$x_i x_{i-2} x_{i-3}$	x_{i-2}
2n	2n+3	$x_{2n+1}^{-1} z$	-
2n+2	2n+1	$z^{-1} x_{2n+2} x_{2n+1}$	-

and

h2		R-S	(tidied)
1	6	$w x_1^{-1} x_2^{-1} x_4^{-1} x_5^{-1}$	$w x_2^{-1} x_5^{-1}$
3	4	$x_2 w^{-1} x_2^{-1} x_3^{-1}$	-
5	2	$x_4 x_1 w^{-1} x_1^{-1}$	-
i	i+6	$x_{i+1}^{-1} x_{i+2}^{-1} x_{i+4}^{-1} x_{i+5}^{-1}$	$x_{i+2}^{-1} x_{i+5}^{-1}$
i+1	i-5	$x_i x_{i-3}$	-
2n-2	2n+3	$x_{2n-1}^{-1} x_{2n}^{-1} x_{2n+2}^{-1} z^{-1}$	$x_{2n}^{-1} z^{-1}$
2n	2n+1	$x_{2n+1}^{-1} x_{2n+2}^{-1} z$	-
2n+2	2n-1	$z^{-1} x_{2n+1}$	-

(where the words in the rightmost column, where present, show the result of

Tietze transformations using the relators (3-1.3) for $H(n)$. Let (3-1.4.i) be the Reidemeister relator arising from the product of coset i with relator (3-1.2) of $Y(n)$.

3-1.1 LEMMA

In $Y(n)$, the following two relations

$$ab^2aba^2b = 1 \tag{3-1.5.1}$$

$$b^2ab^2a^2 = ab^2 \tag{3-1.5.2}$$

are equivalent.

Proof

From (3-1.1) we have $ab^2 = ba^{-2}b^{-1}a$. Substituting for ab^2 in (3-1.5.2), we have $bab^2a = a^{-2}b^{-1}$, which is (3-1.5.1).

3-1.2 LEMMA

In $H(n)$, the following relations

$$x_1wx_1w^2 = wx_1 \tag{3-1.6.1}$$

$$x_4 = w^2 \tag{3-1.6.2}$$

$$x_6 = x_2 \tag{3-1.6.3}$$

are equivalent.

Proof

First, we show that (3-1.6.1) and (3-1.6.2) are equivalent:

$$1 = wx_1^{-1} w^{-1} x_1 wx_1 w \quad (3-1.6.1)$$

$$= x_2 x_1 wx_1 w \quad \text{by (3-1.3.1)}$$

$$= x_4^{-1} x_2 wx_1 w \quad \text{by (3-1.3.3)}$$

$$= x_1^{-1} w^2 \quad \text{by (3-1.3.1)}$$

Now we show that (3-1.6.1) and (3-1.6.3) are equivalent. First, if (3-1.6.1) holds, then (3-1.6.3) holds:

$$x_2^{-1} x_6 = x_2^{-1} x_4 x_3^{-1} x_4^{-1} \quad \text{using (3-1.3.5)}$$

$$= x_1^{-1} x_2^{-1} x_3^{-1} x_2 x_1 x_2^{-1} \quad \text{using (3-1.3.3) twice}$$

$$= x_1^{-1} wx_1 w^{-1} x_1 w^2 x_1^{-1} wx_1^{-1} w^{-1} x_1 wx_1 w^{-1}$$

using (3-1.3.1) and (3-1.3.2)

$$= 1 \quad \text{using (3-1.6.1) thrice}$$

The reverse implication, that (3-1.6.1) holds whenever (3-1.6.3) holds, will be proved as corollary 3-2.1.1.

3-1.3 COROLLARY

To prove that $Y(n) \cong \overline{X}(n)$, it suffices to show that any of the relations (3-1.6) holds in $H(n)$.

Proof

In the above application of the modified Todd-Coxeter method to the cosets of $H(n)$ in $Y(n)$, the subgroup generators yield the relations $w = a$ and $x_1 = b^2$. So the relation (3-1.5.2) in $Y(n)$ is the same as relation (3-1.6.1) in $H(n)$.

Now, to show that $Y(n) \cong \overline{X}(n)$, we need to show that relation (3-1.5.1) holds in $Y(n)$. By Lemma 3-1.1, this is equivalent to showing that (3-1.5.2) holds in $Y(n)$. The above argument shows that this can be done by proving that any of the relations (3-1.6) holds in $H(n)$.

3-2. *Some_Useful_Relations*3-2.1 *LEMMA*

The relations

$$x_3 w^2 = x_7 x_4 \quad (3-2.1.0)$$

$$x_{i+3} x_i = x_{i+7} x_{i+4} \quad (3-2.1.i)$$

$$x_{2n-1} x_{2n-4} = z^2 x_{2n} \quad (3-2.1.2n-4)$$

(for $1 \leq i \leq 2n-5$) all hold in $H(n)$.

Proof

The proof divides into several special cases, depending on i above, as follows:

Case i:

The proof of this case depends upon whether i is odd or even:

i even:

Let i be even, with $2 \leq i \leq 2n-6$. The relator tables from the Reidemeister rewriting process for (3-1.2) produce the following relators

$$x_{i+5}^{-1} \cdot t \cdot x_i^{-1} x_{i+3}^{-1} = 1 \quad (3-1.4.i+4)$$

$$t \cdot x_{i-1}^{-1} x_{i+4}^{-1} x_{i+7}^{-1} = 1 \quad (3-1.4.i+8)$$

(where t is the word obtained by rewriting the product $i+8.h_1^{n-1} = i-2$). Eliminating t , we have

$$\begin{aligned} 1 &= x_{i+5}^{-1} x_{i+3}^{-1} x_i^{-1} x_{i-1}^{-1} x_{i+4}^{-1} x_{i+7}^{-1} \\ &= x_{i+5}^{-1} x_{i+3}^{-1} x_{i+2}^{-1} x_i^{-1} x_{i+4}^{-1} x_{i+7}^{-1} && \text{using (3-1.3.i+1)} \\ &= x_{i+3}^{-1} x_i^{-1} x_{i+4}^{-1} x_{i+7}^{-1} && \text{using (3-1.3.i+4)} \end{aligned}$$

i odd:

Proof as for i even, matching common substrings of the Reidemeister relators (3-1.4.i+2) and (3-1.4.i-2).

Case 0:

Proof as for Case i , matching common substrings of the Reidemeister relators (3-1.4.4) and (3-1.4.8).

Case 1:

Proof as for Case i , matching common substrings of the Reidemeister relators (3-1.4.2) and (3-1.4.3).

Case $2n-4$:

Proof as for Case i, matching common substrings of the Reidemeister relators (3-1.4.2n) and (3-1.4.2n+3).

3-2.1.1 COROLLARY

If (3-1.6.3) holds in $H(n)$, then (3-1.6.1) holds also.

Proof

Matching common strings from relators (3-1.4.1) and (3-1.4.4), as in the above Lemma 3-2.1, we have

$$x_6 x_3 = x_4 x_2 w^{-1} x_2^{-1} x_3^{-1} x_1 w^{-1}$$

Hence,

$$x_6 x_3 = x_2 x_1^{-1} w^{-1} x_2^{-1} x_3^{-1} x_1 w^{-1} \quad \text{using (3-1.3.3)}$$

iff

$$x_6 x_3 = x_2 x_1^{-1} w^{-1} x_2^{-1} x_1 w \quad \text{using (3-1.3.2)}$$

iff

$$x_6 x_3 = x_2 w^{-1} x_1 w \quad \text{using (3-1.3.1)}$$

iff

$$x_3 = w^{-1} x_1 w \quad \text{using (3-1.6.3)}$$

iff

$$x_1 w^{-2} x_1^{-1} = w^{-1} x_1 w \quad \text{using (3-1.3.2)}$$

which is (3-1.6.1), as required.

3-2.2 LEMMA

The relations

$$x_9 x_5 x_1 = x_5 x_1 w^{-2} x_4 x_1 \quad (3-2.2.1)$$

$$x_{10} x_6 x_2 = x_6 x_2 x_1^{-1} x_5 x_2 \quad (3-w.2.2)$$

$$x_{11} x_7 x_3 = x_7 x_3 w^{-1} x_1 w \quad (3-2.2.3)$$

$$x_{12} x_8 x_4 = x_8 x_4 w^2 \quad (3-2.2.4)$$

$$x_{i+8} x_{i+4} x_i = x_{i+4} x_i x_{i-4} \quad (3-2.2.i)$$

(for $5 \leq i \leq 2n-6$) all hold in $H(n)$.

Proof

Let $i \leq N$ be such that $5 \leq i \leq 2n-6$.

$$\begin{aligned} & x_i^{-1} x_{i+4}^{-1} x_{i+8} x_{i+4} x_i \\ &= x_i^{-1} x_{i+4}^{-1} x_{i+6} x_{i+5}^{-1} x_{i+6}^{-1} x_{i+4} x_i \quad \text{using (3-1.3.i+7)} \\ &= x_i^{-1} x_{i+3}^{-1} x_{i+4}^{-1} x_{i+5}^{-1} x_{i+4} x_{i+3} x_i \quad \text{using (3-1.3.i+5) twice} \\ &= x_i^{-1} x_{i+3}^{-1} x_{i+5}^{-1} x_{i+7} x_{i+4} x_{i+3} x_i \quad \text{using (3-1.3.i+6)} \end{aligned}$$

$$= x_i^{-1} x_{i+2} x_{i+3}^{-1} x_{i+7} x_{i+4} x_{i+3} x_i \quad \text{using (3-1.3.i+4)}$$

$$= x_i^{-1} x_{i+2} x_{i+3}^{-1} x_{i+3} x_i x_{i+3} x_i \quad \text{using (3-2.1.i)}$$

$$= x_{i-1}^{-1} x_{i+3} x_i \quad \text{using (3-1.3.i+1)}$$

$$= x_{i-1}^{-1} x_{i-1} x_{i-4} \quad \text{using (3-2.1.i-4)}$$

The other cases are similar, arising from the special cases of the Reidemeister relations (3-1.4).

3-2.3 LEMMA

The relation

$$x_{2n+1} x_{2n} x_{2n-1} x_{2n-4} x_{2n-8} = x_{2n} x_{2n-1} x_{2n-4} \quad (3-2.3)$$

holds in $H(n)$.

Proof

$$\begin{aligned} & x_{2n-4}^{-1} x_{2n-1}^{-1} x_{2n}^{-1} x_{2n+1} x_{2n} x_{2n-1} x_{2n-4} \\ &= x_{2n-4}^{-1} x_{2n-1}^{-1} x_{2n}^{-1} z^{-2} x_{2n+1} x_{2n-1} x_{2n-4} \quad \text{using (3-1.3.2n+2)} \\ &= x_{2n-4}^{-1} x_{2n-1}^{-1} x_{2n}^{-1} z^{-2} x_{2n-1} x_{2n-2}^{-1} x_{2n-4} \quad \text{using (3-1.3.2n)} \\ &= x_{2n-4}^{-1} x_{2n-1}^{-1} x_{2n-4}^{-1} x_{2n-1}^{-1} x_{2n-1} x_{2n-2}^{-1} x_{2n-4} \quad \text{using (3-2.1.2n-4)} \end{aligned}$$

$$= x_{2n-4}^{-1} x_{2n-1}^{-1} x_{2n-5}$$

using (3-1.3.2n-3)

$$= x_{2n-8}^{-1} x_{2n-5}^{-1} x_{2n-5}$$

using (3-2.1.2n-8)

3-3. *The_Isomorphism_Theorem*

Before proceeding to prove the main theorem of this Chapter, we define some words which will be useful shortly: Let

$$p = x_{2n-5} x_{2n-9} \cdots x_{13} x_9 x_5 \quad (n \text{ odd}) \quad (3-3.1)$$

$$x_{2n-5} x_{2n-9} \cdots x_{11} x_7 x_3 \quad (n \text{ even})$$

and

$$q = x_{2n-4} x_{2n-8} \cdots x_{10} x_6 x_2 \quad (n \text{ odd}) \quad (3-3.2)$$

$$x_{2n-4} x_{2n-8} \cdots x_{12} x_8 x_4 \quad (n \text{ even})$$

and

$$r = x_{2n-1}^{-1} x_{2n}^{-1} x_{2n+1} x_{2n} x_{2n-1} \quad (3-3.3)$$

Then (3-1.4.2n+3) can be expressed as

$$x_{2n}^{qwx_1^{-1}} p^{-1} x_{2n}^{-1} z^{-1} = 1 \quad (3-1.4.2n+3)$$

while Lemma 3-2.3 states that

$$x_{2n-4}^{-1} r x_{2n-4} = x_{2n-8}^{-1}$$

3-3.1 LEMMA

$$\begin{aligned}
 p^{-1}x_{2n-1}p &= w^{-1}x_1w && \text{when } n \equiv 0 \pmod{6} \\
 &x_1 && \text{when } n \equiv 1 \pmod{6} \\
 &x_3 && \text{when } n \equiv 2 \pmod{6} \\
 &x_5 && \text{when } n \equiv 3 \pmod{6} \\
 &x_3^{-1}x_7x_3 && \text{when } n \equiv 4 \pmod{6} \\
 &x_5^{-1}x_9x_5 && \text{when } n \equiv 5 \pmod{6}
 \end{aligned}$$

Proof

Repeated application of Lemma 3-2.2 to the middle seven elements of $p^{-1}x_{2n-1}p$ shortens the word by six letters each time.

3-3.2 LEMMA

$$\begin{aligned}
 q^{-1}rq &= \\
 &x_4^{-1} && \text{when } n \equiv 0 \pmod{6} \\
 &x_2^{-1}x_6^{-1}x_2 && \text{when } n \equiv 1 \pmod{6} \\
 &x_4^{-1}x_8^{-1}x_4 && \text{when } n \equiv 2 \pmod{6}
 \end{aligned}$$

$$x_2^{-1} x_5^{-1} x_1 \quad \text{when } n \equiv 3 \pmod{6}$$

$$w^{-2} \quad \text{when } n \equiv 4 \pmod{6}$$

$$x_2^{-1} \quad \text{when } n \equiv 5 \pmod{6}$$

Proof

The result follows using Lemma 3-2.3, followed by repeated use of Lemma 3-2.2, as in the proof of Lemma 3-3.1 above.

3-3.3 THEOREM

The groups

$$Y(n) = \langle a, b \mid (abab^{-1})^n = ba^{-1}bab^{-1}a, ab^2a^{-1}ba^2b^{-1} = 1 \rangle$$

and

$$\bar{X}(n) = \langle a, b \mid (abab^{-1})^n = ba^{-1}bab^{-1}a, ab^2aba^2b = 1, ab^2a^{-1}ba^2b^{-1} = 1 \rangle$$

are equal whenever $n \not\equiv 0 \pmod{3}$.

Proof

Substituting for x_{2n+2} in the Reidemeister relator (3-1.3.2n+3), we have:

$$1 = x_{2n+2} x_{2n+1} x_{2n+2}^{-1} z x_{2n+2} z^{-1} \quad (3-1.3.2n+3)$$

iff

$$1 = x_{2n}^{-1} x_{2n-1}^{-1} x_{2n}^{-1} x_{2n+1}^{-1} x_{2n}^{-1} x_{2n-1}^{-1} x_{2n}^{-1} z x_{2n}^{-1} x_{2n-1}^{-1} x_{2n}^{-1} z^{-1} \quad \text{using (3-1.3.2n+1)}$$

iff

$$1 = r(x_{2n}^{-1} z x_{2n}^{-1}) x_{2n-1}^{-1} (x_{2n}^{-1} z x_{2n}^{-1})^{-1} \quad \text{using (3-3.3)}$$

iff

$$1 = r(qwx_1^{-1} p^{-1}) x_{2n-1}^{-1} (qwx_1^{-1} p^{-1})^{-1} \quad \text{using (3-1.4.2n+3)}$$

iff

$$1 = (q^{-1} r q) wx_1^{-1} (p^{-1} x_{2n-1}^{-1} p)^{-1} x_1^{-1} w^{-1}$$

Consequently, using Lemmas 3-3.1 and 3-3.2, one of the following relations holds in $H(n)$ whenever $n \not\equiv 0 \pmod{3}$:

$$x_2^{-1} x_6^{-1} x_2^{-1} wx_1^{-1} w^{-1} = 1 \quad n \equiv 1 \pmod{6}$$

$$x_4^{-1} x_8^{-1} x_4^{-1} wx_1^{-1} x_3^{-1} x_1^{-1} w^{-1} = 1 \quad n \equiv 2 \pmod{6}$$

$$w^{-2} x_1^{-1} x_3^{-1} x_7^{-1} x_3^{-1} x_1^{-1} = 1 \quad n \equiv 4 \pmod{6}$$

$$x_2^{-1} wx_1^{-1} x_5^{-1} x_9^{-1} x_5^{-1} x_1^{-1} w^{-1} = 1 \quad n \equiv 5 \pmod{6}$$

Each of these is equivalent to one of the relations (3-1.6), as the following transformations show:

$n \equiv 1 \pmod{6}$:

$$1 = x_6^{-1} x_2^{-1} w x_1^{-1} w^{-1} x_2^{-1}$$

$$= x_6^{-1} x_2$$

using (3-1.3.1)

which is relation (3-1.6.3).

$n \equiv 2 \pmod{6}$:

$$1 = x_4^{-1} x_8^{-1} x_4 w x_1^{-1} x_3^{-1} x_1 w^{-1}$$

iff

$$1 = x_4^{-1} x_8^{-1} x_4 w^2$$

using (3-1.3.2)

iff

$$1 = x_8^{-1} x_{12}$$

using (3-2.2.4)

iff

$$1 = x_5 x_9^{-1}$$

using (3-2.1.5)

iff

$$1 = x_2^{-1} x_6$$

using (3-2.1.2)

which is relation (3-1.6.3).

$n \equiv 4 \pmod{6}$:

$$1 = w^{-2} x_1^{-1} x_3^{-1} x_7^{-1} x_3^{-1} x_1$$

iff

$$1 = x_7^{-1} x_3$$

using (3-1.3.2)

iff

$$1 = x_4 w^{-2}$$

using (3-2.1.0)

which is relation (3-1.6.2).

$n \equiv 5 \pmod{6}$:

$$1 = x_2^{-1} w x_1^{-1} x_5^{-1} x_9^{-1} x_5 x_1 w^{-1}$$

$$= x_2^{-1} w x_1^{-1} x_4^{-1} w$$

using (3-2.2.1)

$$= w^2 x_4^{-1}$$

using (3-1.3.1)

which is relation (3-1.6.2).

Hence, by Corollary 3-1.3, Campbell and Robertson's conjecture [11] is proved.

CHAPTER 4

CONCLUSIONS

The success of the tool-kit design presented in Chapter 1 is reviewed in the light of the results obtained in Chapter 2. Future directions and research questions are explored. The results obtained by the research presented in this thesis are summarised.

4-1. *Further_Tools*

Clearly the primitive tool-set presented in this thesis represents only a beginning. Below are a few tools which could be implemented quite cheaply using the current library of support functions, to plug some gaps in the current tool-kit's functionality:

- ⊕ a tool which determines whether two P-files are "identical". This would entail finding a generator naming scheme and canonical form for one which matched those of the other;
- ⊕ a tool which sorts a collection of P-files into an ordering based on various length and/or symmetry criteria. Such a tool would be useful for automatic scripts which run random *tt* schemas in search of a "nice" presentation;
- ⊕ a tool which determines whether two C-files are isomorphic, in the sense of [42];
- ⊕ a coset enumerator which uses the Knuth-Bendix procedure developed by Sims [43]. Such a tool would be able to perform enumerations which cannot be completed by *tc* in the same memory space;
- ⊕ a variant of *tcs* which outputs, for each subgroup of small index which is found, a set of generators for the subgroup [40]. The generators would be expressed as words in the generators of the supergroup, and would probably contain much redundancy;
- ⊕ a version of *tt* which includes Knuth-Bendix pattern matching [43];

- ⊕ a companion to *tt* in which only Nielsen transformations (and their meta-level support) are available for editing the current presentation;
- ⊕ a version of *tt* which uses a set of relators to reduce another set of words. Such a tool would be useful in some cases to simplify the generating sets produced by the above version of *tcs*;
- ⊕ a tool which produces the derivation of a word in a group, based on trace output from *tt*. The tool would use the methods obtained in Section 2-2.3.3, organised around a directed graph of substitutions relating triples of relators in the group. Subgraphs which are referenced more than once could be isolated and printed as "lemmas" in the final derivation;
- ⊕ a tool which produces all non-isomorphic spanning trees for a coset table, in separate P-files.

Furthermore, the basic set of structured data types could be expanded to encompass strong bases, generating sets etc. For each new data type which is introduced, we should also introduce primitive tools which create and primitive tools which use objects of that type.

Each new tool adds significantly to the problem-solving range of the set, but adds little to the overall complexity. Future use of the tool-set will see its capabilities consolidated around the types of problem most commonly addressed at any site which uses it. Users then construct scripts which compose various tools to solve a particular type of problem (this occurred throughout the tool-set's development, and frequently during the tests for

4-2. *Open Questions*

The design of the primitive tool-set and the test results and scripts presented in Chapter 2 raise many questions which have yet to be answered. The most important, from the point of view of this research, relate to the granularity of the tool-set and to the flexibility of individual tools:

- ⊕ Does *canon* provide enough basic forms ? For instance, are there still better forms which would assist the success of *tc* for certain examples ?
- ⊕ Does *tc* show the same effects as TC does [15] when the group relators or subgroup generators are re-ordered ?
- ⊕ A preliminary run of *tc* is used to generate a partial coset table when enumerating the cosets of a non-trivial subgroup. Can any correlation be found between the size of, or redundancy in, this partial coset table and the definition strategy which was used to generate it ?
- ⊕ Does redundancy in the subgroup's partial coset table help or hinder the final enumeration ?
- ⊕ Consider the coset table seeding mechanisms suggested in Section 2-2.5. Can any direct relationship be found between definition strategies and choice of spanning tree in cases which succeed ? Or in cases which fail ?
- ⊕ Do either of the seeding mechanisms work in a significant number of instances ?

- ⊕ The results of Section 2-1.2 suggest that many coset enumeration examples can be performed by *tc* in less memory than by *TC*. This is of little practical value in circumstances in which we don't know the index in advance. What practical use can be made of *tc*'s definition strategies in these circumstances? How sensible would it be to implement a script which tries several (all?) strategies in the search for success?
- ⊕ The motivation for the inclusion of *tcs* in the primitive tool-kit was to allow an automatic script to address the above situation by looking for subgroups which could be enumerated. Is this a practical proposition? Would such a script have to be so intelligent that user interaction would be required?
- ⊕ Let P_1 be a presentation produced by *rewrite* for a group G , and suppose we know that G has a "nice" presentation P_2 . Given that the facilities of *tt* model a directed graph on the set of all presentations of G , do there exist spanning trees for G such that P_2 cannot be reached from P_1 using *tt*?
- ⊕ Suppose that extensive use of *tt* allowed us to construct a set of reduction meta-schemas which can reduce almost any Reidemeister-Schreier output to a manageable size. Is it then worth having a choice of coset table spanning tree?

Most of these questions will be answered as a side-effect of continued problem-solving using the tool-kit. The results of such work will therefore be interesting from a methodological point of view, as well as in their own right.

APPENDIX A

DATA FILE FORMATS

Concrete formats are defined for the types of data file used by the primitive tools for external communication. These formats are independent of the implementation of the tools which use them.

A-0. *Introduction*

Each of the Sections of this Appendix defines the contents of one of the data file types supported by the primitive tools described in Chapter 1. An extended Backus-Naur Form is used to describe the file syntaxes. Non-terminal symbols appear on the right-hand side of a production in angled brackets (<>); alternatives are shown using the metasympol '|'; optional text is enclosed in square brackets; the metasympol '*' indicates that the text it follows may occur any number of times (possibly zero).

A-1. *P-File*

A P-file contains a representation of a set of free group generators, followed by a list of words in the free group, expressed in ASCII format. The generators are denoted by giving them names, which consist of any upper- or lower-case letter followed by any number of primes (single quote marks). The words may be written as relators or as relations, with the special symbol "1" standing for the empty word when appearing alone on the right-hand-side of a relation.

The contents of a P-file have the following structure:

```
pfile ::= <bra> <gens> <bar> [<rels>] <ket>
gens  ::= <gen> [, <gen>]*
rels  ::= <rel> [, <rel>]*
words ::= <word> [, <word>]*
rel    ::= <word> [= <rhs>]
rhs    ::= 1 | <word>
word   ::= <term> [<term>]*
term   ::= <bterm> [<power>]
bterm  ::= <gen> | ( <word> ) | <sbra> <word> , <word> <sket>
gen    ::= <let> ['']*
power  ::= [+ | -] <int>
int    ::= <digit> [<digit>]*
digit  ::= 0|1|2|3|4|5|6|7|8|9
let    ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|
        A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
bra    ::= <
ket    ::= >
sbra   ::= [
sket   ::= ]
bar    ::= |
```

White space (blanks, tabs or newlines in any combination) may appear anywhere except inside an <int>. Comments, which begin with a '#' and terminate with the next newline character, are treated as one space. Styles for denoting free generators may be mixed, and no meaning is attached to the letter used or the number of single-quote marks attached.

A-1.1 Examples

The following are three ways of presenting the quaternion group of order 8.

Each will cause `readPfile()` to create identical memory models:

```
<x,y|xyxy-1,yxyx-1>
<b, a |
    bab = a,
    aba = b>
<H'', H' | (H'' H') H'' = ((H')), H'H''H'H''-1 = 1>
```

The P-file

`<S, T | S = TST, T = STS>`

although also a presentation for the same group, produces a different memory image when read, as one would expect.

A-2. C-File

The C-file contains an array of 32-bit unsigned integer values. In particular, it is used to store complete or partial coset-id multiplication tables.

The BNF specification of its structure is:

```
cfile ::= <magic> <ncols> <nrows> <entry>*
magic ::= C - ^G ^P
```

where '^G' represents octal 007, the ASCII audible bell character, and '^P' represents octal 020. Neither of these characters is likely to occur by accident in an ordinary text file.

The fields <ncols>, <nrows> and <entry> are all integral values represented in two's-complement form. Their sizes and meanings are as follows:

Field	Size	Meaning
ncols	16 bits, signed	no. of columns in the array
nrows	32 bits, unsigned	no. of rows in the array
entry	32 bits, unsigned	an array entry

The file is syntactically invalid unless there are at least

`<nrows> * <ncols>`

`<entry>` fields.

APPENDIX B

STANDARD MEMORY MODELS

The data files defined in Appendix A are represented in an executing process by complex data structures. These data structures are defined in concrete terms, and in terms of a 'C' binding.

B-0. Introduction

In order to increase the portability and reusability of the "mathematical" software in the primitive tools, each is implemented in a way which is independent of the details of the interchange file formats. This is achieved by the definition of standard memory models for common data structures. Each model is supported by a library of functions which provide portable file i/o, memory management and basic management capabilities for the data structures.

This Appendix describes the data models. C-language structure definitions are provided for each. Some features of the particular implementation employed in Chapter 2 are also noted. The library of support functions is defined in Appendix C.

B-1. Simple_Data_Types

Before describing the memory models, we need to define the elements from which they will be constructed. These are:

Type	Size	Range
gen_t	16-bit signed	-32768 - 32767
coset_t	32-bit unsigned	0 - 4294967296

All library functions here are written using these C-language types. This (nearly) guarantees portability of the data file formats, allowing them to be exchanged between sites possessing different implementations of the software. All integral values which are used purely internally to the software use standard scalar types, as appropriate.

B-1.1 *Generator-id*

A variable of type *gen_t* is used to hold objects such as:

- ⊕ an identifier for a group generator;
- ⊕ an identifier for the inverse of a group generator. This is represented as the negation of the identifier for the generator;
- ⊕ any quantity which enumerates or symbolises one of a set of objects which relate to the group generators.

The value 0 for a *gen_t* indicates the group identity. This value is never found in the words of a Presn.

On most machines the 'C' definition will be:

```
typedef short gen_t;
```

B-1.2 *Coset-id*

A variable of type `coset_t` is a symbolic identifier for a coset-id (see Section 1-3). It isn't a "coset", which is a concept requiring a data structure. Coset-id 1 is always taken to mean to mean the identity coset, wherever this notion is important or meaningful. Coset-id 0 is not defined, and hence is used to indicate "unknown value".

On most machines the 'C' definition

```
typedef long coset_t;
```

will suffice, but on a 32-bit machine we can be more precise:

```
typedef unsigned int coset_t;
```

B-2. *Words_in_a_Free_Group*

This Section defines the memory model used in this implementation to house the information contained in a P-file. Let G be a group, with a presentation $P = \langle X \mid R \rangle$, and let X^+ be the inverse closure of X .

B-2.1 *Motivations*

When we are manipulating a group presentation in software, the generators we see are merely place-holders. The values we use to identify them are immaterial. However, the user (ie. any agency which operates externally to a primitive tool) may have used names for the generators which carry meaning in some external context. Consequently we cannot just throw away the names used in a P-file; we must use them whenever we communicate information pertaining to P . Similarly, if we happen to be working on two presentations which depend

upon the same generating set, their memory models should depend upon the same set of generator names.

The most common activity within a primitive tool which uses a group presentation is to alter that presentation. In particular, the set of relators and the form of any particular relator are under constant review. Our memory model must therefore be managed dynamically, in order that these operations be implemented efficiently.

The practice of storing group words as a pair consisting of a word and an exponent [15] will not be followed. Each word is spelled out completely. It is not clear whether this is a benefit (in terms of simpler software) or a drawback (in terms of slower pattern matching). At some stage the Presn design could be augmented with an *exponent* field in Rel objects, together with a modification to all Presn-based functions, but at present it seems more important to discover simple techniques rather than possibly artificial data implementations.

B-2.2 Design

An object of type Presn is a data structure which ties together two lists, one of generator names and one of words or relators.

In turn, the generator names are held in an object of type Namelist. A generator-id (an object of type gen_t) can be applied to a Namelist in order to obtain the current name (text string) for that group generator. Namelists may be shared between several Presn objects.

The set R of group relators is held in an ordered list of structured objects

of type Rel. The principal sub-component of a Rel is a list of generator-ids, which constitutes a representation for the group relator itself. The particular ordering of relators, and the particular cyclic permutation used to represent each one, are the subject of canonicalisation (Section 1-1). In addition to the actual word, the Rel object contains a set of flags which can be used by any of the functions of the primitive library when dealing with the object. The flags are:

RELATOR this Rel represents a group relator. It may be cyclically permuted and inverted without changing its value;

WORD the Rel represents a word on the group G . It may not be cyclically permuted;

CANONICAL this relator is represented in its canonical form, for the present.

The object types which make up the composite type Presn have the following fields:

Presn:

Field	Type	Usage
rels	pointer to relator	list of relators defining G
nrels	16-bit unsigned integer	no. of relators in the list
names	pointer to Namelist	list of generator names

Namelist:

Field	Type	Usage
n	16-bit unsigned integer	no. of names in the list
v	array of strings	list of names of generators

Rel:

Field	Type	Usage
flags	array of 1-bit flags	information and status
len	16-bit unsigned integer	physical length of word
elen	16-bit unsigned integer	effective length under current length function
p	pointer to Letter	pointer to start of word
next	pointer to Rel	pointer to next relator in R

Letter:

Field	Type	Usage
id	generator-id	id of a member of X^+
next	pointer to Letter	next generator in relator
prev	pointer to Letter	previous generator in relator

B-2.3 C-Language_Binding

The Presn memory model is implemented here as a collection of C-language structured types. The definitions of these types are as follows (see elsewhere for definitions of the scalar types used):

Presn:

```
typedef struct {
    Rel *rels;
    gen_t nrels;
    Namelist *names;
} Presn;
```

Namelist:

```
typedef struct {
    gen_t n;
    char **v;
} Namelist;
```

Rel:

```
typedef struct stR {
    ul6bit flags;
    gen_t len;
    gen_t elen;
    Letter *p;
    struct stR *next;
} Rel;

#define RELATOR    0x01
#define WORD      0x02
#define CANONICAL 0x04
```

Letter:

```
typedef struct stL {
    gen_t id;
    struct stL *next;
    struct stL *prev;
} Letter;
```

B-2.4 *Implementation_Notes*

The following implementation notes apply to the use of objects of these types within C programs built on the GP library:

- ⊕ the list of names in the Namelist object is implemented as an array of strings. A linked list would be easier to edit dynamically, but the penalty incurred by having to scan for generator names would be too great;
- ⊕ when the name of a generator is required, it is found by indexing the names->v array using the absolute value of the gen_t. That is, the name of a generator is the same as that of its inverse;
- ⊕ when a generator is added to the list (eg. using one of the Tietze or Nielsen transformations, Section 1-2), the array names->v is copied into a new space of the right size. This does not entail copying the names themselves; the copy will still house a list of pointers to the original strings. This mechanism is not too costly, since this operation is fairly infrequent in most applications. It is necessitated by the need to use an array instead of a linked list;
- ⊕ the list of Letters in a Rel is doubly linked, to facilitate scanning and string matching. If the Rel is a RELATOR, the chain of pointers is cyclically closed in both directions.

B-3. *Coset-id_Multiplication_Table*

The action of the group generators X^+ on the set T of coset-ids (using the terminology introduced in Section 1-3) is defined by the partial function $f : T \times X^+ \rightarrow T \cup \{0\}$. This Section defines the memory model used to represent the set T and the action f . This model contains a *coset-id multiplication table*. When the set T is *sufficient* (as defined in Section 1-3) the model can be called a *coset table*, and this usage is often extended to cover the general (partial) case.

B-3.1 *Motivations*

Many implementations of coset-id methods represent the multiplication table as a simple array, in which the columns represent group generators and their inverses, and row i represents the set $\{j : f(i, g) = j, \text{ for all } g \in X^+\}$. Some values in the array are then overloaded with flag values during processing by various parts of an application process. In order that the overloaded entries are not misunderstood, the modules of the application must be tightly coupled: each must know the state of the others. We reject this organisation in favour of a data structure which allows us to easily note facts of interest about coset-ids, cosets or the table itself in a self-evident way, such that every value in every data structure always means exactly one thing.

B-3.2 *Design*

The data model centres around instances of an object type *Table*. Each *Table* represents the current state of a coset-id multiplication table. Each is self-contained, housing all information necessary to use or update the table.

A Table's primary sub-structure is a fixed-size array of objects of type `Coset`. Each array entry represents a coset-id, which can only be fully understood within the context of a Table. The offset i of a `Coset` object within the Table's array is its coset-id.

The object types `Table` and `Coset` have the following fields:

Table:

Field	Type	Usage
<code>ngens</code>	<code>gen_t</code>	no. of group generators
<code>rows</code>	array of <code>Cosets</code>	the "coset table"
<code>maxcosets</code>	long int	size of <code>rows</code> array
<code>nactive</code>	long int	current subgroup index
<code>hidef</code>	coset-id	current highest <code>Coset</code> in use
<code>freeptr</code>	coset-id	head of <code>Coset</code> free list

Coset:

Field	Type	Usage
<code>image</code>	array of <code>Coset</code> ids	coset image under gens
<code>flags</code>	array of 1-bit flags	information and status
<code>class</code>	pointer to <code>Class</code>	pointer to equivalence class

B-3.3 C-Language_Binding

The above data object types have been implemented here as C-language structures, defined as follows (see other Sections for definitions of the types

used):

```
typedef struct {
    gen_t ngens;
    Coset *rows;
    long int maxcosets;
    long int nactive;
    Coset *hidedf;
    Coset *freeptr;
} Table;

typedef struct COSET_STR {
    struct COSET_STR **image;
    short int flags;
    Class *class;
} Coset;

#define next(cp)      ((cp)->image[0])
```

B-3.4 *Implementation Notes*

The following implementation notes apply to the use of these structures within the modules discussed in Section 1-3:

- ⊕ the *rows* field of a *Table* is declared as a pointer. The actual array of *Coset* structures is created at run-time so that its size is not built into any tool (*tcs* and *tc* usually require very different table sizes). The memory space for it is allocated so as to house *maxCosets* structures, and a pointer to this memory is placed in *rows*;
- ⊕ we use direct pointers to *Cosets* instead of indexes into *rows*. This speeds up access by around 50 per-cent, and has reduced the run-time of *tc* by around 7 per-cent in most cases;
- ⊕ as with the *rows* field of a *Table*, the *image* field of a *Coset* is allocated dynamically when the *Coset* is used for the first time. This allows the size of X^+ to be specified *after* the *Table* has been created, and can

considerably reduce the actual amount of memory allocated during enumerations which require only a small number of coset-ids (in relation to `max-Cosets`);

- ⊕ the memory allocation scheme for Coset images places in *image* a pointer to a newly-allocated memory block. The block contains space for $2n + 1$ Coset pointers, where n is the cardinality of X^+ (twice the Table's *ngens* field). The pointer placed in *image* is the address of the central ($n+1$ th) element of this array. Since group generators (objects of type `gen_t`) in the Presn which contains the relators are represented as integers in the range $-n$ to $+n$, we can use array indexing on *image* without having to rewrite the Letters of the Presn, and without recourse to a function or data structure which converts generator-ids to "columns";
- ⊕ the value 0 is illegal for objects of type `gen_t`, so we have a free entry in the middle of every *image*. This is used to hold the free-list chain pointer used by the Coset allocation functions.

B-4. *Coset_Table_Spanning_Trees*

This Section defines the memory model which can be used to hold a coset table spanning tree, or indeed any collection of words organised according to common initial substrings. The tree can be called a *minimal spanning tree* for a coset table.

B-4.1 *Design*

A spanning tree represents a subgraph of the Schreier coset diagram for some subgroup H of a group G , such that the subgraph is a spanning tree. That is, the subgraph has no cycles, and is maximal in this respect.

The tree constructed here consists of nodes (structured data objects) and directed links (pointers to nodes). Each node in the tree is labelled with a group generator, in order to avoid the complexity of managing labelled links (edges of the Schreier diagram are labelled with generator-ids). This is unambiguous, since each node is the destination of exactly one link.

The degree of any node, while certain to lie between 0 and $2n$, where n is the number of group generators in use, is essentially unpredictable. It depends completely on the particular tree we have chosen. A fixed-size array, of length $2n$, to hold the links emanating from each node, will be expensive in memory space for such a sparse model. However, since each node can only be reached by one link from one other node (the structure represents a graph without cycles), we can collect nodes together into chains without ambiguity. Each node therefore points to a chain of nodes which are defined from it.

Each node of the tree is a structured data object of type *Node*, and has the following structure:

Field	Type	Usage
gen	gen_t	

		the generator which was used to define
		this coset
list	list of Nodes	the Nodes which can be reached from here
next	pointer to Node	the next Node in the current list

B-4.2 C-Language_Binding

The *Node* structure is defined for C programmers in the file `<gp/tree.h>`, as follows:

```
typedef struct DEFS {
    gen_t gen;
    struct DEFS *list;
    struct DEFS *next;
} Node;
```

APPENDIX C

THE PRIMITIVE TOOLS

A particular implementation of the primitive tool-set is defined. Unix-style manual pages are given for each of the tools. These manual pages describe little of the process model in the tool, concentrating instead on a description of run-time parameters and usage.

C-0. Introduction

This Appendix describes a particular implementation of the primitive tools described in Chapter 1. The approach adopted here takes the form of a complete set of Unix-style manual pages for the tools, and for the library functions upon which they depend.

C-0.1 Implementation_Details

The primitive tools are each written in C and use the Unix run-time support libraries and system calls. Each runs as a single Unix process. Those which accept run-time parameters extract them from the command line, using the standard Unix convention that arguments denoting algorithm options begin with a '-' character.

Much of the software in primitive tools, especially that which deals with group-theoretic data files, memory management or support for memory models, is common to several tools. These functions have been compiled separately and placed in a public library.

In fact, most of the group-theoretic algorithms and methods have also been

placed in this library. Many tools thus consist of a main routine which calls common functions from the library. An example of the utility of this approach is found in *tcs*, which computes a coset table for every subgroup of small index in a given group. All of the work of coset enumeration, coset table management and i/o and memory management is done by library functions which are also used in other tools, such as *tc*. The actual source code of *tcs* now consists only of a main routine which implements a stack-based backtracking mechanism.

C-0.2 *Tool-Set_Installation*

The software comprising a primitive tool can be found in 3 places:

- ⊕ the source code of the functions which are specific to the tool itself. This usually consists simply of a main routine, implemented in C, which co-ordinates the actions of library functions which read in data, act upon it and then write it out again. The tool is compiled into a program which is resident in a public pool (/usr/gbin) and which runs as a single Unix process;
- ⊕ a library of functions, each of which is implemented in C. The library (/usr/lib/libgp.a) consists of functions which manage and define important data structures, perform memory management and i/o, perform group-theoretic calculations, etc.;
- ⊕ a collection of data types and flag values. These are defined for C, in header files found in the /usr/include/gp directory.

NAME

abel - abelianise a group presentation

SYNOPSIS

abel

DESCRIPTION

abel reads a P-file, containing a presentation for the group G , from standard input. It writes a P-file to standard output which presents the group G / G' . This is done by appending relators to the input P-file, each of which is a commutator of two generators of G .

EXAMPLE

```
$ cat a4
<a, b, c | abac = 1, bcba = 1, cacb = 1>
$ abel < a4
<a, b, c | abac = 1, bcba = 1, cacb = 1, [a, b], [b, c], [c, a]>
```

NAME

canon - put a presentation into canonical form

SYNOPSIS

canon [-P file] [-l func] [-g afr] [-i bgrR] [-r lmr] [-o gir]

DESCRIPTION

The primitive tool *canon* edits a group presentation into one of several "canonical forms", writing the resulting P-file to standard output. If the argument *-P P-file* is present, *P-file* is read for the input group presentation. Otherwise, or if *P-file* is "-", standard input is read. The option *-l* allows the user to specify the length function which is to be used when determining the effective length of a word. The option argument must be one of the key-letters:

b the *blen* function is used;

f the *len* function is used.

No support for weighted generators is provided at present.

The relators are freely and cyclically reduced, after which the following algorithm is performed:

a. *Generator Processing*

If the *-g* option is specified, the group generators are sorted.

The option argument must be one of these key-letters:

- r the generators are placed into a random order;
- a the generators are sorted so that their names fall into ascending "alphabetical order". The ASCII collating sequence is used;
- f the generators are sorted into ascending frequency order.

b. *Presentation Pre-processing*

If the *-i* option is present, the presentation is now globally altered prior to canonicalising the relators. The value of the option argument is a string consisting of any combination of the following key-letters:

- g the order of the generators is reversed. This option is performed first, if selected;
- r the relators are each cyclically permuted so as to present the minimal word according to the current ordering of generators;
- R the relators are each cyclically permuted by random amounts;
- b the relators are each rotated so as to present the minimal effective word length under the BASE length function.

c. *Relator Processing*

If the *-r* option is specified, the group relators are organised into a canonical form. The precise form is given in the option argument, which must consist of exactly one of the following key-letters:

- m relators are cyclically permuted and sorted so that longest common substrings are brought to the front of nearby words. Any relator which is found to be identical with another is removed;
- r the sequence of relators is randomised;
- l the relators are sorted into ascending order of effective length (according to the currently active length function). Relators of equal length are ordered using the current ordering of generators. Any relator which is found to be identical with another is removed.

d. *Presentation Post-processing*

If the `-o` option is specified, the presentation is now post-processed. The option argument is a string consisting of any combination of the following key-letters:

- r the order of the relators is reversed;
- g the order of the generators is reversed;
- i every relator is inverted.

EXAMPLE

The usage

```
canon < p
```

freely and cyclically reduces the file *p*, while

```
canon -P p -ir -rl
```

applies to p the canonical form for group presentations defined in [20,21].

SEE ALSO

canon(3), length(3), presn(3)

NAME

`def` - group deficiency

SYNOPSIS

`def [-rgd] [names . . .]`

DESCRIPTION

`def` outputs the number of relators, number of generators and the deficiency of the presentations in the named P-files, or in the standard input if no *names* appear.

The options *r*, *g* and *d* may be used in any combination to specify that a subset of relators, generators and deficiency is to be reported. The default is `-rgd`.

When *names* are specified on the command line, they will be printed along with the counts. Files read by `def` must be P-files.

SEE ALSO

`pfile(3)`.

NAME

dindex - print the derived index of a group

SYNOPSIS

dindex

DESCRIPTION

dindex reads a P-file, containing a presentation for a group G , from standard input. It then writes the derived index $[G : G']$ to standard output. An output value of 0 indicates that the derived factor has infinite order.

The derived index is calculated as follows. If the deficiency of the input presentation is zero (equal numbers of generators and relators), the output value is the determinant of the presentation's relation matrix. However, if the deficiency is non-zero (ie. the relation matrix is not square) the output value is the h.c.f. of the determinants of all maximal square matrices.

EXAMPLE

```
$ cat a4
<a, b, c | abac = 1, bcba = 1, cacb = 1>
$ dindex < a4
3
$
```

NAME

`extend` - extend a coset table

SYNOPSIS

`extend [-C] -P pfile [-c cid]`

DESCRIPTION

`extend` reads a C-file from standard input, extends it and writes a new C-file to standard output. It therefore acts as a filter on coset tables, extending them so that the products of the words in *pfile* with coset-id *cid* are all fully defined. If *cid* is not specified, the default value of 1 is used.

Each word in *pfile* is applied in turn to coset-id *cid*. If the product is not defined (zero), new coset-ids are defined to complete the product.

If the `-C` argument is specified, `extend` does not read a C-file from standard input. Instead, an "empty coset table" is created, extension therefore starting from scratch.

EXAMPLES

The following simple shell script implements part of a "feed-back loop". A coset table is extended, with the definition of new coset-ids being driven by a minimal spanning tree from another coset table. The driving table is specified as the first argument to the shell script.

```
# Coset table feed-back filter.  
st -b < $1 > /tmp/st.$$  
extend -P /tmp/st.$$ -c 1
```

NAME

`gsh` - a group theory shell

SYNOPSIS

`gsh`

DESCRIPTION

`gsh` is a simple shell-like interface to the other tools in the primitive tool-set. It keeps note of a *current group*, given by its presentation, and a *current subgroup*, given by a set of generating words. Command-line parameters to the primitive tools are determined by a collection of environment variables.

`gsh` supports the following built-in commands:

`group` allows the user to type in a set of defining relations for a group. The resulting presentation becomes the 'current group'. The current subgroup, and any attendant information, is deleted;

`subgp` allows the user to type in a set of defining words for a subgroup of the current group. The resulting presentation becomes the 'current subgroup'. Any information relating to the previous 'current subgroup' is deleted;

`enum` attempts to enumerate the cosets of the current subgroup in the current group, producing a 'current coset table'. The actual command used is

tc \$TC

unless there is no current subgroup, in which case

tc -C \$TC

is used. The user can alter this command's effect by changing the value of the run-time variable \$TC;

rs applies the Reidemeister-Schreier method to the current group presentation and the current coset table, to produce a presentation for the current subgroup. The user's choice of coset table spanning tree is given in the run-time variable SST, which is passed on the command-line to the primitive tool *st*;

tt invokes the primitive tool *tt* on the current group presentation;

canon canonicalises the current group's presentation, using the command-line arguments set by the user in \$CANON;

recurse the current subgroup becomes the current group. If there is no current subgroup presentation, this is calculated by simulating the "rs" command (and "enum" if necessary). All information pertaining to the parent group is deleted.

Any other command is evaluated by *gsh*'s underlying Bourne shell (*sh*(1)).

Default values for the run-time variables used here are:

Variable	Default
TC	-t
ST	-b
CANON	-ir -rl

Note that the "enum" command produces no coset table until the user specifically asks for it. This saves time and space while a suitable subgroup is being sought.

NAME

`rewrite` - rewrite a presentation

SYNOPSIS

`rewrite -C cfile -P pfile`

DESCRIPTION

`rewrite` applies Reidemeister rewriting to the presentation given in the P-file supplied to standard input, producing a subgroup presentation on standard output. The subgroup is specified by its coset table, which is found in *cfile*. The Schreier transversal, to be used in constructing a set of Schreier generators for the subgroup, is given in *pfile*.

NAME

st - construct a coset table spanning tree

SYNOPSIS

st -P file [-b | -d] [-c]

DESCRIPTION

st constructs a Schreier transversal for a complete coset table. The table is read from standard input, which must be in the form of a C-file. The transversal is written to standard output in the form of a P-file containing only the maximal representatives.

The option -P *file* supplies a P-file which will be used to provide a Namelist for the output P-file, and possibly an ordering for generator visits while constructing the coset table spanning tree.

One of the options -b or -d must be present. Each selects method to be used in the construction of the tree:

-b a breadth-first scan of the coset table is produced. This tends to lead to "bushy" trees;

-d a depth-first scan of the coset table is performed. This tends to lead to "skinny" trees.

If the -c option is given, the words in the P-file *file* are used to order the visiting of generators during the construction of the coset table spanning tree.

EXAMPLE

Suppose *cfile* is a file which contains the coset table

	-x	x	-y	y
1:	2	8	4	7
2:	6	1	3	5
3:	4	7	8	2
4:	5	3	6	1
5:	7	4	2	8
6:	8	2	7	4
7:	3	5	1	6
8:	1	6	5	3

for the cosets of $\langle 1 \rangle$ in $Q = \langle x, y \mid xyx = y, yxy = x \rangle$, and suppose that *pfile* contains $\langle x, y \mid y, x \rangle$. Then the command

```
st -P pfile -d < cfile
```

produces the output $\langle x, y \mid x-3y-1x-3 \rangle$, whereas

```
st -P pfile -d -c < cfile
```

produces $\langle x, y \mid y3xy3 \rangle$.

NAME

tc - Todd-Coxeter coset-id equivalence filter

SYNOPSIS

```
tc [-s] [-t] [-C] [-P filename]
    [-f fillers] [-m maxCosets] [-c maxCoincs]
    [-d maxDefs] [-p maxPasses]
```

DESCRIPTION

tc reads a list of coset-ids from standard input, filters out those which are redundant (ie. equivalent to others on the list) according to some context, and writes the resulting list of coset-ids to standard output. Both coset-id lists are represented as *partial coset tables* over a set of free group generators. Both therefore have the structure of C-files. The outgoing C-file is compacted prior to being written to standard output.

According to options on the command-line, tc may attempt to output a *complete coset table*, in which the action of every free group generator is defined on every coset-id in the list. This usually involves augmenting the original list of coset-ids.

tc uses the Todd-Coxeter method to determine the equivalence (or otherwise) of pairs of coset-ids in the list. This in turn depends upon a set of words on the free group generators, which define a context (ie. a group) within which the testing occurs. These words are supplied in a

P-file, via the `-P` command-line parameter. Thus, if the P-file is given and the other parameters to `tc` are such that new coset-ids may be introduced into the list, `tc` will effectively perform Todd-Coxeter coset enumeration, enumerating the cosets of some subgroup of the group presented in the P-file. The subgroup in question is determined by the C-file on standard input.

Parameters to `tc` are:

- `-t` If this parameter is specified, `tc` will not write the resulting coset table to standard output;
- `-s` This parameter requests that closure checking be terminated when the first coset-id in the list is *R-closed*, ie. when the application of every word W to coset l yields $l.W = l$ with no contradictions;
- `-C` The presence of `-C` on the command-line tells `tc` that there is no C-file on standard input, ie. that the enumeration (if that is the user's wish) is to be performed over the trivial subgroup. If this flag is not specified, the process will block reading stdin;
- `-P` The file *filename* must contain a P-file on the same number of free generators as the incoming coset table. The relators in the P-file are used for the algorithm's closure context.

The following parameters affect the automatic introduction of coset-ids into the list:

- `-m n` places a physical limit on the number of coset-ids allowed in the

list at any time;

-d *n* restricts the internal definitions engine to make at most *n* coset-id definitions at each request;

-c *n* causes a request for coset-id definitions to terminate when *n* real coincidences have been placed in the queue;

-f *s* causes each pass of each request for coset-id definitions to use the strategies listed in the string *s* in turn. The strategies available are:

C fill one row of every relator table. That is, R-close one coset-id;

G fill one column of the coset table;

I fill one row of the coset table. That is, I-close one coset-id;

R define the application of every coset-id with one particular relator.

One application of each of these strategies from *s* is a defining *pass*;

-p *n* restricts the definitions engine to at most *n* passes in the service of any one request.

The default settings of these parameters are:

Parameter	Default
-s	off
-t	off
-C	false
-P	none
-c	10000
-d	1000
-m	10000
-p	10000
-f	C

DIAGNOSTICS

The library functions upon which this tool is built can output trace information in various subjects. Typical use of *tc*, for coset enumeration, uses a tracefile such as

```
stats = tc_stats close_stats coinc_stats defn_stats;
info = tc_info defn_info;
```

```
info : 3;
stats : 6;
```

with the environment variable TRACE set to 3 or 6.

EXAMPLES

Suppose that the file *group.p* contains

```
<a, b | a2b3 = 1, (ab)5 = 1>
```

and that *./trc* (our current tracefile) is set up as suggested above.

Using the Unix Bourne shell (*/bin/sh*) we could have the following interactive session (in every case, we use *-t* so as not to confuse the issue by having to save the output coset table somewhere):

First, with the global tracing level set at 3:

```
$ TRACE=3
$ tc -t -C -P group.p
```

```
start coinc left ndefs
  1      0      1 1000
1001  646  355 1000
1355  798  557 1000
1557  957  600    0
  600    0  600
```

```
Total cosets defined: 3000
Index = 600
```

Now, increasing the tracing level to 6, we obtain more information from the definitions engine:

```
$ TRACE=6
$ tc -t -C -P group.p
```

```
start coinc left npass Cdefs Gdefs Idefs Rdefs ndefs qpush
  1      0      1  117 1000    0    0    0 1000   36
1001  646  355  143 1000    0    0    0 1000   36
1355  798  557  137 1000    0    0    0 1000   36
1557  957  600    1    0    0    0    0    0    0
  600    0  600
```

```
Total cosets defined: 3000
Index = 600
```

Using a popular definitions strategy:

```
$ tc -t -C -P group.p -fCI
```

```
start coinc left npass Cdefs Gdefs Idefs Rdefs ndefs qpush
  1      0      1  108  798    0  202    0 1000   54
1001  672  329  121  784    0  216    0 1000   43
1329  749  580  120  781    0  219    0 1000   56
1580  980  600
```

```
Total cosets defined: 3000
Index = 600
```

Restricting the number of coincidences we allow to be queued before applying them to the list of coset-ids:

```
$ tc -t -C -P group.p -fCI -c20
```

start	coinc	left	npass	Cdefs	Gdefs	Idefs	Rdefs	ndefs	qpush
1	0	1	33	252	0	62	0	314	20
315	120	195	32	188	0	52	0	240	20
435	120	315	45	266	0	72	0	338	20
653	186	467	63	342	0	107	0	449	20
916	324	592	42	254	0	79	0	333	20
925	325	600	2	0	0	0	0	0	0
600	0	600							

Total cosets defined: 1674
Index = 600

Finally, if the file sg.p contains

$$\langle a, b \mid a \rangle$$

we can enumerate the cosets of the subgroup $\langle a \rangle$ of the above group thus:

```
$ tc -C -P sg.p -m10 | tc -t -P group.p
```

start	coinc	left	npass	Cdefs	Gdefs	Idefs	Rdefs	ndefs	qpush
1	0	1							

Total cosets defined: 0
Index = 1

start	coinc	left	npass	Cdefs	Gdefs	Idefs	Rdefs	ndefs	qpush
1	0	1	118	1000	0	0	0	1000	40
1001	971	30							

Total cosets defined: 1000
Index = 30

showing that $\langle a \rangle$ has index 30.

NAME

`tcs` - find all subgroups of low index

SYNOPSIS

```
tcs [-t] [-C] [-P P-file]
    [-f fillers] [-m maxCosets] [-c maxCoincs]
    [-d maxDefs] [-p maxPasses]
    [-F n] [-i index] [-n name]
```

DESCRIPTION

The primitive tool `tcs` creates a coset table for every subgroup of a given group, such that the subgroup's index lies below a certain threshold. The method used is Dietze & Schaps', and consists of a series of coset enumerations and forced coset-id equivalences. The tool is built from the same libraries as is `tc`, so most of the arguments to `tcs` are also borrowed from `tc`. Those which need a little explanation are:

- `-C` as with `tc`, the standard input to `tcs` can contain a coset table which existentially defines a subgroup of the group presented in the *P-file* argument. In that case, `tcs` will list large subgroups of that subgroup. If the `-C` argument is specified, `tcs` will not expect a C-file on standard input;
- `-t` prevents `tcs` from creating a C-file describing each subgroup found. This parameter can be useful with tracing enabled, to determine the kind of results one may expect;

- F *n* causes *tcs* to exit after the *n*th C-file has been created;
- i *index* causes *tcs* to create a C-file only for those subgroups whose index is equal to *index*. If this argument is not used, every subgroup whose index is less than or equal to *maxCosets* will be output;
- n *name* causes the output C-files to have the names *name.n*, where *n* is some integer value.

DEFAULT VALUES

The default values for the coset enumeration parameters are very different to those of *tc*. This is primarily because *tcs* usually has very many more partial coset tables active at any given time, so that memory space can be restricted. The defaults for the parameters to *tcs* are:

Parameter	Default
maxCosets	24
maxPasses	100
maxDefs	100
maxCoincs	100
fillers	C
name	"cfile"

NAME

`tt` - interactive editor for group presentations

SYNOPSIS

`tt [file]`

DESCRIPTION

`tt` is a simple editor for group presentations, modelled on the Unix tool `ed(1)` but providing instead commands which embody Tietze and Nielsen transformations. Parameter *file* must contain a P-file, if it is supplied.

`tt` reads commands from standard input, until end-of-file is reached. It then quits. The command '?' lists the available commands.

FILES

The following files are read, in order, before `tt` starts to interact with the user but after the presentation has been loaded:

`/usr/lib/ttrc`

`$HOME/.ttrc`

`./.ttrc`

Each is read using the 'source' built-in command, and is expected to contain valid `tt` commands.

SEE ALSO

pfile(3), presn(3)

NAME

`canon` - put a presentation into canonical form

SYNOPSIS

```
#include <gp/presn.h>
#include <gp/canon.h>

canon(pres, gens, pre, rels, post)
Presn *pres;
int gens, pre, rels, post;
```

DESCRIPTION

The function `canon()` edits the group presentation `pres` into one of several "canonical forms". The relators in `pres` are freely and cyclically reduced, after which the following algorithm is performed:

a. *Generator Processing*

The group generators are sorted. `gens` must contain one of the following values:

0 the generators are not touched;

GenRandom the generators are placed into a random order;

GenAlpha the generators are sorted so that their names fall into ascending "alphabetical order". The ASCII collating sequence is used;

GenFreq the generators are sorted into ascending frequency order.

b. *Presentation Pre-processing*

The presentation *pres* may now be globally altered prior to canonicalising the relators. The value of *pre* contains the bit-wise "or" of any of the following symbols:

PreGrev the order of the generators is reversed. This option is performed first, if selected;

PreRotate the relators are each cyclically permuted so as to present the minimal word according to the current ordering of generators;

PreRandom the relators are each cyclically permuted by random amounts;

PreBase the relators are each rotated so as to present the minimal effective word length under the *BASE* length function.

c. *Relator Processing*

The group relators are organised into a canonical form. The precise form is given in *rels*, using exactly one of the following options:

RelMatch relators are cyclically permuted and sorted so that longest common substrings are brought to the front of nearby words. Any relator which is found to be identical with another is removed;

RelRandom the sequence of relators is randomised;

RelLength the relators are sorted into ascending order of effective length (according to the currently active length function). Relators of equal length are ordered using the current ordering of generators. Any relator which is found to be identical with another is removed.

d. *Presentation Post-processing*

The presentation *pres* is now post-processed. The parameter *post* may contain the bit-wise "or" of any of the following symbols:

PostRrev the order of the relators is reversed;

PostGrev the order of the generators is reversed;

PostInvert every relator is inverted.

RETURN VALUE

canon() returns 0 on successful completion, but -1 if any supplied parameter contained an invalid flag.

FILES

/usr/include/gp/canon.h defines the manifests which communicate the algorithm selection to the function.

EXAMPLE

The function call

`canon(p, 0, PreRotate, RelLength, 0)`

applies to p the canonical form for group presentations defined in [20,21]. The usage

`canon(p, 0, 0, 0, PostInvert)`

replaces every relator in p by its inverse.

SEE ALSO

`canon(1), length(3), presn(3)`

NAME

length - effective length of words in groups

SYNOPSIS

```
#include <gp/length.h>

gen_t elength(w, len, wt)
Letter *w;
int len;
int *wt;
```

DESCRIPTION

elength() returns the current effective length of the word *w* upto physical length *len* Letters. The current length function is denoted by the external variable *lenFunc*, which is declared in *<gp/length.h>* as follows:

```
typedef enum {PHYSICAL, BASE, WEIGHTED} LenFunc;

extern LenFunc lenFunc;
```

The initial value of *lenFunc* will be *PHYSICAL* until altered by user software.

NAME

presn - Presn support functions

SYNOPSIS

```
#include <gp/pfile.h>

int addName(nlist, name)
Namelist *nlist;
char *name;

int addRel(p, r, atEnd)
Presn *p;
Rel *r;
int atEnd;

Letter *commutator(v, w)
Letter *v, *w;

Letter *copyWord(w)
Letter *w;

Generator findName(name, nlist)
char *name;
Namelist *nlist;

Letter *invertWord(w)
Letter *w;
```

DESCRIPTION

These functions perform most of the simple access and update tasks common to the use of the data structures defined in `<gp/presn.h>`, generically known as group presentations.

`addName()` appends the string `name` to the `Namelist nlist`, updating the internal length counter if successful. Note that, although memory allocation and de-allocation are likely to occur within this function, the address of the `Namelist` header structure itself will not change.

addName() returns 0 upon success, -1 if an error occurred.

addRel() augments the presentation data structure pointed to by *p* with the relator structure *r*. If the parameter *atEnd* is non-zero (ie. "true") the relator is appended to the original list; otherwise it becomes the first relator in the chain.

commutator() creates the word $@v \text{ sup } -1 w \text{ sup } -1 v w@$, returning a pointer to the start of the resulting chain of Letters. The new word is created entirely from scratch; the words *v* and *w* are untouched by the operation. In order for this function to work, both *v* and *w* must be terminated by null Letter pointers.

copyWord() uses *newLetter()* to create a complete copy of the (null-terminated) word *w*.

findName() looks for the free generator called *name* in the Namelist *nlist*. If the generator is found, its index in the list is returned. Otherwise *findName()* returns -1.

invertWord() inverts the (null-terminated) word *w*. The inversion is done *in situ*. That is, *w* itself is edited: the Letter pointers are reversed and the *id* fields of the constituent Letters are negated (thereby inverting the generator represented at each position). The function returns a pointer to the start of the inverted word, ie. to what was the last Letter of the original *w*.

SEE ALSO

presn(3)

GP Library

presn(3)

canon(1), rPfile(3), trace(3), wPfile(3)

NAME

readPfile - P-file to Presn syntax conversion

SYNOPSIS

```
#include <gp/pfile.h>

Namelist *getNlist(s)
char *s;

Rel *getRel(s, nlist)
char **s;
Namelist *nlist;

Letter *getWord(s, namelist)
char **s;
Namelist *namelist;

Presn *readPfile(fd)
int fd;
```

DESCRIPTION

Interactive tools may wish to allow their users to add to the generators or relators of a P-file. The functions `getNlist()`, `getWord()` and `getRel()` are provided to allow input of these objects in the tool-set's common syntax.

`getNlist()` parses the string `s`, constructing from it a `Namelist`. The string must have the following structure:

```
gens ::= gen [, <gen>]* NUL
gen  ::= LETTER [']*
```

where `LETTER` may be any upper or lower-case letter and `NUL` is the ASCII zero character which terminates C-language strings. White space may occur anywhere in the string, and is ignored.

getRel() parses the NUL-terminated string pointed to by **s*, constructing a Rel and a chain of Letters if successful. The string must have the following syntax:

```
rel ::= <word> [= <word>] NUL
```

The function *getWord()* (see below) is used to parse the <word> non-terminals. The Namelist *nlist* is used to determine the validity of the generators found. *getRel()* returns a pointer to the newly-created Rel structure, which has all its fields correctly initialised, or NULL upon error. The value of *s* is updated so that **s* points to the place in the string at which parsing terminated (or failed).

getWord() parses the string **s* to build a chain of letters representing its contents, which must be of the form:

```
word ::= <term> [<term>]*
term  ::= <bterm> [<power>]
bterm ::= <gen> | ( <word> ) | <sbra> <word> , <word> <sket>
power ::= [+ | -] <int>
int   ::= <digit> [<digit>]*
sbra  ::= [
sket  ::= ]
```

White space may occur anywhere in the string (except inside an <int>), and is ignored. The Namelist *nlist* is used to determine the validity of the generators found. The pointer **s* is updated to point to the end of the word parsed or to the character at which an error occurred. *getWord()* returns NULL upon error.

readPfile() reads a specification from open file *fd*, creates a Presn structure and any attendant substructures required and returns a pointer to the new memory model.

DIAGNOSTICS

The above functions are all implemented as recursive descent compilers [17], and all use the error reporting and recovery method developed by Turner for his SASL compiler [47,48,49]. In all cases, syntax errors found during the parsing are treated as if the error had not occurred, wherever that is possible. Error messages are output by these functions in level 3 of the *presn_info* trace subject.

SEE ALSO

canon(3), trace(3), writePfile(3)

NAME

trace - run-time diagnostics library

SYNOPSIS

```
#include "trace.h"

trace(subj, level, format, value)
int subj;
int level;
char *format;

trcWord(subj, level, w, len, nlist)
int subj;
int level;
Letter *w;
int len;
Namelist *nlist;

trcTable(subj, level, table)
int subj;
int level;
Table *table;

trcCoset(subj, level, cp, table)
int subj;
int level;
Coset *cp;
Table *table;

trcQueue(subj, level, queue, table)
int subj;
int level;
Coinc *queue;
Table *table;

trcctl()
```

DESCRIPTION

The functions listed here provide trace support for the rest of the GP library. In each case, the function call represents a trace statement, which is actually printed only if the subject *subj* is enabled up to *level*

at least, and the environment variable TRACE is also enabled to this level.

`trace()` uses `fprintf()` to print the single value on the current trace output file, using the string *format* to denote the value's type, size and output context.

Each of the other functions is specific to the GP application library. All except `trcWord()` output a newline character after printing the particular object specified.

ENVIRONMENT

TRACE contains an integer, which represents the overall enablement level of the tracing system. No function may output diagnostics at any higher level than this. If TRACE is undefined or null, it is assumed to have the value zero.

TRACEFILE contains the name of the file to which trace statements will be written. The special names *stdout* and *stderr* are recognised. If TRACEFILE is undefined or is null, the value "stderr" is assumed.

Each of these variables is read at the time the first call to one of the above functions is encountered. Thereafter, either value may be altered via the `trcctl()` primitive.

FILES

When the first call to a tracing function is encountered during the running of a process, the trace system initialises itself before servicing

that function. Part of this initialisation concerns the global trace level and the output file, as above. The remainder concerns the reading of *tracefiles*, which create a configuration of subjects, meta-subjects and enablement levels. Subsequently to this initialisation, the *trcctl()* primitive can be used to alter the levels attached to the subjects and meta-subjects.

Three files are read. They each have the same structure, and need not exist at all. The files are:

```

/usr/lib/trc    system-wide macro definitions and level configuration
$HOME/.trc     personal macro definitions etc.
./trc          definitions and configuration on a per-job basis

```

The files have the following syntax:

```

file      ::= <subj> <command> ;
command   ::= : <INTEGER> | = <names>
names     ::= <subj> [<subj>]*
subj      ::= <IDENTIFIER>

```

EXAMPLES

The examples listed under TC(1) used the single tracefile

```

stats = tc_stats close_stats coinc_stats defn_stats;
info = tc_info defn_info;

info : 3;
stats : 6;

```

with the environment variables set to

TRACE(3)

GP Library

TRACE(3)

TRACEFILE=stderr
TRACE=3

or

TRACEFILE=stderr
TRACE=6

NAME

writePfile - Presn to P-file syntax conversion

SYNOPSIS

```
#include <gp/pfile.h>

putGen(s, id, power, nlist)
char *s;
Generator id;
int power;
Namelist *nlist;

putWord(s, w, len, nlist)
char *s;
Letter *w;
int len;
Namelist *nlist;

int writePfile(fd, p)
int fd;
Presn *p;
```

DESCRIPTION

P-files are free-format text files, allowing both input and output functions complete freedom over the placing of new-line characters and white space. Any tool wishing to exert a high degree of control over its user interface may wish to control the writing of a P-file to the user's terminal in some specific way. For such tools, `putWord()` and `putGen()` produce simple textual representations of words and generators in accordance with the standard P-file syntax; these can then be combined to create formatted P-files. The function `writePfile()` implements just one of the many alternatives here.

`putGen()` writes the name of the free generator with identifier `id` to the

character buffer *s*, followed by an integer decimal representation of the power of the generator.

putWord() uses *putGen()* repeatedly to construct an ASCII representation of the word *w* (whose length is *len*) in the character buffer *s*. The length parameter allows the scanning of cyclically-closed words. Alternatively, if the word is NULL-terminated, a *len* of 0 will cause *putWord()* to continue to the end of the word. Identical Letters are telescoped, putting an integer after the generator's name to signify a power of that generator.

writePfile() writes the Presn structure pointed to by *p* onto the open file *fd*, in such a way that the resulting P-file would cause *readPfile()* to create an identical memory model again. Generators are written out using the names in the *p->names* Namelist. Strings consisting of repetitions of a single generator or its inverse are collected together, being represented as a power of that generator.

BUGS

putWord() has no intelligence. The word representation written to the character buffer is not structured into commutators or powers of words. Consequently the string can be more verbose than is absolutely necessary.

SEE ALSO

canon(3), *presn(3)*, *readPfile(3)*

APPENDIX D

PRIMITIVE TOOL MODELS

Models are given for the primitive tools which are described in Chapter 1 and specified in Appendix C. The models are pictured as atomic elements of the process diagrams used in Chapter 2.

D-0. *Introduction*

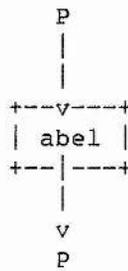
In Chapter 2, compositions of the primitive tools are presented in a diagrammatic form which is independent of the composition language. The elements of these diagrams are presented in this Appendix. In each diagram:

- ⊕ data flows down the page;
- ⊕ each box represents a primitive tool or a high-level utility built from primitive tools and depicted elsewhere;
- ⊕ the standard input of each tool or utility enters at the top;
- ⊕ the standard output of each tool leaves from the bottom of its box in the diagram;
- ⊕ auxiliary files used by a tool enter from the side, implying that they are used by the filter to affect the data flowing through from top to bottom;
- ⊕ run-time parameters to the tools are not shown. These will be described where necessary.

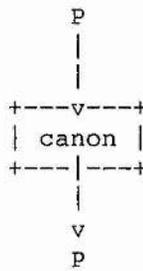
D-1. *PRIMITIVE_TOOL_MODELS*

The process interface models for the primitive tools specified in Appendix C are as follows:

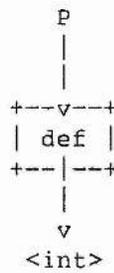
abel:



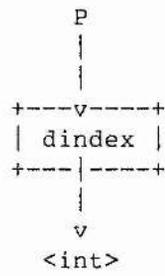
canon:



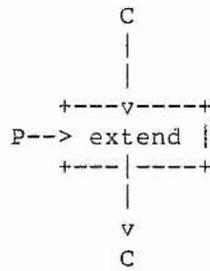
def:



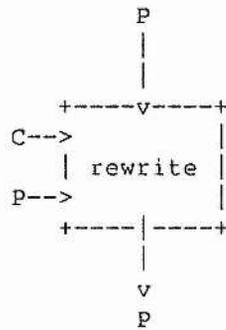
dindex:



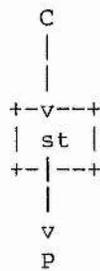
extend:



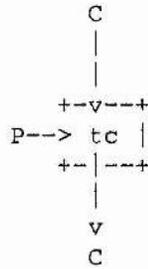
rewrite:



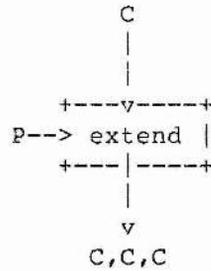
st:



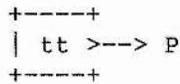
tc:



tcs:



tt:



Clearly, *tt* is the only primitive tool which isn't a filter. This is a direct consequence of the design choice that *tt* should emulate the simple Unix line editor *ed(1)* in philosophy.

However, it is quite straightforward to implement shell scripts around *tt* which perform specialised filtering tasks. For instance, *canon* could be implemented in the Bourne shell *sh(1)* as follows:

```
# Canonicalisation shell, using tt.  
TMP=/tmp/canon.$$
```

```
cat > $TMP  
tt $TMP <<-ENDFILE  
    canon $*  
    w  
    ENDFILE  
cat $TMP  
rm -f $TMP
```

APPENDIX E

REFERENCES

- [1] Arrell, Manrai, Warboys: *A procedure for obtaining simplified defining relations for a subgroup*, in *Groups - St.Andrews 1981*, (LMS 71) ed. Campbell & Robertson.
- [2] Arrell, Robertson: *A modified Todd-Coxeter algorithm*, in *Computational Group Theory* (ed. Atkinson) Academic Press (1984).
- [3] Atkinson, Hassan, Thorne: *Group theory on a micro-computer*, in *Computational Group Theory* (ed. Atkinson) Academic Press (1984).
- [4] Beetham: *Saving space in coset enumeration*, in *Computational Group Theory* (ed. Atkinson) Academic Press (1984).
- [5] Beetham, Campbell: *A note on the Todd-Coxeter coset enumeration algorithm*, Proc. Roy. Edin. Math. Soc. 20 (1976).
- [6] Benson, Mendelsohn: *A calculus for a certain class of word problems in groups*, J. Combinatorial Th. 1 (1966), 202-208.
- [7] Campbell: *Some examples using coset enumeration*, in *Computational Problems in Abstract Algebra*, edited by J.Leech, Pergamon (1970).
- [8] Campbell: *Computational techniques and the structure of groups in a certain class*, Proc. ACM Symposium on Symbolic and Algebraic Comp. (1976). 19 (1975), 297-305.

- [9] Campbell, Robertson: *Remarks on a class of 2-generator groups of deficiency zero*, J. Austral. Math. Soc 19 (1975), 297-305.
- [10] Campbell, Robertson: *Applications of the Todd-Coxeter algorithm to generalised Fibonacci groups*, Proc. Roy. Soc. Edin., 73A (1974/75), 163-166.
- [11] Campbell, Robertson: *Deficiency zero groups involving Fibonacci and Lucas numbers*, Proc. Roy. Soc. Edinburgh, 81A (1978), 273-286.
- [12] Campbell, Robertson: *Some problems in group presentations*, J. Korean Math. Soc.
- [13] Cannon: *A Language for Group Theory*, University of Sydney.
- [14] Cannon: *An introduction to the group theory language CAYLEY*, in *Computational Group Theory* (ed. Atkinson) Academic Press (1984).
- [15] Cannon, Dimino, Havas, Watson: *Implementation and analysis of the Todd-Coxeter algorithm*, Math. Comp. 27 (1973), 463-490.
- [16] Coxeter, Moser: *Generators and Relations for Discrete Groups*, 3rd edn, Springer Verlag (1972).
- [17] Davie, Morrison: *Recursive Descent Compiling*, Ellis Horwood, 1981.
- [18] Dietze, Schaps: *Determining subgroups of a given finite index in a finitely presented group*, Canad. J. Math., XXVI (1974) 769-782.
- [19] Greiner, Lenat: *RLI: A Representation Language Language*, Proc. 1st Annual Meeting American Assoc. for Artificial Intelligence, Stanford

(1980).

- [20] Havas: *A Reidemeister-Schreier program*, Proc. 2nd Int. Conf. Theory of Groups (Canberra 1973), *Lecture Notes in Mathematics* 372 Springer-Verlag, Berlin, 347-356.
- [21] Havas, Kenne, Richardson, Robertson: *A Tietze transformation program*, in *Computational Group Theory* (ed. Atkinson) Academic Press (1984).
- [22] Huppert: *Endliche Gruppen I*, Springer Verlag (1967).
- [23] Johnson: *Topics in the Theory of Group Presentations*, LMS Lecture Notes No. 42, (1980).
- [24] Kernighan, Pike: *The UNIX Programming Environment*, Prentice Hall (1984).
- [25] Kernighan, Plauser: *The Elements of Programming Style*, McGraw-Hill (1978).
- [26] Kernighan, Ritchie: *The C Programming Language*, Prentice Hall (1978).
- [27] Leech: *Coset enumeration on digital computers*, Proc. Camb. Phil. Soc. 59 (1963), 257-268.
- [28] Leech: *Computer proof of relations in groups*, in *Topics in Group Theory and Computation*, (Curran ed.) Academic Press (1977).
- [29] Leech: *Coset enumeration*, in *Computational Group Theory* (ed. Atkinson) Academic Press (1984).

- [30] Lenat: *AM: Discovery in mathematics as heuristic search*, in Davis, Lenat: *Knowledge-Based Systems in Artificial Intelligence*, McGraw-Hill (1982).
- [31] Lenat: *The nature of heuristics*, *Artificial Intelligence*, 19 (1982) 189-249.
- [32] Lenat: *Eurisko: A program that learns new heuristics and domain concepts*, *AI Journal*, (1983).
- [33] Lenat: *Theory formation by heuristic search*, *AI Journal*, (1983).
- [34] Lenat, Harris: *Designing a rule system that searches for scientific discoveries*, *Artificial Intelligence*, (1977).
- [35] Linton: *Double coset enumeration*, (to appear).
- [36] Macdonald: *An application of coset enumeration*, *Austral. Comp. J.* 6 (1974) 46-48.
- [37] McLain: *An algorithm for determining defining relations for a subgroup*, *Glasgow Math. J.* 18 (1977) 51-56.
- [38] Mendelsohn: *An algorithmic solution for a word problem in group theory*, *Canad. J. Math* 16 (1964), 509-516.
- [39] Minsky: *Frames*, in Winston: *The Psychology of Computer Vision*, McGraw-Hill (1975).
- [40] Neubuser: *An elementary introduction to coset table methods*, in *Groups - St. Andrews 1981*, (LMS 71) ed. Campbell & Robertson.

- [41] Robertson: *Tietze transformations with weighted substring search*, J. Symbolic Computation 6 (1988) 59-64.
- [42] Sims: *Algorithms related to finitely presented groups*, notes from lectures given at 'Computational Group Theory', Durham 1982.
- [43] Sims: *Recent developments in computational group theory*, Notes from an address to the annual meeting of the Deutsche Mathematiker-Vereinigung, Marburg, 1986.
- [44] Stillwell: *The word problem and the isomorphism problem for groups*, Bull. Amer. Math. Soc. 6 (1982) 33-56.
- [45] Todd, Coxeter: *A practical method for enumerating cosets in an abstract finite group*, Proc. Edin. Math. Soc. 5 (1936) 25-36.
- [46] Trotter: *A machine program for coset enumeration*, Canad. Math. Bull. 7 (1964) 357-368.
- [47] Turner: *SASL Language manual*, University of St. Andrews (1976).
- [48] Turner: *An implementation of SASL*, University of St. Andrews, Dept. of Comp. Science, Report TR/75/4.
- [49] Turner: *A new implementation technique for applicative languages*, Software - Practice and Experience, 9 (1979) 31-49.