On Implementing Semigroups and Nearrings in GAP

Diplomarbeit
zur Erlangung des akademischen Grades eines Diplom-Ingenieurs in der Studienrichtung Technische Mathematik

von
Christof Nöbauer

Angefertigt am Institut für Mathematik der Technisch-Naturwissenschaftlichen Fakultät der Johannes Kepler Universität Linz

bei
A.Univ.-Prof. Dr. Günter Pilz

Linz, im Mai 1995

Introduction

GAP is a powerful computer algebra system being developed a the RWTH in Aachen. Some of its most exciting features are:

- it has a highly developed, easy to understand programming language incorporated.
- it is especially powerful when dealing with groups.
- it is free. Everybody can obtain it freely via anonymous ftp from the server `samson.math.rwth-aachen.de`.
- it is easily portable to a wide variety of operating systems on many hardware platforms.

Unfortunately, GAP has some disadvantages, too.

- the built in programming language is an *interpreter* language, which makes GAP programs relatively slow compared to compiler languages such as C or Pascal. GAP source cannot (yet) be compiled.
- the demands on system ressources (main memory and processor performance) are quite high if you want to do serious calculations.
- GAP has little or no knowledge about algebraic structures other than groups.

Therefore, a starting point for this work was to consider the possibility of implementing a basis for doing calculations on semigroups and nearrings in GAP. The result of this effort are a variety of library functions which will be presented in chapter 4.

In the process, the idea came up, that it would be nice to have examples of semigroups and nearrings at hand, preferrably in form of complete GAP libraries, which led to the problem of computing and storing all semigroups and nearrings of small orders. How this can be accomplished and up to which order one can compute will be a main theme throughout chapters 1, 2, and 3, cumulating in chapter 3, which is devoted to Yearby's algorithm.

Chapters 1 and 2 also give a brief introduction to those parts of semigroup and nearring theory used as basis for the implementation of some of the library functions.

Contents

Chapter 1

Semigroups

1.1 Some Theory on Transformation Semigroups

This first section deals with some theory about transformation semigroups which can be implemented nicely in GAP. A more detailed introduction to the basics about semigroup theory can be found e.g. in [How76]; much information about transformation semigroups (including everything contained in the first two sections) can be found e.g. in [Lal79].

1.1.1 Basic definitions and examples

A *semigroup* $(S, \cdot)$ is a set S together with a binary operation $\cdot$ on S s.t. $\cdot$ is *associative*, i.e. $\forall a, b, c \in S: \ (a \cdot b) \cdot c = a \cdot (b \cdot c)$. The operation $\cdot$ is usually referred to as *multiplication* and if there is no chance of confusion we shall write ab instead of $a \cdot b$ and S instead of $(S, \cdot)$.

A semigroup S is called a *monoid* if it has an *identity* element i.e. an element i s.t. $\forall s \in S:$ $si = is = s$. If S has an identity i then it is uniquely defined for if i and i' are identities then $i = i \cdot i' = i'$.

If T and U are subsets of S then $T \cdot U := \{t \cdot u \in S \mid t \in T \ \& \ u \in U\}$. A subset T of S is called a *subsemigroup* of S if $T \cdot T \subseteq T$.

A mapping ϕ from a semigroup $(S, \circ)$ into a semigroup $(T, *)$ is called a semigroup *homomorphism* if $\forall s_1, s_2 \in S: \ \phi(s_1 \circ s_2) = \phi(s_1) * \phi(s_2)$. A 1-1 (onto, bijective) homomorphism is called a monomorphism (epimorphism, isomorphism).

We say that a semigroup $(S, \circ)$ can be *embedded* into a semigroup $(T, *)$ if there exists a monomorphism from $(S, \circ)$ into $(T, *)$.

Definition 1.1 (transformation) Let X be a set. A mapping t from X into X shall be called a *transformation on* X. If x and y are two elements of X and the transformation t on X maps x into y then we shall denote this by $t(x) = y$.

Definition 1.2 (composition of transformations) Let t_1 and t_2 be two transformations on X, then we can define a new transformation, say t_3 on X by performing *composition* of mappings: $t_3 = t_1 \circ t_2$ where the transformation t_3 is defined by: $\forall x \in X: \ t_3(x) := t_1(t_2(x))$.

Example 1.3 Let $X := \{1, 2, 3\}$. Then by $t_1 := \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 2 \end{pmatrix}$ shall be denoted a transformation t_1 that maps 1 into 3, 2 into 2, and 3 into 2 which we may also write as $t_1(1) = 3, t_1(2) = 2, t_1(3) = 2$. Similarly, $t_2 := \begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{pmatrix}$ is a transformation that maps 1 into 2, 2 into 1, and 3 into 3. Note that composition of mappings (and therefore in particular of transformations) works from right to left. So we get $t_3 := t_1 \circ t_2$ by performing *first* t_2 and *then* t_1. It is

$$\begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 2 \end{pmatrix} \circ \begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 2 \end{pmatrix}$$

and *not* $\begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 1 \end{pmatrix}$.

Since composition of transformations is associative, we can make the following

Definition 1.4 (transformation semigroup) Let X be a set. The set of all transformations on X together with composition is called the *full transformation semigroup* of all transformations on X and denoted by $(\mathcal{T}(X), \circ)$. Any subsemigroup of $(\mathcal{T}(X), \circ)$ is called a *transformation semigroup on* X.

Proposition 1.5 Every semigroup S can be embedded into a monoid S^1.

Proof:
We construct the monoid S^1 by distinguishing two cases:

1. S has already an identity: nothing needs to be done; $S^1 := S$.
2. S has no identity: we adjoin a new element 1 to S and define: $\forall s \in S \; s1 = 1s := s$ and $11 := 1$. Thus $S^1 := S \cup \{1\}$ is the required monoid.

In both cases the inclusion mapping $\mathcal{I}$ (which in case 1. happens to be the identity mapping) is a monomorphism from S into S^1. □

S^1 is sometimes referred to as the monoid obtained from S by adjoining an identity if necessary.

Since the set of all subsemigroups of a semigroup S (including S itself) is a Moore - system, (see e.g. [Pil89] for the definition of a Moore - system) it makes sense to talk about generated semigroups: given any subset A of a semigroup S, the from A *generated* subsemigroup T consists of all finite products of elements of A. This shall be denoted by $\langle A \rangle := T$. The elements of A are called the *generators* of T. We shall consider this in the context of transformation semigroups:

Example 1.6 ([Lal79]) Let $S := (\mathcal{T}(\{1, 2, 3\}), \circ)$ be the semigroup of all transformations on the set $\{1, 2, 3\}$. Choose $A := \{\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 3 \end{pmatrix}, \begin{pmatrix} 1 & 2 & 3 \\ 1 & 1 & 2 \end{pmatrix}\}$. Then the generated semigroup $\langle A \rangle = \langle \{\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 3 \end{pmatrix}, \begin{pmatrix} 1 & 2 & 3 \\ 1 & 1 & 2 \end{pmatrix}\} \rangle$ consists of the following seven elements: the two generators $a_1 = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 3 \end{pmatrix}$ and $a_2 = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 1 & 2 \end{pmatrix}$ as well as $a_1 \circ a_1 = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 3 & 3 \end{pmatrix}$, $a_2 \circ a_2 = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 1 & 1 \end{pmatrix}$, $a_2 \circ a_1 \circ a_1 = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 2 & 2 \end{pmatrix}$, $a_1 \circ a_2 = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 2 & 3 \end{pmatrix}$, and $a_2 \circ a_1 = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 2 \end{pmatrix}$.

In a sense, the transformation semigroups are already all semigroups. This result is stated in the following proposition which is the semigroup theoretic pendant to Cayley's theorem in group theory.

Proposition 1.7 Every semigroup $(S, \cdot)$ is isomorphic to a transformation semigroup.

Proof:
We will show this by embedding $(S, \cdot)$ into a subsemigroup of $(\mathcal{T}(S^1), \circ)$:

- Construct S^1 by adjoining an identity to S if necessary as in the proof of proposition 1.5.
- Now, for all $s \in S$, construct a transformation $t_s : S^1 \to S^1$ by:
$$\forall x \in S^1 : \ t_s(x) := sx \tag{1.1}$$
- The mapping $\varphi : (S, \cdot) \to (\mathcal{T}(S^1), \circ);\ s \to t_s$ is a semigroup homomorphism since: $\forall x \in S : \ t_{s_1 \cdot s_2}(x) = s_1 \cdot s_2 \cdot x = t_{s_1}(s_2 \cdot x) = t_{s_1} \circ t_{s_2}(x)$ and therefore $\varphi(s_1 \cdot s_2) = t_{s_1 \cdot s_2} = t_{s_1} \circ t_{s_2} = \varphi(s_1) \circ \varphi(s_2)$.
- Moreover, φ is also one-one: $\varphi(s_1) = \varphi(s_2) \Rightarrow \forall x \in S^1 : \ t_{s_1}(x) = t_{s_2}(x) \Rightarrow \forall x \in S^1 : s_1 \cdot x = s_2 \cdot x \Rightarrow s_1 \cdot 1 = s_2 \cdot 1 \Rightarrow s_1 = s_2$. $\square$

Example 1.8 Consider the following semigroup $S := (\{1, 2, 3\}, \cdot)$, given by a Cayley table:

$\cdot$	1	2	3
1	1	1	1
2	1	2	1
3	3	3	3

Since this semigroup has obviously no identity, as in proposition 1.5, we add an extra element, say 4 and make it an identity, thus obtaining S^1:

$\cdot$	1	2	3	4
1	1	1	1	1
2	1	2	1	2
3	3	3	3	3
4	1	2	3	4

Now the transformations t_1, t_2, t_3 of the form (1.1) can be obtained easily from the first three rows of this Cayley table: $t_1 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 1 & 1 & 1 \end{pmatrix}$, $t_2 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 1 & 2 \end{pmatrix}$, and $t_3 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 3 & 3 & 3 \end{pmatrix}$. The semigroup $T := (\{t_1, t_2, t_3\}, \circ)$ is a transformation semigroup isomorphic to S.

1.1.2 Semigroup ideals and Greens's relations

In this section we provide the theory necessary to efficiently compute the minimal ideal of a transformation semigroup on a finite set X. Also, we shall see that Green's relations can be obtained almost immediately in a transformation semigroup.

Definition 1.9 (ideal of a semigroup) Let $(S, \cdot)$ be a semigroup. A nonempty subset R of S, such that $\forall r \in R, s \in S: \; r \cdot s \in R$ is called a *right ideal.* Analogously, a nonempty subset L of S, such that $\forall l \in L, s \in S: \; s \cdot l \in L$ is called a *left ideal.* A nonempty subset I of S, such that I is both, a left and a right ideal is called a (two sided) *ideal* of the semigroup S.
The fact that R (L) is a right (left) ideal of S may also be denoted by $R \cdot S \subseteq R$ $(S \cdot L \subseteq L)$.

Remark: The smallest right ideal containing a subset A of S is $A \cdot S^1 = A \cup A \cdot S$, the smallest left ideal containing a subset A of S is $S^1 \cdot A = A \cup S \cdot A$, and the smallest ideal containing a subset A of S is $S^1 \cdot A \cdot S^1 = A \cup A \cdot S \cup S \cdot A \cup S \cdot A \cdot S$.
Furthermore, $R \cdot S \subseteq R$ $(S \cdot L \subseteq L)$ implies $R \cdot R \subseteq R$ $(L \cdot L \subseteq L)$, therefore a (right, left) ideal of a semigroup S is in particular a subsemigroup of S.

Proposition 1.10 The intersection of two semigroup ideals is a semigroup ideal.

Proof:
Let I, J be two ideals of a semigroup S. Consider an arbitrary $i \in I \cap J$. $i \in I \Rightarrow \forall s \in S: \; is \in I \;\&\; si \in I$. $i \in J \Rightarrow \forall s \in S: \; is \in J \;\&\; si \in J$. Therefore $\forall s \in S: \; is \in I \cap J \;\&\; si \in I \cap J$.
$\square$

Definition 1.11 (minimal ideal of a semigroup) An ideal I of a semigroup S is called *minimal* if for any ideal J of S the following condition holds: $J \subseteq I \Rightarrow J = I$. If for all ideals J, the condition $I \subseteq J$ holds, I is called the *smallest* ideal. The definition of a minimal (smallest) right (left) ideal is analogous.

Proposition 1.12 If a semigroup S has a minimal ideal I, then it is the (uniquely defined) smallest ideal.

Proof:
Assume that I is a minimal ideal of S and let J be an arbitrary ideal of S.

- $I \cap J$ is nonempty: I an ideal implies $I \cdot J \subseteq I$, J an ideal implies $I \cdot J \subseteq J$ and therefore $I \cdot J \subseteq I \cap J$.
- By proposition 1.10, $I \cap J$ is an ideal. Certainly, $I \cap J \subseteq I$ and therefore, by assuming minimality of I, $I \cap J = I$ which implies $I \subseteq J$.

$\square$

Remark: Since for a semigroup S, S itself is an ideal, every finite semigroup has a smallest ideal. On the other hand, there exist semigroups which have no smallest ideal:

Example 1.13 All ideals of the semigroup $(\mathrm{N}, +)$ are of the form $\mathrm{N} + n$ for $n \in \mathrm{N}$. For $m \geq n, \mathrm{N} + m \subseteq \mathrm{N} + n$ and so $(\mathrm{N}, +)$ has no smallest ideal.

Two more propositions and another definition are required before we can characterize the form of the minimal ideal of a finite transformation semigroup:

Proposition 1.14 If a semigroup S (finite or not) has a minimal ideal I, then it is given by the union of all minimal right (left) ideals of S.

For the *proof* we refer e.g. to [Lal79].

Proposition 1.15 If a semigroup $(S, \cdot)$ is finite then it has at least one idempotent element, i.e. an element e s.t $e \cdot e = e$.

For the *proof* we refer e.g. to [How76].

Definition 1.16 (image, rank, and kernel of a transformation) Let t be a transformation on a set X. The subset $\{t(x) \mid x \in X\}$ of X is called the *image* of t, denoted by $Im(t)$. The size of $Im(t)$ is called the *rank* of the transformation t: $rank(t)$. Finally, we define an equivalence relation on X as follows: $\forall x, y \in X : \ x \sim_t y$ iff $t(x) = t(y)$. (Sometimes the equivalence relation $\sim_t$ is also referred to as $ker(t)$.)

Remark: For three transformations t_1, t_2, t_3 on a finite set X, the fact $t_1 = t_2 \circ t_3$ implies $rank(t_1) \leq \min(rank(t_2), rank(t_3))$ and $Im(t_1) \subseteq Im(t_2)$. On the other hand, if t_1, t_2 are transformations on a finite set X s.t. $Im(t_1) \subseteq Im(t_2)$, then we can construct a transformation $t \in \mathcal{T}(X)$ s.t $t_1 = t_2 \circ t$ by the following procedure: for $x \in X, t_1(x)$ is in $Im(t_1)$ and therefore in $Im(t_2)$. Hence, we can choose a $y \in X$ s.t. $t_2(y) = t_1(x)$. Then, by setting $t(x) := y$ we are through.

Example 1.17 Let $t_1 := \begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 1 \end{pmatrix}$, Let $t_2 := \begin{pmatrix} 1 & 2 & 3 \\ 2 & 2 & 3 \end{pmatrix}$ be two transformations on the set $X = \{1, 2, 3\}$. Then $rank(t_1) = rank(t_2) = 2; Im(t_1) = \{1, 2\}, Im(t_2) = \{2, 3\}$. $t_1 \circ t_2 = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 1 & 1 \end{pmatrix}$, so $Im(t_1 \circ t_2) = \{1\}$. This shows that the set inclusion in the previous remark may also be strict.

We are now in the position to compute smallest ideals of finite transformation semigroups:

Proposition 1.18 Let $(S, \circ)$ be a transformation semigroup on a finite set X. Then the smallest ideal I of S exists and it consists of all transformations in S whose rank is minimal, i.e. $I = \{t \in S \mid rank(t) = \min\{(rank(s)) \mid s \in S\}\}$.

Proof:
The set I is an ideal, since for $i \in I, s \in S, rank(s \circ i) = rank(i \circ s) \leq \min(rank(i), rank(s))$ and by minimality of $rank(i)$, $\min(rank(i), rank(s)) = rank(i)$. Hence, $rank(s \circ i) = rank(i \circ s)$ is minimal and therefore both, $i \circ s$ and $s \circ i$ are in I.

For an arbitrary $i \in I$, consider the right ideal $i \circ S$: we shall show that $i \in i \circ S$ and that in fact, $i \circ S$ is a *minimal* right ideal: for this, let R be an arbitrary right ideal s.t. $R \subseteq i \circ S$. We have to show: $R = i \circ S$.

- It suffices to show: $i \in R$:
  since $i \in R \Rightarrow i \in i \circ S \Rightarrow i \circ S = i \circ S^1$ is the smallest right ideal containing i and therefore $R = i \circ S$.

- in order to show that $i \in R$, let e be an idempotent in R. (this exists by proposition 1.15). e is of the form $e = i \circ s$ for some $s \in S$, which implies $Im(e) \subseteq Im(i)$ and furthermore, by minimality of $rank(i)$, $Im(i) = Im(e)$, in particular, $Im(i) \subseteq Im(e)$, which means (c.f. the remark after definition 1.16) that we can find a transformation t on X (t *not* necessarily in S!) s.t. $i = e \circ t$.

- e is an idempotent, so $i = e \circ t \Rightarrow e \circ i = e \circ e \circ t = e \circ t = i$. Now, $e \in R, i \in S$, hence $i = e \circ i$ implies $i \in R$.

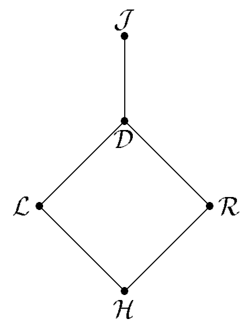
Now we are done since the ideal I is certainly contained in the union of the minimal right ideals $i \circ S$, which, by proposition 1.12 makes I the minimal ideal. $\square$

Example 1.19 Consider the semigroup of example 1.6. The minimal ideal of this semigroup consists of those transformations which have minimal rank, in this case the following three elements of rank 1: $\begin{pmatrix} 1 & 2 & 3 \\ 1 & 1 & 1 \end{pmatrix}$, $\begin{pmatrix} 1 & 2 & 3 \\ 2 & 2 & 2 \end{pmatrix}$, and $\begin{pmatrix} 1 & 2 & 3 \\ 3 & 3 & 3 \end{pmatrix}$.

The last part of this section deals with Green's relations:

Definition 1.20 (Green's relations) Let S be a semigroup. We define the equivalence relations $\mathcal{L}, \mathcal{R}, \mathcal{H}, \mathcal{D}, \mathcal{J}$ as follows: we say that two elements s, t of S are in $\mathcal{L}$-relation, iff they generate the same *left* ideal, i.e.: $s \;\mathcal{L}\; t :\Leftrightarrow S^1 s = S^1 t$. s, t are in $\mathcal{R}$-relation, iff they generate the same *right* ideal, i.e.: $s \;\mathcal{R}\; t :\Leftrightarrow sS^1 = tS^1$. s, t are in $\mathcal{J}$-relation, iff they generate the same *two-sided* ideal, i.e.: $s \;\mathcal{R}\; t :\Leftrightarrow S^1 s S^1 = S^1 t S^1$. $\mathcal{H}$ is defined as the *intersection* of $\mathcal{L}$ and $\mathcal{R}$. Finally, $\mathcal{D}$ is defined as the *join* of $\mathcal{L}$ and $\mathcal{R}$, i.e. the transitive closure of $\mathcal{L} \cup \mathcal{R}$.

As well known, those five equivalence relations form the following lattice: (with the lattice operations intersection and join)



Proposition 1.21 In a finite semigroup $\mathcal{J} = \mathcal{D}$ holds.

For the *proof* we refer e.g. to [How76].

In case that one has to consider Green's relations on the semigroup of all transformations on a finite set X, $\mathcal{T}(X)$, the following is very helpful:

Proposition 1.22 Let $(\mathcal{T}(X), \circ)$ be the semigroup of all transformations on a finite set X. Then for $t_1, t_2 \in S$ we have: $t_1 \mathcal{L} t_2 :\Leftrightarrow ker(t_1) = \ker(t_2)$, $t_1 \mathcal{R} t_2 :\Leftrightarrow Im(t_1) = Im(t_2)$, and $t_1 \mathcal{J} t_2 :\Leftrightarrow rank(t_1) = rank(t_2)$.

For the *proof* we refer e.g. to [How76].

Unfortunately, this nice property is not inherited to subsemigroups of $\mathcal{T}(X)$:

Example 1.23 Consider the following transformation semigroup consisting of the three elements $t_1 = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 1 & 1 \end{pmatrix}$, $t_2 = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 1 & 2 \end{pmatrix}$, and $t_3 = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 1 & 3 \end{pmatrix}$. Those three elements generate three different left ideals S^1t_1, S^1t_2, S^1t_3, yet $ker(t_2) = ker(t_3)$.

But, all is not lost. At least, we know that if S is a subsemigroup of T, then Green's relations $\mathcal{L}, \mathcal{R}$ on S are contained in the restrictions on S of Green's relations $\mathcal{L}, \mathcal{R}$ on T. The previous example shows that those restrictions may indeed be strict. Nonetheless, we can save computation time when computing $\mathcal{L}, \mathcal{R}$ of a transformation semigroup by first applying proposition 1.22 and then refining the resulting relations by applying definition 1.20.

1.2 A First Approach to Computing and Storing Semigroups

As soon as one is able to represent (transformation) semigroups on a computer and having implemented some functions for them, one might have the idea that it could be nice to have at hand a few examples of semigroups on which these functions could be tried out. Yet, even better than a few semigroups are all semigroups of a certain size. (Even if it is just the curiosity to know how many there are).

This leads very naturally to the following two questions:

1. given $n \in \mathrm{N}$,
   - how can one (efficiently) compute all semigroups of size n?
   - Which of these semigroups are isomorphic?
2. how can these semigroups be stored in a data structure s.t.
   - the amount of required memory is not unreasonably high?
   - the effort to retrieve a semigroup from the data structure is within reasonable limits?

The theory we shall present here is a generalization of a method by Clay to compute nearrings ([Cla68], [Cla70], cf. section 2.2) as described in [Yea73].

1.2.1 Theoretic aspects of computing and storing semigroups

We shall begin with the first question:

The first thing one has to think about is how to represent a binary operation $*$ on a given set S of size n. Certainly, a binary operation on S could be represented by an $n \times n$ matrix representing the Cayley table of $*$.

However, there is another possibility:

Proposition 1.24 [Yea73] Let S be a set and denote by S^S the set of all functions from S into S (i.e. S^S is the set of all transformations on S).

There is a 1-1 correspondence between all binary operations on S and all functions $f: S \to S^S$.

Proof:
Let $\mathcal{B}_S$ denote the set of all binary operations on S and let $\mathcal{F}_S$ denote the set of all functions $f: S \to S^S$.

Consider $\phi: \mathcal{F}_S \to \mathcal{B}_S$ where $\phi(f) := *_f$ with $x *_f y := f(y)(x)$ for all $x, y \in S$.

- clearly, ϕ is well-defined.
- ϕ is 1-1: let $f, g \in \mathcal{F}_S$ s.t. $f \neq g$ i.e. there is at least one pair $(\bar{x}, \bar{y})$ s.t. $f(\bar{x})(\bar{y}) \neq g(\bar{x})(\bar{y})$. But then $\bar{y} *_f \bar{x} \neq \bar{y} *_g \bar{x}$.
- ϕ is onto: let $*$ be a binary operation on S. Then we get a function $f_* \in \mathcal{F}_S$ by defining $f_*(x)(y) := y * x$ for all $x, y \in S$. □

The next thing one has to think about is how the representation of a binary relation $*$ on a set S as a function $f_*: S \to S^S$ connects to associativity of $*$:

Proposition 1.25 [CP61] Let $*$ be a binary operation on a set S. Then $*$ is associative iff

$$\forall x, y \in S: \ f_*(f_*(x)(y)) = f_*(x) \circ f_*(y) \tag{1.2}$$

Proof:
$*$ associative on S $\Leftrightarrow \forall x, y, z \in S: \ (z * y) * x = z * (y * x) \Leftrightarrow \forall x, y, z \in S: \ f_*(x)(f_*(y)(z)) = f_*(f_*(x)(y))(z) \Leftrightarrow \forall x, y \in S: \ f_*(x) \circ f_*(y) = f_*(f_*(x)(y))$. □

Notation: if a function f_* has property (1.2) then it shall be called a *valid function*.

Another proposition provides a method to classify semigroups, i.e. to put them into classes of isomorphic semigroups:

Proposition 1.26 [Yea73] Two semigroups on a set S, $(S, *_f)$ and $(S, *_g)$ are isomorphic iff there exists a bijection $\alpha \in S^S$ s.t.

$$\forall x \in S: \ \alpha \circ f(x) = g(\alpha(x)) \circ \alpha \tag{1.3}$$

Proof:
$(S, *_f) \cong_\alpha (S, *_g) \Leftrightarrow \forall x, y \in S: \ \alpha(y *_f x) = \alpha(y) *_g \alpha(x) \Leftrightarrow \forall x, y \in S: \ \alpha(f(x)(y)) = g(\alpha(x))(\alpha(y)) \Leftrightarrow \forall x \in S: \ \alpha \circ f(x) = g(\alpha(x)) \circ \alpha$. □

These three propositions - as Yearby ([Yea73]) wrote - *suggest a procedure for constructing, representing, and classifying all semigroups with the aid of a computer.* This can be summarized by the following two algorithms:

Algorithm 1 (`ValidFunctionsSg`)

```
Input:  n:  a positive integer defining the set S := {1,2,...,n}
Output: valid_functions:  a set of all functions f:  S → S^S
        that satisfy condition (1.2).

  S := {1,2,...,n}; valid_functions := {};
  F := {f:  S → S^S}.
  for f in F do
    if f satisfies (1.2) then
      valid_functions := valid_functions ∪ {f};
    fi;
  od;
  return valid_functions;
```

Algorithm 2 (`ClassifySg`)

```
Input:  S: a set of the form S = {1,2,...,n}.
        valid_functions:  a set of valid functions as generated
        by algorithm 1 for the set S.
Output: classes:  a record consisting of all classes of the
        classified semigroups.

  for f in valid_functions do
    valid_functions := valid_functions - {f};
    class.phi := f;
    class.bijs_yielding_iso_sgps := {I};
    for g in valid_functions do
      if there is a bijection α on S s.t.  f, g, α satisfy (1.3) then
        valid_functions := valid_functions - {g};
        class.bijs_yielding_iso_sgps := class.bijs_yielding_iso_sgps ∪ {g};
      fi;
    od;
    add class to classes;
  od;
  return classes;
```

Both algorithms have been implemented as GAP functions. The source code can be found in appendix B.8. Note that algorithm 2 also solves the problem of storing semigroups.

Remark: when referring to a function $t : S \to S$ we shall make use of the following

Proposition 1.27 There is a 1-1 correspondence between all transformations on the set $S := \{1, 2, \dots, n\}$ and the set $T := \{1, 2, \dots, n^n\}$.

Proof:
Let $t \in S^S$ s.t. $\forall i \in S: \; t(i) = j_i \in S$. Then we define $\Phi : S^S \to T$ as $\Phi(t) := \left(\sum_{i\in\{1,\ldots,n\}}(j_i - 1)\cdot n^{i-1}\right) + 1$. Given a number $m \in T$, we can construct a function $t_m \in S^S$ by computing $(m-1)_n$, the representation of $m-1$ in the n-system and defining t_m by $t_m(i) := (d_i)_{10} + 1$ where $(d_i)_{10}$ is the i^{th} digit of $(m-1)_n$ in the decimal system. This makes Φ onto and by $|S^S| = |T|$ also 1-1. □

Therefore, instead of storing the whole function t, it suffices to store only a number $m \in T$ representing the function.

Notation: a function from an ordered set S of size m into an ordered set T of size n may also be denoted as list: $[i_1, i_2, \ldots, i_m]$ with $i_k \in \{1,\ldots,n\}$ $\forall k \in \{1,\ldots,m\}$ is to be interpreted as the function $f : S \to T$ that maps the first element of S to the i_1^{th} element of T, the 2^{nd} element of S to the i_2^{th} element of T and so on.

1.2.2 A sample computation

Example 1.28 Using algorithm 1 to compute the semigroups of order 2 (i.e. all associative binary operations that can be defined on the set $S := \{1,2\}$) returns the following ordered set `valid_functions` of 8 valid functions representing all existing 8 semigroups of order 2:

```
[ [ 1, 1 ], [ 1, 2 ], [ 1, 4 ], [ 2, 2 ], [ 2, 3 ], [ 2, 4 ], [ 3, 2 ],
  [ 4, 4 ] ]
```

In accordance to proposition (1.27) each single number represents a function: $f : S \to S$: 1 represents `[ 1, 1 ]`, 2 represents `[ 1, 2 ]`, 3 represents `[ 2, 1 ]`, and 4 represents `[ 2, 2 ]`.

Using this we can construct the following Cayley tables of all 8 semigroups definable on the set $S = \{1,2\}$:

$\cdot_1$	1	2
1	1	1
2	1	1

$\cdot_2$	1	2
1	1	1
2	1	2

$\cdot_3$	1	2
1	1	2
1	1	2

$\cdot_4$	1	2
1	1	1
1	2	2

$\cdot_5$	1	2
1	1	2
2	2	1

$\cdot_6$	1	2
1	1	2
2	2	2

$\cdot_7$	1	2
1	2	1
1	1	2

$\cdot_8$	1	2
1	2	2
1	2	2

Using the above set `valid_functions` as input for algorithm (2) in order to classify these 8 semigroups, we get the following record:

```
rec(
  1 := rec(
      phi := [ 1, 1 ],
      bijs_yielding_iso_sgps := [ 2, 3 ] ),
  2 := rec(
      phi := [ 1, 2 ],
```

```
    bijs_yielding_iso_sgps := [ 2, 3 ] ),
3 := rec(
    phi := [ 1, 4 ],
    bijs_yielding_iso_sgps := [ 2 ] ),
4 := rec(
    phi := [ 2, 2 ],
    bijs_yielding_iso_sgps := [ 2 ] ),
5 := rec(
    phi := [ 2, 3 ],
    bijs_yielding_iso_sgps := [ 2, 3 ] ) )
```

which tells us that in fact, $(S, \cdot_1)$ and $(S, \cdot_8)$ are isomorphic, as well as $(S, \cdot_2)$ and $(S, \cdot_6)$ and $(S, \cdot_5)$ and $(S, \cdot_7)$.

1.2.3 A remark on performance

Obviously, the complexity of algorithm 1 is $\mathcal{O}(n^{(n^2)})$ which makes this algorithm impractical for orders greater than 3. The dramatic increase of complexity (including (estimated) timings taken on a 486-33 Linux machine) shall be visualized in the following table:

order	*complexity*	*timing*
1	$1^1 = 1$	< 1 *sec*
2	$2^4 = 16$	< 1 *sec*
3	$3^9 = 19683$	32 *sec*
4	$4^{16} \approx 4 \cdot 10^9$	*est.* 180 *days*
5	$5^{25} \approx 3 \cdot 10^{17}$	*est.* $55 \cdot 10^6$ *years*

Nonetheless, for orders between 1 and 3 we can summarize:

order	*# of semigroups*	*# of classes*
1	1	1
2	8	5
3	113	24

Concluding this section, it remains to say, that this first attempt to compute semigroups yielded no quite satisfactory results. Algorithm 1 is definitely not efficient. More sophisticated methods are required. (What could be achieved by better methods is illustrated by the fact that Petrich ([Pet73]) managed to compute all non-isomorphic semigroups of order 4 by hand).

Chapter 2

Nearrings

2.1 Some Theory on Nearrings

This section provides some material on nearrings. For a detailed introduction to the theory of nearrings we refer e.g. to the books [Cla92] or [Pil83].

Definition 2.1 A nonempty set N together with two binary operations $+$ and $\cdot$ is called a *right nearring* iff

1. $(N, +)$ is a (not necessarily abelian) group.
2. $(N, \cdot)$ is a semigroup.
3. $\cdot$ is right distributive over $+$ i.e. $\forall\, n_1, n_2, n_3 \in N: \ (n_1 + n_2) \cdot n_3 = n_1 \cdot n_3 + n_2 \cdot n_3$.

Remark: certainly, in 3., one could also choose left distributivity (i.e. $\forall\, n_1, n_2, n_3 \in N$: $n_1 \cdot (n_2 + n_3) = n_1 \cdot n_2 + n_1 \cdot n_3$) which would lead to an analogous theory. But we shall stick to right distributivity and whenever we talk about *nearrings* we shall mean *right nearrings*.

Proposition 2.2 Let $(N, +, \cdot)$ be a (right) nearring with 0 the neutral element of $(N, +)$. Then $\forall n \in N: \ 0 \cdot n = 0$.

Proof:
$0 \cdot n = (0 + 0) \cdot n = 0 \cdot n + 0 \cdot n \Rightarrow 0 \cdot n = 0.$ □

Definition 2.3 (subnearrings) Let $(N, +, \cdot)$ be a nearring. A subset M of N is called a *subnearring* of N iff $(M, +)$ is a subgroup of $(N, +)$ and $M \cdot M \subseteq M$ (i.e. $(M, \cdot)$ is a subsemigroup of $(N, \cdot)$).

Definition 2.4 (nearring homomorphism) Let $(N_1, +_1, \cdot_1)$ and $(N_2, +_2, \cdot_2)$ be nearrings. A function $\phi : (N_1, +_1, \cdot_1) \longrightarrow (N_2, +_2, \cdot_2)$ is called a nearring *homomorphism* between N_1 and N_2 iff $\forall m, n \in N_1: \ \phi(m +_1 n) = \phi(m) +_2 \phi(n)$ & $\phi(m \cdot_1 n) = \phi(m) \cdot_2 \phi(n)$. The definition of nearring endomorphisms, isomorphisms, automorphisms is clear.

The most important example for our purposes is:

Definition 2.5 (transformation nearrings) Let G be a group. Then **the set of all transformations on G together with pointwise addition and composition is a (right) nearring**. This nearring shall be denoted by $(M(G), +, \circ)$ or shortly by $M(G)$. All subnearrings of $M(G)$ shall be called transformation nearrings.

Example 2.6 Consider $C_2 = (\{0,1\}, +)$, the cyclic group of order two. Take the following four transformations on C_2, given by the transformation notation as described in chapter 1: $t_0 := \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$, $t_1 := \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix}$, $t_2 := \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$, and $t_3 := \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$. These transformations form the nearring $(M(C_2), +, \circ)$ given by the Cayley tables:

$+$	t_0	t_1	t_2	t_3
t_0	t_0	t_1	t_2	t_3
t_1	t_1	t_0	t_3	t_2
t_2	t_2	t_3	t_0	t_1
t_3	t_3	t_2	t_1	t_0

$\circ$	t_0	t_1	t_2	t_3
t_0	t_0	t_0	t_0	t_0
t_1	t_0	t_1	t_2	t_3
t_2	t_3	t_2	t_1	t_0
t_3	t_3	t_3	t_3	t_3

From this we can see that the additive group of this nearring is isomorphic to Klein's four group $C_2 \times C_2$ and that the multiplicative semigroup has an identity, namely t_1.

Like with semigroups, the subnearrings of a nearring N form an inductive Moore system. Therefore, for a subset M of N we can consider $\langle M \rangle$, the subnearring of N *generated* by M. $\langle M \rangle$ is the smallest subnearring of N that contains M.

Example 2.7 The nearring in example 2.6 is generated by the group transformation $t_2 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ on C_2 since $t_2 + t_2 = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} = t_0$, $t_2 \circ t_0 = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = t_3$, and $t_2 + t_3 = \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix} = t_1$.

Definition 2.8 (nearring ideals) Let $(N, +, \cdot)$ be a nearring. A normal subgroup I of $(N, +)$ is called a *left ideal* of the nearring N if $\forall\, n_1, n_2 \in N\ \forall\, i \in I :\ n_1 \cdot (n_2 + i) - n_1 \cdot n_2 \in I$. I is called a *right ideal* of N if $I \cdot N \subseteq I$ and I is a (two sided) *ideal* of N if it is both, a left and a right ideal.

Example 2.9 Let N be the nearring from example 2.6. The normal subgroups of $(N, +)$ are $I_1 := \{t_0\}$, $I_2 := \{t_0, t_1\}$, $I_3 := \{t_0, t_2\}$, $I_4 := \{t_0, t_3\}$, and I_5: the group $(N, +)$ itself. I_1, I_4, and I_5 are two sided ideals, I_2 and I_3 are left ideals but not right ideals.

When implementing some features for nearrings, we shall make use of a famous result from group theory:

Proposition 2.10 (Cayley's theorem) Every group $(G, \cdot)$ is embeddable into $(\mathcal{S}(G), \circ)$ i.e. the group of all bijective transformations from G into G (= permutations on G) together with composition. (And therefore G is isomorphic to a permutation group on G).

Proof:
The proof is similar to the proof of proposition 1.7, except that now we need not adjoin an identity to G since we already have one. As in 1.7 for all $g \in G$, we construct a transformation $t_g : G \to G$ (which is now bijective, of course) by $\forall x \in G :\ t_g(x) := gx$. Then the remaining steps of the proof can be performed in total analogy to 1.7. $\square$

2.2 A First Approach to Computing and Storing Nearrings

With the same motivation as in section 1.2 we can now try to compute nearrings and like before we can ask:

1. given a group $(G,+)$,
   - how can one (efficiently) compute all nearrings definable on this group G (i.e. how many binary operations $*$ can one define on G s.t. $*$ is associative and right distributive over $+$ thus making $(G,+,*)$ into a nearring)?
   - Which of these nearrings are isomorphic?
2. how can these nearrings be stored in a data structure s.t.
   - the amount of required memory is not unreasonably high?
   - the effort to retrieve a nearring from the data structure is within reasonable limits?

We shall start by presenting a method developed by Clay ([Cla68] and [Cla70]).

2.2.1 Clay's method to compute nearrings

Given a group $(G,+)$, as in section 1.2, we could represent all binary operations $*$ on G as functions $f_* : G \to G^G$. Yet, since we are only interested in *right distributive* binary operations we can make use of:

Proposition 2.11 Let $(G,+)$ be a group and denote by $End(G)$ the set of all endomorphisms from G into G.

There is a 1-1 correspondence between all over $+$ right distributive binary operations $*$ on G and all functions $f: G \to End(G)$.

Proof:
Let $\mathcal{RD}_G$ denote the set of all over $+$ right distributive binary operations on G and let $\mathcal{F}_G$ denote the set of all functions $f: G \to End(G)$.

As in proposition 1.24, we define $\phi: \mathcal{F}_G \to \mathcal{RD}_G$ where $\phi(f) := *_f$ with $a *_f b := f(b)(a)$ for all $a, b \in G$. Because of proposition 1.24 it remains only to show

- right distributivity of $*_f$: for all $a,b,c \in G$ we have: $(a+b) *_f c = f(c)(a+b)$ and since $f(c) \in End(G)$, $f(c)(a+b) = f(c)(a) + f(c)(b) = a *_f c + b *_f c$.
- $f_*(c) \in End(G)$: let $*$ be right distributive, i.e. for all $a,b,c \in G$: $(a+b)*c = a*c+b*c \Leftrightarrow f_*(c)(a+b) = f_*(c)(a) + f_*(c)(b) \Leftrightarrow f_*(c) \in End(G)$. □

The question, what a function $f_* : G \to End(G)$ must look like s.t. $*$ is associative can be treated analogously to proposition 1.25:

Proposition 2.12 Let $*$ be a binary operation on a group G. Then $*$ is associative iff

$$\forall a, b \in G: \; f_*(f_*(a)(b)) = f_*(a) \circ f_*(b) \tag{2.1}$$

The *Proof* is analogous to the proof of proposition 1.25 and can therefore be omitted.

Note that condition (2.1) is identical to condition (1.2) in section 1.2 and therefore we can recall:
Notation: if a function f_* has property (2.1) then it shall be called a *valid function*.

The next proposition, which provides a method to classify nearrings, is also quite similar to the semigroup case (cf. proposition 1.26):

Proposition 2.13 Two nearrings on a group $(G, +)$, $(G, +, *_f)$ and $(G, +, *_g)$ are isomorphic iff there exists a group automorphism $\alpha \in Aut((G, +))$ s.t.

$$\forall a \in G: \; \alpha \circ f(a) = g(\alpha(a)) \circ \alpha \tag{2.2}$$

The *Proof* is similar to the proof of proposition 1.26 and can therefore be omitted.

Theoretically, we are now in position to effectively (but by no means efficiently) compute and classify all nearrings on a given group G. However, before we proceed, we shall present two more propositions that will reduce the complexity of the problem of computing all nearrings. The first proposition gives a *necessary* condition for the associativity of a right distributive binary operation $*$:

Proposition 2.14 Let $(G, +)$ be a group with zero 0 and let $*$ be an over $+$ right distributive binary operation on G.

If $*$ is associative then $f_*(0) \in End(G)$ is an idempotent endomorphism.

Proof:
From condition (2.1) we know that $*$ associative $\Leftrightarrow \forall a, b \in G: \; f_*(f_*(a)(b)) = f_*(a) \circ f_*(b)$.
In particular, for $a = b = 0$, this implies $f_*(f_*(0)(0)) = f_*(0) = f_*(0) \circ f_*(0)$. □

The next proposition gives us a criterion when we can stop the computation:

Proposition 2.15 Let $(G, +)$ be a group with zero 0 and let $*$ be an over $+$ right distributive binary operation on G. Denote by $\mathcal{I}$ the identity endomorphism on $(G, +)$.

If $f_*(0) = \mathcal{I}$ then $*$ is associative iff $f_*(a) = \mathcal{I} \; \forall a \in G$.

Proof:

"$\Leftarrow$": let $f_*(a) = \mathcal{I} \; \forall a \in G$. Then $f_*(f_*(a)(b)) = f_*(a) \circ f_*(b) \; \forall a, b \in G$ and therefore $*$ is associative.

"$\Rightarrow$": for all $a \in G$ it is $f_*(a) = f_*(a) \circ \mathcal{I} = f_*(a) \circ f_*(0) \overset{(2.1)}{=} f_*(f_*(a)(0)) = f_*(0) = \mathcal{I}$. □

Now we are in position to summarize:

Algorithm 3 (`ValidFunctionsNr`)

```
Input:  G: a group
Output: valid_functions:  a set of all functions f:  G → End(G)
        that satisfy condition (2.1).

  valid_functions := {};
  F := {f:  G → End(G) | f(0) ≠ I }.
  for f in F do
    if f(0) is an idempotent endomorphism then
      if f satisfies (2.1) then
        valid_functions := valid_functions ∪ {f};
      fi;
    fi;
  od;
  construct h as h: G → End(G) s.t. for all a ∈ G: h(a) := I;
  return valid_functions ∪ {h};
```

Algorithm 4 (`ClassifyNr`)

```
Input:  G: a group
        valid_functions:  a set of valid functions as generated
        by algorithm 3 for the group G.
Output: classes:  a record consisting of all classes of the
        classified nearrings.
  for f in valid_functions do
    valid_functions := valid_functions - {f};
    class.phi := f;
    class.autos_yielding_iso_nrs := {I};
    for g in valid_functions do
      if there is an automorphism α on G s.t.  f, g, α satisfy (2.2) then
        valid_functions := valid_functions - {g};
        class.autos_yielding_iso_nrs := class.autos_yielding_iso_nrs ∪ {g};
      fi;
    od;
    add class to classes;
  od;
  return classes;
```

The source code of both algorithms implemented as GAP functions can be found in appendix B.7.

2.2.2 A sample computation

Example 2.16 Performing `ValidFunctionsNr` on the cyclic group C_3 of order 3 yields the following list of valid functions:

```
[ [ 1, 1, 1 ], [ 1, 1, 3 ], [ 1, 2, 3 ], [ 1, 3, 1 ], [ 1, 3, 2 ],
  [ 1, 3, 3 ], [ 3, 3, 3 ] ]
```

Every sublist in this list represents a valid function $f : C_3 \to End(C_3)$. Now we use `ClassifyNr` on this list (and also the function `ConstructNr` which is used to put the record together - cf. the source code in appendix B.7) to get the the following record which contains all the information necessary to represent all the nearrings that are there on C_3:

```
rec(
  group_name := "C3",
  group_generators := [ (1,2,3) ],
  elements := rec(
      1 := (),
      2 := (1,2,3),
      3 := (1,3,2) ),
  group_endomorphisms := rec(
      1 := [ 1, 1, 1 ],
      2 := [ 1, 3, 2 ],
      3 := [ 1, 2, 3 ] ),
  classes := rec(
      1 := rec(
          phi := [ 1, 1, 1 ],
          autos_yielding_iso_nrs := [ 3 ] ),
      2 := rec(
          phi := [ 1, 1, 3 ],
          autos_yielding_iso_nrs := [ 2, 3 ] ),
      3 := rec(
          phi := [ 1, 2, 3 ],
          autos_yielding_iso_nrs := [ 2, 3 ] ),
      4 := rec(
          phi := [ 1, 3, 3 ],
          autos_yielding_iso_nrs := [ 3 ] ),
      5 := rec(
          phi := [ 3, 3, 3 ],
          autos_yielding_iso_nrs := [ 3 ] ) ) )
```

Each of the 5 subrecords of the record field `classes` represents an equivalence class of isomorphic nearrings. The entry `phi` in the 5 subrecords represents a valid function $\phi : C_3 \to End(C_3)$ which in turn represents the representative nearring of the class. Each single number in a list `phi` points to the corresponding endomorphism on C_3 listed in the record field `group_endomorphisms`. Each single number in a list `autos_yielding_iso_nrs` points to a corresponding endomorphism in `group_endomorphisms` which is in fact a group automorphism and yields an isomorphic nearring. Note that by convention, the last endomorphism is always the identity automorphism $\mathcal{I}$.

Remark: this is not as confusing as it may look at first sight. In fact, it is a "computerized" version of how nearrings are presented in the appendix of [Pil83].

Example 2.17 In the previous example, what is 1 times 3 in the 4^{th} class of nearrings represented by `phi = [ 1, 3, 3 ]`? In theory, $a * b = \phi(b)(a)$ - and this is exactly what we do here: `1*3 = phi[3][1]`: `phi[3] = 3`, i.e. it points to the 3^{rd} endomorphism which is the identity $\mathcal{I}$ and therefore `phi[3][1]` = $\mathcal{I}$`[1]` $= 1$.

We use the computer to determine the nearrings:

Certainly, the addition of all seven nearrings is addition of C_3:

+	1	2	3
1	1	2	3
2	2	3	1
3	3	1	2

The seven different nearring multiplications are given by the following Cayley tables (in the same order as their representations occurred in the list of valid functions):

$*_1$	1	2	3
1	1	1	1
2	1	1	1
3	1	1	1

$*_2$	1	2	3
1	1	1	1
2	1	1	2
3	1	1	3

$*_3$	1	2	3
1	1	1	1
2	1	3	2
3	1	2	3

$*_4$	1	2	3
1	1	1	1
2	1	2	1
3	1	3	1

$*_5$	1	2	3
1	1	1	1
2	1	2	3
3	1	3	2

$*_6$	1	2	3
1	1	1	1
2	1	2	2
3	1	3	3

$*_7$	1	2	3
1	1	1	1
2	2	2	2
3	3	3	3

Multiplications $*_2$ and $*_4$ yield isomorphic nearrings as well as $*_3$ and $*_5$ (which is in fact the field of order 3).

2.2.3 A few remarks on performance

For a given group $(G, +)$, algorithm 3 loops over all functions $f : G \rightarrow End(G)$ s.t. $f(0)$ is an idempotent endomorphism and $f(0) \neq \mathcal{I}$. Let i denote the total number of idempotent endomorphisms on G (including $\mathcal{I}$); let $ord(G)$ be the size of the group G and denote by $|End(G)|$ the total number of endomorphisms of G.

Then algorithm 3 loops over a total of $(i-1) \cdot |End(G)|^{ord(G)-1}$ functions.

Performing test runs with the GAP function `ValidFunctionsNr` (cf. appendix B.7) reveals that the practical use of this algorithm comes to its limits as soon as the order of the group G reaches 8. The following table shall give a survey over the total number of required loop executions for all groups of low order and an (estimated) timing (again taken on a 486-33 Linux machine) for the groups of orders 2 to 8:

group	*# of endo-morphisms on G*	*# of idempotent endomorphisms on G*	*# of loop executions*	*time*
C_2	2	2	2	< 1 *sec.*
C_3	3	2	9	< 1 *sec.*
C_4	4	2	64	< 1 *sec.*
V_4	16	8	28672	58 *sec.*
C_5	5	2	625	2 *sec.*
C_6	6	4	23328	63 *sec.*
S_3	10	5	400000	17 *min.*
C_7	7	2	117649	8 *min.*
C_8	8	2	$\approx 2 \times 10^6$	156 *min.*
$C_2 \times C_4$	32	10	$\approx 3 \times 10^{11}$	*est.* 117 *years*
$C_2 \times C_2 \times C_2$	512	58	$\approx 5.3 \times 10^{20}$	*est.* 2.3×10^{11} *years*
D_8	36	10	$\approx 7 \times 10^{11}$	*est.* 276 *years*
Q_8	28	2	$\approx 1.3 \times 10^{10}$	*est.* 5 *years*
C_9	9	2	$\approx 4.3 \times 10^7$	
$C_3 \times C_3$	81	14	$\approx 2.4 \times 10^{16}$	
C_{10}	10	4	3×10^9	
D_{10}	26	7	$\approx 3.3 \times 10^{13}$	
C_{11}	11	2	$\approx 2.6 \times 10^{10}$	
C_{12}	12	4	$\approx 2.2 \times 10^{12}$	
$C_2 \times C_6$	48	16	$\approx 4.7 \times 10^{19}$	
D_{12}	64	21	$\approx 1.5 \times 10^{21}$	
A_4	33	6	$\approx 2.5 \times 10^{17}$	
T	20	5	$\approx 8.2 \times 10^{14}$	
C_{13}	13	2	$\approx 2.3 \times 10^{13}$	
C_{14}	14	4	$\approx 2.4 \times 10^{15}$	
D_{14}	50	9	$\approx 9.8 \times 10^{22}$	
C_{15}	15	4	$\approx 8.8 \times 10^{16}$	

Remark: There would, of course be no use in trying `ValidFunctionsNr` on a supercomputer (after all, it does not matter if one waits one million years or only one thousand years for the result...). It would be nice to have a better method at hand or, to put it with Clay's words ([Cla70]): *The computer, in a sense, has assumed the job of looking for the needles in the haystack. There still remains the problem, however, of too big of a haystack for the number of needles therein. I.e. there is a need for better searching techniques to construct near-rings on groups of orders greater than seven.*

Chapter 3

Yearby's Algorithm

In this chapter we shall present an algorithm introduced by Yearby ([Yea73]). An implementation of this algorithm in Fortran IV enabled him to compute all nearrings on all groups of orders 2 to 11 and on three of the five groups of order 12.

We shall present an implementation of this algorithm in GAP which turned out to be useful for computing all nearrings on all groups of small order (i.e. on the 27 groups of orders 2 to 15) as well as all semigroups of orders 2 to 5.

3.1 Computing Valid Functions

The bad performance of the functions `ValidFunctionsSg` (algorithm 1) and `ValidFunctionsNr` (algorithm 3) is a result of their basic design: first all possible functions $f : S \to S^S$ (resp. $f : G \to End(G)$) are generated and then each of these functions is checked for property (1.2) (resp. (2.1)) i.e. if it is a valid function.

One may expect a much better performance if the process of checking for property (1.2) (resp. (2.1)) could somehow be integrated into the process of creating all functions i.e. while creating a function $f : S \to S^S$ (resp. $f : G \to End(G)$), if it turns out that it never will satisfy property (1.2) (resp. (2.1)), we could immediately turn to constructing the next function, thus (hopefully) saving a lot of unnecessary computation time.

We shall start the presentation of Yearby's method with:

3.1.1 Extending partial multiplications

The following definition provides a generalized environment where the algorithm can be applied:

Definition 3.1 (Yearby pair) Let S be a nonempty arbitrary set. A pair $(E, \Gamma)_S$ shall be called a *Yearby pair w.r.t. S* if

- E is a nonempty set of transformations on S, closed under composition $\circ$ (i.e. $(E, \circ)$ is a subsemigroup of $(\mathcal{T}(S), \circ)$, the semigroup of all transformations on S together with composition)

- Γ is a set of bijections on S

s.t.

$$\forall \gamma \in \Gamma, e \in E: \ \gamma^{-1} \circ e \circ \gamma \in E \tag{3.1}$$

Example 3.2

1. let $(G, +)$ be a group. Set $E := End((G,+)), \Gamma := Aut((G,+))$ then $(E,\Gamma)_G$ is a Yearby pair.
2. let S be a set. Set $E := \mathcal{T}(S), \Gamma := \{\alpha \in \mathcal{T}(S) \mid \alpha \text{ is } \textit{bijective}\}$ then $(E,\Gamma)_S$ is a Yearby pair.

Now we are in position to define the fundamental concept of this chapter:

Definition 3.3 (partial multiplication) [Yea73] Let S, T be sets s.t. $\{\} \neq S \subseteq T$. A mapping $\nabla : S \times S \to T$ is called a *partial multiplication on S relative to T.*

In analogy to binary operations (cf. proposition 1.24), partial multiplications on S relative to T and functions $f : S \to T^S$ "are the same":

Proposition 3.4 Let S, T be sets s.t. $\{\} \neq S \subseteq T$. There is a 1-1-correspondence between all partial multiplications on S relative to T and all functions $f : S \to T^S$.

Proof:
In analogy to the proof of proposition 1.24, we can switch between ∇_f and f_∇ by the setting: $s\nabla_f t := f(t)(s)$ resp. $f_\nabla(s)(t) := t\nabla s$ for all $s, t \in S$. $\square$

In what follows, we want to construct functions $f : G \to End(G)$ for a group G (or, more generally, functions $f : S \to E$ for a Yearby pair $(E,\Gamma)_S$ w.r.t. S).

Therefore, we need a modified version of proposition 3.4:

Proposition 3.5 Let S, T be sets s.t. $\{\} \neq S \subseteq T$. Let $(E,\Gamma)_T$ be a Yearby pair w.r.t. T. Let $f(S)(S)$ denote the set $\{f(s)(t) \in T \mid s, t \in S\}$.

- Then each function $f : S \to E(\subseteq T^T)$ determines a partial multiplication on S relative to T given by: $\forall s, t \in S: \ s\nabla_f t := f(t)(s)$.
- ∇_f is a binary operation on S iff $f(S)(S) \subseteq S$.

Proof:
The first part is trivial.
For the second part:

"$\Rightarrow$": for a given f, let ∇_f be a binary operation on S, i.e. $\nabla_f : S \times S \to S$. But then $\forall s, t \in S: \ f(s)(t) = t\nabla_f s \in S$.

"$\Leftarrow$": $f(S)(S) \subseteq S \Leftrightarrow \forall s, t \in S: \ f(s)(t) \in S \Leftrightarrow \forall s, t \in S: \ t\nabla_f s \in S$. $\square$

Now we are in position to extend partial multiplications to binary operations:

Proposition 3.6 (extend partial multiplications) [Yea73] Let S, T be *finite* sets s.t. $\{\} \neq S \subseteq T$. Let $\leq_T$ be a fixed ordering on T. On $T \times T$ take $\leq_{T\times T}$, the induced lexicographical ordering. (Then, certainly, each subset of $T \times T$ has a smallest element). Let $(E, \Gamma)_T$ be a Yearby pair w.r.t. T.

Consider a function $f : S \to E$ s.t. $f(S)(S) \nsubseteq S$ i.e. ∇_f is a partial multiplication on S (by proposition 3.5).

1. There exists (at least one) sequence of functions $f = f_1, \ldots, f_n$ and corresponding sequence $S = L_1, \ldots, L_n$ of subsets of T s.t.
   - for $1 \leq i < n$:
     - f_i is a function $f_i : L_i \to E$.
     - $f_i(L_i)(L_i) \nsubseteq L_i$ (i.e. ∇_{f_i} is a partial multiplication on L_i).
     - $L_i \subset L_{i+1}$ (i.e. L_i is a *proper* subset of L_{i+1}).
     - $f_i = f_{i+1}|_{L_i}$ (i.e. f_i is the restriction of f_{i+1} to L_i).
   - $f_n(L_n)(L_n) \subseteq L_n$ (i.e. ∇_{f_n} is a binary operation on L_n by proposition 3.5).
2. If f_n satisfies condition (2.1), i.e. $\forall k, l \in L_n : \; f_n(f_n(k)(l)) = f_n(k) \circ f_n(l)$, then
   - for $1 \leq i < n$:

$$\forall k, l \in L_i \text{ s.t. } f_i(k)(l) \in L_i : \quad f_i(f_i(k)(l)) = f_i(k) \circ f_i(l)) \tag{3.2}$$

Proof:

1. We explicitly construct such a sequence. Consider the k^{th} iteration step:

   case 1: (a) $L_k \subset T$ & $f_k(L_k)(L_k) \subseteq L_k$: we can stop the iteration and are done.
   (b) $L_k = T$: since $E \subseteq T^T$, $f_k(T) \subseteq T$; hence $f_k(T)(T) \subseteq T$ and we are done.

   case 2: $f_k(L_k)(L_k) \nsubseteq L_k$: We construct L_{k+1} and $f_{k+1} : L_{k+1} \to E$ as follows:
   - $S_{k+1} := f_k(L_k)(L_k) - L_k$.
   - $L_{k+1} := L_k \cup S_{k+1}$. (note that this is a disjoint union)
   - define $f_{k+1} : L_{k+1} \to E$ as:
     - $\forall l \in L_k : \; f_{k+1}(l) := f_k(l)$
     - $\forall l \in S_{k+1} : \; f_{k+1}(l) := f_k(s) \circ f_k(t)$ where (s, t) is the smallest pair in $L_k \times L_k$ (w.r.t $\leq_{T\times T}$) s.t. $f_k(s)(t) = l$.

   Then, since $(E, \circ)$ is a semigroup, $f_{k+1} : L_{k+1} \to E$ is well-defined and by construction L_k is a proper subset of L_{k+1} and $f_i = f_{i+1}|_{L_i}$.

   Since in every iteration step at least one element is added to the set L_i and each $L_i \subseteq T$, the iteration must terminate after at most $n := |T| - |S|$ iteration steps.
2. suppose for a fixed i that for $\bar{k}, \bar{l} \in L_i$ condition (3.2) does not hold. Then, since $f_i = f_n|_{L_i}$ the condition does not hold for f_n either. □

Example 3.7 (carrying out an example of the extension procedure) Consider the group $C_7 = \{0,1,2,3,4,5,6\}$ with addition modulo 7. Take the 7 endomorphisms of this group:

$$\begin{aligned}
\alpha_1 &= \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \\
\alpha_2 &= \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 2 & 4 & 6 & 1 & 3 & 5 \end{pmatrix} \\
\alpha_3 &= \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 3 & 6 & 2 & 5 & 1 & 4 \end{pmatrix} \\
\alpha_4 &= \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 4 & 1 & 5 & 2 & 6 & 3 \end{pmatrix} \\
\alpha_5 &= \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 5 & 3 & 1 & 6 & 4 & 2 \end{pmatrix} \\
\alpha_6 &= \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 6 & 5 & 4 & 3 & 2 & 1 \end{pmatrix} = -\mathcal{I} \\
\alpha_7 &= \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix} = \mathcal{I}
\end{aligned}$$

As set T take the elements of C_7: $T := \{0,1,2,3,4,5,6\}$ with the canonical ordering. Let E be the endomorphisms on C_7: $E := \{\alpha_1,\ldots,\alpha_7\}$ and Γ be the automorphisms on C_7: $\Gamma := \{\alpha_2,\ldots,\alpha_7\}$. Then $(E,\Gamma)_T$ is a Yearby pair w.r.t. the set T.
As $S \subseteq T$ take for instance $S := \{0,1\}$ and as function $f : S \to E$ let f be defined by:

$$\begin{aligned}
f(0) &:= \alpha_1 \\
f(1) &:= \alpha_4
\end{aligned}$$

Now we explicitly perform the extension procedure:
$L_1 := S = \{0,1\}; f_1 := f$.
We test $f_1(L_1)(L_1)$:

$$\begin{aligned}
f_1(L_1)(L_1) &= \{f_1(0)(0), f_1(0)(1), f_1(1)(0), \underline{f_1(1)(1)}\} = \\
&\quad \{\alpha_1(0), \alpha_1(1), \alpha_4(0), \alpha_4(1)\} = \{0, \underline{4}\} \not\subseteq L_1.
\end{aligned}$$

Hence we have to perform a first iteration step:
$S_2 := f_1(L_1)(L_1) - L_1 = \{0,4\} - \{0,1\} = \{4\}$.
$L_2 := L_1 \cup S_2 = \{0,1\} \cup \{4\} = \{0,1,4\}$.
$f_2 : \{0,1,4\} \to E$ with

$$\begin{aligned}
f_2(0) &:= f_1(0) = \alpha_1 \\
f_2(1) &:= f_1(1) = \alpha_4 \\
f_2(4) &:= f_1(1) \circ f_1(1) = \alpha_4 \circ \alpha_4 = \alpha_2
\end{aligned}$$

We test $f_2(L_2)(L_2)$:

$$\begin{aligned}
f_2(L_2)(L_2) &= \{f_2(0)(0), f_2(0)(1), f_2(0)(4), \\
&\quad f_2(1)(0), f_2(1)(1), \underline{f_2(1)(4)}, \\
&\quad f_2(4)(0), f_2(4)(1), f_2(4)(4)\} = \{0, 4, \underline{2}, 1\} \not\subseteq L_2.
\end{aligned}$$

Another iteration step is required:
$S_3 := f_2(L_2)(L_2) - L_2 = \{0,4,2,1\} - \{0,1,4\} = \{2\}$.
$L_3 := L_2 \cup S_3 = \{0,1,4\} \cup \{2\} = \{0,1,2,4\}$.
$f_3 : \{0,1,2,4\} \to E$ with

$$\begin{aligned}
f_3(0) &:= f_2(0) = \alpha_1 \\
f_3(1) &:= f_2(1) = \alpha_4 \\
f_3(4) &:= f_2(4) = \alpha_2 \\
f_3(2) &:= f_2(1) \circ f_2(4) = \alpha_4 \circ \alpha_2 = \alpha_7
\end{aligned}$$

We test $f_3(L_3)(L_3)$:

$$\begin{aligned}
f_3(L_3)(L_3) = \; & \{f_3(0)(0), f_3(0)(1), f_3(0)(2), f_3(0)(4), \\
& f_3(1)(0), f_3(1)(1), f_3(1)(2), f_3(1)(4), \\
& f_3(2)(0), f_3(2)(1), f_3(2)(2), f_3(2)(4), \\
& f_3(4)(0), f_3(4)(1), f_3(4)(2), f_3(4)(4)\} = \{0,1,2,4\} \subseteq L_3.
\end{aligned}$$

Hence we are done.

Now, how can we make use of proposition 3.6? (Remember: for a Yearby pair $(E,\Gamma)_T$, we want to compute all valid functions $f : T \to E$).

During the extension process, we can use condition (3.2) (Yearby called this the *compatibility check*) to determine if a function being extended will never ever become a valid function and immediately turn to creating the next function.

We shall summarize the entire extending process (including the compatibility check) in the following

Algorithm 5 (`Extend`)

```
Input:  (E,Γ)_T:  a Yearby pair
        S: a set s.t.  {} ≠ S ⊆ T
        f:  a function f:S → E
Output: f_n, L_n:  the extended function f_n:L_n → E with S ⊆ L_n ⊆ T s.t.
        ∇_{f_n} is an associative binary operation on L_n if f can be
        extended and an error message otherwise

  f_1:=f; L_1:=S; C_0:= {}; k:=1;
  start:
    S_{k+1} := f_k(L_k)(L_k) - L_k;
    L_{k+1} := L_k ∪ S_{k+1};
    C_k := {(s,t) ∈ L_k × L_k  | f_k(s)(t) ∈ L_k } - C_{k-1};
    if not for all (s,t) ∈ C_k:  f_k(f_k(s)(t)) = f_k(s) ∘ f_k(t) then
      return "error, f cannot be extended to a valid function";
    fi;
    if S_{k+1} = {} then
      return f_k, L_k;
```

```
    else
      for c in L_{k+1} do
        if c ∈ L_k then
          f_{k+1}(c) := f_k(c);
        else
          (s,t) := min{(s,t) ∈ L_k × L_k | f_k(s)(t) = c };
          f_{k+1} := f_k(s) ∘ f_k(t);
        fi;
      od;
    fi;
    k := k+1;
  goto start;
```

The GAP source of this function can be found in appendix B.7.

Proposition 3.8 (uniqueness of an extension sequence) Let $(E, \Gamma)_T$ be a Yearby pair, S a set s.t. $\{\} \neq S \subseteq T$, and f a function $f : S \to E$.

Suppose the sequences $f = f_1, \ldots, f_n; S = L_1, \ldots, L_n \subseteq T$ have been constructed by algorithm 5.

1. $Im(f_n)$ (i.e. $(\{f_n(l) \mid l \in L_n\}, \circ)$) is a subsemigroup of E.
2. If the extension of f to f_n is successful (i.e. no error returned) then it is unique.

Proof:

1. Clearly, $\circ$ is associative. It remains only to show that $Im(f_n)$ is closed: for f_n, condition (3.2) must hold: $\forall k, l \in L_n : f_n(f_n(k)(l)) = f_n(k) \circ f_n(l)$. Suppose k, l arbitrary, fixed. Since $f_n(L_n)(L_n) \subseteq L_n$ by construction, $f_n(k)(l)$ equals some element, say $e \in L_n$ and therefore $f_n(k) \circ f_n(l) = f_n(e) \in Im(f_n)$.

2. Suppose that in the i^{th} iteration step, we have two different pairs, $(k_1, l_1), (k_2, l_2) \in L_i \times L_i$ s.t. $f_i(k_1)(l_1) = f_i(k_2)(l_2) = c \notin L_i$ and $f_i(k_1) \circ f_i(l_1) \neq f_i(k_2) \circ f_i(l_2)$ (which means that dependent on the choice of the pair (k_1, l_1) or (k_2, l_2) we would get different extension sequences). But this cannot happen:
   w.l.o.g. choose (k_1, l_1) and construct f_{i+1}:

$$\begin{array}{rcl} & \vdots & \\ f_{i+1}(k_2) & := & f_i(k_2) \\ f_{i+1}(l_2) & := & f_i(l_2) \\ f_{i+1}(c) & := & f_i(k_1) \circ f_i(l_1) \\ & \vdots & \end{array}$$

   Suppose the compatibility check holds for f_{i+1}, then in particular:

$$f_{i+1}(c) = f_{i+1}(f_i(k_2)(l_2)) = f_{i+1}(f_{i+1}(k_2)(l_2)) = f_{i+1}(k_2) \circ f_{i+1}(l_2) = f_i(k_2) \circ f_i(l_2)$$

   But on the other hand, $f_{i+1}(c) = f_i(k_1) \circ f_(l_1)$ by construction, a contradiction to the assumption $f_i(k_1) \circ f_i(l_1) \neq f_i(k_2) \circ f_i(l_2)$. □

We carry out another example, now with the compatibility check included:

Example 3.9 (extension procedure with compatibility check) Take for $(E,\Gamma)_T$ the same settings as in example 3.7.

As $S \subseteq T$ now take for instance $S := \{0,1,2\}$ and as function $f: S \to E$ let f be defined by:

$$\begin{aligned} f(0) &:= \alpha_1 \\ f(1) &:= \alpha_1 \\ f(2) &:= \alpha_5 \end{aligned}$$

Again, we explicitly perform the extension procedure:

$L_1 := S = \{0,1,2\}; f_1 := f$.
We test $f_1(L_1)(L_1)$:

$$\begin{aligned} f_1(L_1)(L_1) &= \{f_1(0)(0), f_1(0)(1), f_1(0)(2), \\ &\quad f_1(1)(0), f_1(1)(1), f_1(1)(2), \\ &\quad f_1(2)(0), \underline{f_1(2)(1)}, \underline{\underline{f_1(2)(2)}}\} = \{0, \underline{5}, \underline{\underline{3}}\} \not\subseteq L_1. \end{aligned}$$

Hence we have to perform a first iteration step:
$S_2 := f_1(L_1)(L_1) - L_1 = \{0,5,3\} - \{0,1,2\} = \{3,5\}$.
$L_2 := L_1 \cup S_2 = \{0,1,2\} \cup \{3,5\} = \{0,1,2,3,5\}$.
We insert the compatibility check:
$C_1 := \{(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0)\}$

$$\begin{aligned} f_1(f_1(0)(0)) = f_1(0) = \alpha_1 = \alpha_1 \circ \alpha_1 = f_1(0) \circ f_1(0) \\ f_1(f_1(0)(1)) = f_1(0) = \alpha_1 = \alpha_1 \circ \alpha_1 = f_1(0) \circ f_1(1) \\ f_1(f_1(0)(2)) = f_1(0) = \alpha_1 = \alpha_1 \circ \alpha_5 = f_1(0) \circ f_1(2) \\ f_1(f_1(1)(0)) = f_1(0) = \alpha_1 = \alpha_1 \circ \alpha_1 = f_1(1) \circ f_1(0) \\ f_1(f_1(1)(1)) = f_1(0) = \alpha_1 = \alpha_1 \circ \alpha_1 = f_1(1) \circ f_1(1) \\ f_1(f_1(1)(2)) = f_1(0) = \alpha_1 = \alpha_1 \circ \alpha_5 = f_1(1) \circ f_1(2) \\ f_1(f_1(2)(0)) = f_1(0) = \alpha_1 = \alpha_5 \circ \alpha_1 = f_1(2) \circ f_1(0) \end{aligned}$$

Since everything fits together, we can continue with constructing f_2:
$f_2 : \{0,1,2,3,5\} \to E$ with

$$\begin{aligned} f_2(0) &:= f_1(0) = \alpha_1 \\ f_2(1) &:= f_1(1) = \alpha_1 \\ f_2(2) &:= f_1(2) = \alpha_5 \\ f_2(3) &:= f_1(2) \circ f_1(2) = \alpha_5 \circ \alpha_5 = \alpha_4 \\ f_2(5) &:= f_1(2) \circ f_1(1) = \alpha_5 \circ \alpha_1 = \alpha_1 \end{aligned}$$

We test $f_2(L_2)(L_2)\ldots$

$$f_2(L_2)(L_2) = \{f_2(0)(0), f_2(0)(1), f_2(0)(2), f_2(0)(3), f_2(0)(5),$$

$$
\begin{aligned}
&f_2(1)(0), f_2(1)(1), f_2(1)(2), f_2(1)(3), f_2(1)(5),\\
&f_2(2)(0), f_2(2)(1), f_2(2)(2), f_2(2)(3), \underline{f_2(2)(5)},\\
&f_2(3)(0), \underline{f_2(3)(1)}, f_2(3)(2), f_2(3)(3), \underline{\underline{f_2(3)(5)}},\\
&f_2(5)(0), f_2(5)(1), f_2(5)(2), f_2(5)(3), f_2(5)(5)\} = \{0, 5, 3, 1, \underline{4}, \underline{\underline{6}}\} \not\subseteq L_2.
\end{aligned}
$$

... and perform another iteration step:
$S_3 := f_2(L_2)(L_2) - L_2 = \{0, 5, 3, 1, 4, 6\} - \{0, 1, 2, 3, 5\} = \{4, 6\}$.
$L_3 := L_2 \cup S_3 = \{0, 1, 2, 3, 5\} \cup \{4, 6\} = \{0, 1, 2, 3, 4, 5, 6\}$.
Compatibility check:

$$
\begin{aligned}
C_2 \quad := \quad & \{(0,0), (0,1), (0,2), (0,3), (0,5),\\
& (1,0), (1,1), (1,2), (1,3), (1,5),\\
& (2,0), (2,1), (2,2), (2,3),\\
& (3,0), (3,2), (3,3),\\
& (5,0), (5,1), (5,2), (5,3), (5,5)\} - C_1 =\\
= \quad & \{(0,3), (0,5), (1,3), (1,5), (2,1), (2,2), (2,3), (3,0),\\
& (3,2), (3,3), (5,0), (5,1), (5,2), (5,3), (5,5)\}
\end{aligned}
$$

$$
\begin{aligned}
f_2(f_2(0)(3)) &= f_2(0) = \alpha_1 = \alpha_1 \circ \alpha_4 = f_2(0) \circ f_2(3)\\
f_2(f_2(0)(5)) &= f_2(0) = \alpha_1 = \alpha_1 \circ \alpha_1 = f_2(0) \circ f_2(5)\\
f_2(f_2(1)(3)) &= f_2(0) = \alpha_1 = \alpha_1 \circ \alpha_4 = f_2(1) \circ f_2(3)\\
f_2(f_2(1)(5)) &= f_2(0) = \alpha_1 = \alpha_1 \circ \alpha_1 = f_2(1) \circ f_2(5)\\
f_2(f_2(2)(1)) &= f_2(5) = \alpha_1 = \alpha_5 \circ \alpha_1 = f_2(2) \circ f_2(1)\\
f_2(f_2(2)(2)) &= f_2(3) = \alpha_4 = \alpha_5 \circ \alpha_5 = f_2(2) \circ f_2(2)\\
f_2(f_2(2)(3)) &= f_2(1) = \alpha_1 \neq \alpha_5 \circ \alpha_4 = f_2(2) \circ f_2(3)
\end{aligned}
$$

Since the compatibility check does not hold, we can terminate the iteration.

Now we shall show how the extension procedure can be used to compute all valid functions:

3.1.2 Computing all valid functions and classifying them

The extension algorithm can be encorporated into a backtracking procedure (depth first search - for this topic we refer to any book on algorithm theory such as [Baa78]) that for a given Yearby pair $(E, \Gamma)_T$ computes all valid functions $\phi : T \to E$.

For a detailed description of this backtracking procedure we refer to [Yea73]. An implementation as GAP function `ValidFunctionsYearby` can be found in appendix B.7. Since recursions are not recommended in GAP, this is an iterative implementation of the procedure.

Recall example 3.2: theoretically, for a set S, the Yearby pair $(\mathcal{T}(S), \mathcal{S}(S))_S$ allows us to compute and classify all associative binary operations on S and for a group G, the Yearby pair $(End(G), Aut(G))_G$ allows us to compute all nearrings on G. As it turned out, the function `ValidFunctionsYearby` proved powerful enough to compute all semigroups of orders 2 to 5 and all nearrings of orders 2 to 15.

However, there remains the problem of classifying all these nearrings and semigroups. Yearby ([Yea73]) put it this way: *The problem of classifying large sets of near rings that have been generated remains the more formidable problem for computation. Difficulties with memory requirements become critical when the set of near rings to classify exceeds 5000.*

In order to optimize performance and memory demands, the classification algorithm has also been implemented in C. The source code (intended for compilation with gcc under a UNIX operating system) can be found in appendix B.9. Nonetheless, the classification problem is still very time consuming. (A matter of days and weeks even on a HP workstation computer).

For low order groups, the largest number of nearrings occurs for D_{12} (namely 562096). In this case the memory demand of the classification program is approx. 13 MBytes, which is quite modest, given nowadays usual memory sizes. (In fact, the classification of the nearrings on D_{12} was performed on a HP workstation with 80 MBytes main memory and took about 25 days).

We shall conclude this chapter with summarizing the results of our computations:

Table 3.1: a tally table of nearrings on low order groups

order	*group G*	*# of nearrings on G*	*# of isomorphism classes of nearrings on G*
2	C_2	3	3
3	C_3	7	5
4	C_4	17	12
	V_4	99	23
5	C_5	29	10
6	C_6	98	60
	S_3	160	39
7	C_7	112	24
8	C_8	350	135
	$C_2 \times C_4$	6982	1159
	$C_2 \times C_2 \times C_2$	99746	834
	D_8	9308	1447
	Q_8	4692	281
9	C_9	1170	222
	$C_3 \times C_3$	8907	264
10	C_{10}	1200	329
	D_{10}	3454	206
11	C_{11}	1312	139
12	C_{12}	5522	1749
	$C_2 \times C_6$	34316	3501
	D_{12}	562096	48137
	A_4	8728	483
	T	6571	824
13	C_{13}	5264	454
14	C_{14}	15761	2716
	D_{14}	71747	1821
15	C_{15}	27998	3817

Table 3.2: a tally table of small semigroups

order	*# of semigroups*	*# of isomorphism classes*
2	8	5
3	113	24
4	3492	188
5	183732	1915

Chapter 4

A Manual to the Implemented Functions

In this chapter, we introduce all functions that are available in the present implementation. For understanding this chapter, it is not necessary to have read the previous chapters of this thesis, however, we recommend that you have read at least the introductory chapter of the GAP manual ([S+94]).

4.1 Transformations

A **transformation** is a mapping with equal source and range, say X. For example, X may be a set or a group. A transformation on X then acts on X by **transforming** each element of X into (precisely one) element of X.

Note that a transformation is just a special case of a mapping. So all GAP functions that work for mappings (see the according chapters of the GAP manual [S+94]) will also work for transformations.

For the following, it is important to keep in mind that in GAP sets are represented by **sorted** lists without holes and duplicates. Throughout this section, let X be a set or a group with n elements. **A transformation on X is uniquely determined by a list of length n without holes and with entries which are integers between 1 and n.**

For example, for the set `X := [1,2,3]`, the list `[1,1,2]` determines the transformation on `X` which transforms `1` into `1`, `2` into `1`, and `3` into `2`.

Analogously, for the cyclic group of order 3: `C3`, with (the uniquely ordered) set of elements `[(),(1,2,3),(1,3,2)]`, the list `[2,3,3]` determines the transformation on `C3` which transforms `()` into `(1,2,3)`, `(1,2,3)` into `(1,3,2)`, and `(1,3,2)` into `(1,3,2)`.

Such a list which on a given set or group uniquely determines a transformation will be called **transformation list** (short **tfl**).

Transformations are created by the constructor functions `Transformation` or `AsTransformation` and they are represented by records that contain all the information about the transformations. The functions described in this section can be found in the file `T.G`.

4.1.1 Transformation

`Transformation(` *obj*, *tfl* `)`
The constructor function `Transformation` returns the transformation determined by the transformation list `tfl` on *obj* where *obj* must be a group or a set.

```
gap> t1:=Transformation([1..3],[1,1,2]);
Transformation on [ 1, 2, 3 ]:
  1 -> 1
  2 -> 1
  3 -> 2

gap> g:=Group((1,2),(3,4));
Group( (1,2), (3,4) )
gap> gt:=Transformation(g,[1,1,2,5]);
Error, Usage: Transformation( <obj>, <tfl> ) where <obj> must be a set
or a group and <tfl> must be a valid transformation list for <obj> in
Transformation( Group( (1,2), (3,4) ), [ 1, 1, 2, 5 ] ) called from
main loop
gap> gt:=Transformation(g,[4,2,2,1]);
Transformation on Group( (1,2), (3,4) ):
  () -> (1,2)(3,4)
  (3,4) -> (3,4)
  (1,2) -> (3,4)
  (1,2)(3,4) -> ()
```

4.1.2 AsTransformation

`AsTransformation(` *map* `)`
The constructor function `AsTransformation` returns the mapping *map* as transformation. Of course, this function can only be applied to mappings with equal source and range, otherwise an error will be signaled.

```
gap> s3:=Group((1,2),(1,2,3));               # this defines S3
Group( (1,2), (1,2,3) )
gap> i:=InnerAutomorphism(s3,(2,3));
InnerAutomorphism( Group( (1,2), (1,2,3) ), (2,3) )
gap> AsTransformation(i);
Transformation on Group( (1,2), (1,2,3) ):
  () -> ()
  (2,3) -> (2,3)
  (1,2) -> (1,3)
  (1,2,3) -> (1,3,2)
  (1,3,2) -> (1,2,3)
  (1,3) -> (1,2)
```

4.1.3 IsTransformation

`IsTransformation(` *obj* `)`
`IsTransformation` returns `true` if the object *obj* is a transformation and `false` otherwise.

```
gap> IsTransformation( [1,1,2] );
false                          # a list is not a transformation
gap> IsTransformation( (1,2,3) );
false                          # a permutation is not a transformation
gap> IsTransformation( t1 );
true
```

4.1.4 IsSetTransformation

`IsSetTransformation(` *obj* `)`
`IsSetTransformation` returns `true` if the object *obj* is a set transformation and `false` otherwise.

```
gap> IsSetTransformation( t1 );
true
gap> g:= Group((1,2),(3,4));
Group( (1,2), (3,4) )
gap> gt:=Transformation(g,[4,2,2,1]);
[ 4, 2, 2, 1 ]
gap> IsSetTransformation( gt );
false
```

4.1.5 IsGroupTransformation

`IsGroupTransformation(` *obj* `)`
`IsGroupTransformation` returns `true` if the object *obj* is a group transformation and `false` otherwise.

```
gap> IsGroupTransformation( t1 );
false
gap> IsGroupTransformation( gt );
true
```

Note that transformations are defined to be either a set transformation or a group transformation.

4.1.6 IdentityTransformation

`IdentityTransformation(` *obj* `)`
`IdentityTransformation` is the counterpart to the GAP standard library function `IdentityMapping`. It returns the identity transformation on *obj* where *obj* must be a group or a set.

```
gap> IdentityTransformation([1..3]);
Transformation on [ 1, 2, 3 ]:
  1 -> 1
  2 -> 2
  3 -> 3

gap> IdentityTransformation(s3);
Transformation on Group( (1,2), (1,2,3) ):
  () -> ()
  (2,3) -> (2,3)
  (1,2) -> (1,2)
  (1,2,3) -> (1,2,3)
  (1,3,2) -> (1,3,2)
  (1,3) -> (1,3)
```

4.1.7 Ker

`Ker(` *t* `)`
For a transformation t on X, the **kernel** of t is defined as an equivalence relation $Ker(t)$ as: $\forall x, y \in X\ (x, y) \in Ker(t)$ iff $t(x) = t(y)$.

`Ker` returns the kernel of the transformation t as a list `l` of lists where each sublist of `l` represents an equivalence class of the equivalence relation `Ker(t)`.

```
gap> t:=Transformation([1..5],[2,3,2,4,4]);
Transformation on [ 1, 2, 3, 4, 5 ]:
  1 -> 2
  2 -> 3
  3 -> 2
  4 -> 4
  5 -> 4

gap> Ker(t);
[ [ 1, 3 ], [ 2 ], [ 4, 5 ] ]
```

4.1.8 Rank

`Rank(` *t* `)`
For a transformation t on X, the **rank** of t is defined as the size of the image of t, i.e. $|\{t(x) \mid x \in X\}|$, or, in GAP language: `Length( Image( t ) )`.

`Rank` returns the rank of the transformation t.

```
gap> t1;
Transformation on [ 1, 2, 3 ]:
  1 -> 1
  2 -> 1
  3 -> 2
```

```
gap> Rank(t1);
2
gap>
gap> gt;
Transformation on Group( (1,2), (3,4) ):
  () -> (1,2)(3,4)
  (3,4) -> (3,4)
  (1,2) -> (3,4)
  (1,2)(3,4) -> ()

gap> Rank(gt);
3
```

4.1.9 Operations for transformations

t1 * *t2*
The product operator * returns the transformation which is obtained from the transformations *t1* and *t2* by composition of *t1* and *t2* (i.e. performing *t1 after t2*). This function works for both set transformations as well as group transformations.

```
gap> t1:=Transformation([1..3],[1,1,2]);
Transformation on [ 1, 2, 3 ]:
  1 -> 1
  2 -> 1
  3 -> 2

gap> t2:=Transformation([1..3],[2,3,3]);
Transformation on [ 1, 2, 3 ]:
  1 -> 2
  2 -> 3
  3 -> 3

gap> t1*t2;
Transformation on [ 1, 2, 3 ]:
  1 -> 1
  2 -> 2
  3 -> 2

gap> t2*t1;
Transformation on [ 1, 2, 3 ]:
  1 -> 2
  2 -> 2
  3 -> 3
```

t1 + *t2*
The add operator + returns the group transformation which is obtained from the group

transformations *t1* and *t2* by pointwise addition of *t1* and *t2*. (Note that in this context addition means performing the GAP operation `p * q` for the corresponding permutations `p` and `q`).

t1 `-` *t2*
The subtract operator `-` returns the group transformation which is obtained from the group transformations *t1* and *t2* by pointwise subtraction of *t1* and *t2*. (Note that in this context subtraction means performing the GAP operation `p * q^-1` for the corresponding permutations `p` and `q`).

Of course, those two functions `+` and `-` work only for group transformations.

```
gap> g:=Group((1,2,3));
Group( (1,2,3) )
gap> gt1:=Transformation(g,[2,3,3]);
Transformation on Group( (1,2,3) ):
  () -> (1,2,3)
  (1,2,3) -> (1,3,2)
  (1,3,2) -> (1,3,2)

gap> gt2:=Transformation(g,[1,3,2]);
Transformation on Group( (1,2,3) ):
  () -> ()
  (1,2,3) -> (1,3,2)
  (1,3,2) -> (1,2,3)

gap> gt1+gt2;
Transformation on Group( (1,2,3) ):
  () -> (1,2,3)
  (1,2,3) -> (1,2,3)
  (1,3,2) -> ()

gap> gt1-gt2;
Transformation on Group( (1,2,3) ):
  () -> (1,2,3)
  (1,2,3) -> ()
  (1,3,2) -> (1,2,3)
```

4.1.10 TransformationPrintLevel

`TransformationPrintLevel(` *string* `)`
In the previous examples, one may have noticed, that transformations are printed out quite detailed. Per default, transformations will appear on the screen in this way.

However, in many cases it will be sufficient (and clearer) to print only the transformation list (**tfl**) of a transformation. This can be achieved by using the function `TransformationPrintLevel` with the string `"short"` or `"s"` as argument. Calling `TransformationPrintLevel` with the argument `"long"` or `"l"` will reset the long printout mode and calling `TransformationPrintLevel` with no argument at all will return the current printout mode.

```
gap> t:=Transformation([1..5],[3,3,2,1,4]);
Transformation on [ 1, 2, 3, 4, 5 ]:
  1 -> 3
  2 -> 3
  3 -> 2
  4 -> 1
  5 -> 4

gap> TransformationPrintLevel();
"long"
gap> TransformationPrintLevel("s");
gap> t;
[ 3, 3, 2, 1, 4 ]
```

4.1.11 Transformation records

As almost all objects in GAP, transformations, too, are representend by records. Such a transformation record has the following components:

`isGeneralMapping`
: this is always `true`, since in particular, any transformation is a general mapping.

`domain`
: the entry of this record field is `Mappings`.

`isMapping`
: this is always `true` since a transformation is in particular a single valued mapping.

`isTransformation`
: always `true` for a transformation.

`isSetTransformation`
: this exists and is set to `true` for set transformations exclusively.

`isGroupTransformation`, `isGroupElement`
: these two exist and are set to `true` for group transformations exclusively.

`elements`
: this record field holds a list of the elements of the source.

`source`, `range`
: both entries contain the same set in case of a set transformation, resp. the same group in case of a group transformation.

`tfl`
: this contains the transformation list which uniquely determines the transformation.

`operations`
: the operations record of the transformation. E.g. `*` or `=`, etc. can be found here.

`image`, `rank`, `ker`
: these are bound and contain image, rank, ker in case they have already been computed for the transformation.

4.2 Transformation Semigroups

Having established transformations and being able to perform the associative operation **composition** (which in GAP is denoted as `*`) with them, the next step is to consider **transformation semigroups**.

All functions described in this section are intended for **finite** transformation semigroups, in particular transformation semigroups on a finite set or group X. A transformation semigroup is created by the constructor function `TransformationSemigroup` and it is represented by a record that contains all the information about the transformation semigroup. The functions described in this section can be found in the source file `SG.G`.

4.2.1 TransformationSemigroup

`TransformationSemigroup(` t_1`, ...,` t_n `)` or
`TransformationSemigroup( [` t_1`, ...,` t_n `] )`
When called in this form, the constructor function `TransformationSemigroup` returns the transformation semigroup generated by the transformations t_1, ..., t_n. There is another way to call this function:

`TransformationSemigroup(` n `)`
If the argument is a positive integer n, `TransformationSemigroup` returns the semigroup of all transformations on the set $\{1, 2, \ldots, n\}$. Since the size of the resulting semigroup is, of course, n^n, the size of the parameter n is limited to a value of 5 (by the variable `MAX_SET_SIZE` in the file `SG.G` - you may increase it if you are adventurous or your computer has a really offensive amount of memory).

```
gap> t1:=Transformation([1..3],[1,1,2]);
Transformation on [ 1, 2, 3 ]:
  1 -> 1
  2 -> 1
  3 -> 2

gap> t2:=Transformation([1..3],[2,3,3]);
Transformation on [ 1, 2, 3 ]:
  1 -> 2
  2 -> 3
  3 -> 3

gap> s:=TransformationSemigroup(t1,t2);
TransformationSemigroup( Transformation on [ 1, 2, 3 ]:
  1 -> 1
  2 -> 1
  3 -> 2
Transformation on [ 1, 2, 3 ]:
  1 -> 2
  2 -> 3
  3 -> 3
```

```
)

gap> TransformationPrintLevel("s");
gap> s27 := TransformationSemigroup( 3 );
TransformationSemigroup( [ 1, 1, 1 ] , [ 1, 1, 2 ] , [ 1, 1, 3 ] ,
[ 1, 2, 1 ] , [ 1, 2, 2 ] , [ 1, 2, 3 ] , [ 1, 3, 1 ] , [ 1, 3, 2 ] ,
[ 1, 3, 3 ] , [ 2, 1, 1 ] , [ 2, 1, 2 ] , [ 2, 1, 3 ] , [ 2, 2, 1 ] ,
[ 2, 2, 2 ] , [ 2, 2, 3 ] , [ 2, 3, 1 ] , [ 2, 3, 2 ] , [ 2, 3, 3 ] ,
[ 3, 1, 1 ] , [ 3, 1, 2 ] , [ 3, 1, 3 ] , [ 3, 2, 1 ] , [ 3, 2, 2 ] ,
[ 3, 2, 3 ] , [ 3, 3, 1 ] , [ 3, 3, 2 ] , [ 3, 3, 3 ]  )
```

4.2.2 IsSemigroup

`IsSemigroup(` *obj* `)`
`IsSemigroup` returns `true` if the object *obj* is a semigroup and `false` otherwise. This function simply checks whether the record component *obj*`.isSemigroup` is bound and is `true`.

```
gap> IsSemigroup( t1 );
false                         # a transformation is not a semigroup
gap> IsSemigroup( Group( (1,2,3) ) );
false                         # a group is not a semigroup
gap> IsSemigroup( s27 );
true
```

4.2.3 IsTransformationSemigroup

`IsTransformationSemigroup(` *obj* `)`
`IsTransformationSemigroup` returns `true` if the object *obj* is a transformation semigroup and `false` otherwise.

```
gap> IsTransformationSemigroup( s27 );
true
```

4.2.4 Elements

`Elements(` *sg* `)`
`Elements` computes the elements of the semigroup *sg*. Note: the GAP standard library dispatcher function `Elements` calls the function *sg*`.operations.Elements` which performs a simple closure algorithm.

```
gap> t1:=Transformation([1..3],[1,1,2]);
[ 1, 1, 2 ]
gap> t2:=Transformation([1..3],[2,3,3]);
[ 2, 3, 3 ]
gap> s := TransformationSemigroup( t1,t2 );
```

```
TransformationSemigroup( [ 1, 1, 2 ] , [ 2, 3, 3 ]  )
gap> Elements( s );
[ [ 1, 1, 1 ] , [ 1, 1, 2 ] , [ 1, 2, 2 ] , [ 2, 2, 2 ] , [ 2, 2, 3 ] ,
  [ 2, 3, 3 ] , [ 3, 3, 3 ]  ]
```

4.2.5 Size

`Size(` *sg* `)`
`Size` returns the number of elements in *sg*.

```
gap> Size( s );
7
```

4.2.6 PrintTable

`PrintTable(` *sg* `)`
`PrintTable` prints the Cayley table of the semigroup *sg*. Note: The dispatcher function `PrintTable` calls the function *sg*`.operations.Table` which performs the actual printing. `PrintTable` has no return value.

```
gap> Table( s );
Let:
s0 := [ 1, 1, 1 ]
s1 := [ 1, 1, 2 ]
s2 := [ 1, 2, 2 ]
s3 := [ 2, 2, 2 ]
s4 := [ 2, 2, 3 ]
s5 := [ 2, 3, 3 ]
s6 := [ 3, 3, 3 ]

  *  | s0 s1 s2 s3 s4 s5 s6
 ---------------------------
  s0 | s0 s0 s0 s0 s0 s0 s0
  s1 | s0 s0 s0 s0 s1 s2 s3
  s2 | s0 s1 s2 s3 s3 s3 s3
  s3 | s3 s3 s3 s3 s3 s3 s3
  s4 | s3 s3 s3 s3 s4 s5 s6
  s5 | s3 s4 s5 s6 s6 s6 s6
  s6 | s6 s6 s6 s6 s6 s6 s6
```

4.2.7 IdempotentElements

`IdempotentElements(` *sg* `)`
An element i of a semigroup $(S, \cdot)$ is called an **idempotent** (element) iff $i \cdot i = i$.

The function `IdempotentElements` returns a list of those elements of the semigroup *sg* that are idempotent. (Note that for a finite semigroup this can never be the empty list).

```
gap> IdempotentElements( s );
[ [ 1, 1, 1 ] , [ 1, 2, 2 ] , [ 2, 2, 2 ] , [ 2, 2, 3 ] , [ 3, 3, 3 ]  ]
```

4.2.8 IsCommutative

`IsCommutative(` *sg* `)`
A semigroup $(S, \cdot)$ is called **commutative** if $\forall a, b \in S: \; a \cdot b = b \cdot a$.

The function `IsCommutative` returns the according value `true` or `false` for a semigroup *sg*.

```
gap> IsCommutative( s );
false
```

4.2.9 Identity

`Identity(` *sg* `)`
An element i of a semigroup $(S, \cdot)$ is called an **identity** iff $\forall s \in S: \; s \cdot i = i \cdot s = s$. Since for two identities, i, i': $i = i \cdot i' = i'$, an identity is unique if it exists.

The function `Identity` returns a list containing as single entry the identity of the semigroup *sg* if it exists or the empty list `[  ]` otherwise.

```
gap> Identity( s );
[  ]
gap> TransformationPrintLevel("s");
gap> tr1 := Transformation([1..3],[1,1,1]);
[ 1, 1, 1 ]
gap> tr2 := Transformation([1..3],[1,2,2]);
[ 1, 2, 2 ]
gap> sg := TransformationSemigroup( tr1, tr2 );
TransformationSemigroup( [ 1, 1, 1 ] , [ 1, 2, 2 ]  )
gap> Elements( sg );
[ [ 1, 1, 1 ] , [ 1, 2, 2 ]  ]
gap> Identity( sg );
[ [ 1, 2, 2 ] ]
```

The last example shows that the identity element of a transformation semigroup on a set X needs not necessarily be the identity transformation on X.

4.2.10 SmallestIdeal

`SmallestIdeal(` *sg* `)`
A subset I of a semigroup $(S, \cdot)$ is defined as an **ideal** of S if $\forall i \in I, s \in S: \; i \cdot s \in I \;\&\; s \cdot i \in I$. An ideal I is called **minimal**, if for any ideal J, $J \subseteq I$ implies $J = I$. If a minimal ideal exists, then it is unique and therefore the **smallest** ideal of S.

The function `SmallestIdeal` returns the smallest ideal of the transformation semigroup *sg*. Note that for a finite semigroup the smallest ideal always exists. (Which is not necessarily true for an arbitrary semigroup).

```
gap> SmallestIdeal( s );
[ [ 1, 1, 1 ] , [ 2, 2, 2 ] , [ 3, 3, 3 ]  ]
```

4.2.11 IsSimple

`IsSimple(` *sg* `)`
A semigroup S is called **simple** if it has no honest ideals, i.e. in case that S is finite the smallest ideal of S equals S itself.

The GAP standard library dispatcher function `IsSimple` calls the function `sg.operations.-IsSimple` which checks if the semigroup *sg* equals its smallest ideal and if so, returns `true` and otherwise `false`.

```
gap> IsSimple( s );
false
gap> c3 := TransformationSemigroup( Transformation([1..3],[2,3,1]));
TransformationSemigroup( [ 2, 3, 1 ]  )
gap> IsSimple( c3 );
true
```

4.2.12 Green

`Green(` *sg*, *string* `)`
Let $(S, \cdot)$ be a semigroup and $a \in S$. The set $a \cdot S^1 := a \cdot S \cup \{a\}$ is called the principal right ideal generated by a. Analogously, $S^1 \cdot a := S \cdot a \cup \{a\}$ is called the principal left ideal generated by a and $S^1 \cdot a \cdot S^1 := S \cdot a \cdot S \cup S \cdot a \cup a \cdot S \cup \{a\}$ is called the principal ideal generated by a.

Now, Green's equivalence relation $\mathcal{L}$ on S is defined as: $(a, b) \in \mathcal{L} :\Leftrightarrow\ S^1 \cdot a = S^1 \cdot b$ i.e. a and b generate the same principal left ideal. Similarly: $(a, b) \in \mathcal{R} :\Leftrightarrow\ a \cdot S^1 = b \cdot S^1$ i.e. a and b generate the same principal right ideal and $(a, b) \in \mathcal{J} :\Leftrightarrow\ S^1 \cdot a \cdot S^1 = S^1 \cdot b \cdot S^1$ i.e. a and b generate the same principal ideal. $\mathcal{H}$ is defined as the intersection of $\mathcal{L}$ and $\mathcal{R}$ and $\mathcal{D}$ is defined as the join of $\mathcal{L}$ and $\mathcal{R}$.

In a finite semigroup, $\mathcal{D} = \mathcal{J}$.

The arguments of the function `Green` are a finite transformation semigroup *sg* and a one character string *string* where *string* must be one of the following: `"L"`, `"R"`, `"D"`, `"J"`, `"H"`. The return value of `Green` is a list of lists of elements of *sg* representing the equivalence classes of the according Green's relation.

```
gap> s;
TransformationSemigroup( [ 1, 1, 2 ] , [ 2, 3, 3 ]  )
gap> Elements( s );
[ [ 1, 1, 1 ] , [ 1, 1, 2 ] , [ 1, 2, 2 ] , [ 2, 2, 2 ] , [ 2, 2, 3 ] ,
  [ 2, 3, 3 ] , [ 3, 3, 3 ]  ]
gap> Green( s, "L" );
[ [ [ 1, 1, 1 ] , [ 2, 2, 2 ] , [ 3, 3, 3 ]  ],
  [ [ 1, 1, 2 ] , [ 2, 2, 3 ]  ], [ [ 1, 2, 2 ] , [ 2, 3, 3 ]  ] ]
```

```
gap> Green( s, "R" );
[ [ [ 1, 1, 1 ]  ], [ [ 1, 1, 2 ] , [ 1, 2, 2 ]  ], [ [ 2, 2, 2 ]  ],
  [ [ 2, 2, 3 ] , [ 2, 3, 3 ]  ], [ [ 3, 3, 3 ]  ] ]
gap> Green( s, "H" );
[ [ [ 1, 1, 1 ]  ], [ [ 1, 1, 2 ]  ], [ [ 1, 2, 2 ]  ], [ [ 2, 2, 2 ]  ],
  [ [ 2, 2, 3 ]  ], [ [ 2, 3, 3 ]  ], [ [ 3, 3, 3 ]  ] ]
gap> Green( s, "D" );
[ [ [ 1, 1, 1 ] , [ 2, 2, 2 ] , [ 3, 3, 3 ]  ],
  [ [ 1, 1, 2 ] , [ 1, 2, 2 ] , [ 2, 2, 3 ] , [ 2, 3, 3 ]  ] ]
```

4.2.13 Rank

`Rank(` *sg* `)`
The **rank** of a transformation semigroup S is defined as the minimal rank of the elements of S, i.e. $\min\{rank(s) \mid s \in S\}$.

The function `Rank` returns the rank of the semigroup *sg*.

```
gap> Rank( s );
1
gap> c3;
TransformationSemigroup( [ 2, 3, 1 ]  )
gap> Rank( c3 );
3
```

4.2.14 LibrarySemigroup

`LibrarySemigroup(` *size*, *num* `)`
The semigroup library contains all semigroups of sizes 1 up to 5, classified into classes of isomorphic semigroups. `LibrarySemigroup` retrieves a representative of an isomorphism class from the semigroup library and returns it as a transformation semigroup. The parameters of `LibrarySemigroup` are two positive integers: *size* must be in $\{1, 2, 3, 4, 5\}$ and indicates the size of the semigroup to be retrieved, *num* indicates the number of an isomorphism class. In dependence of *size*, *num* must be one of the following:

size	*num*
1	$1 \leq num \leq 1$
2	$1 \leq num \leq 5$
3	$1 \leq num \leq 24$
4	$1 \leq num \leq 188$
5	$1 \leq num \leq 1915$

```
gap> ls := LibrarySemigroup( 4, 123 );
TransformationSemigroup( [ 1, 1, 3, 3 ] , [ 1, 2, 3, 4 ] ,
[ 1, 3, 3, 1 ] , [ 1, 4, 3, 2 ]  )
gap> PrintTable( ls );
Let:
```

```
s0 := [ 1, 1, 3, 3 ]
s1 := [ 1, 2, 3, 4 ]
s2 := [ 1, 3, 3, 1 ]
s3 := [ 1, 4, 3, 2 ]


  *  | s0 s1 s2 s3
 ------------------
  s0 | s0 s0 s2 s2
  s1 | s0 s1 s2 s3
  s2 | s0 s2 s2 s0
  s3 | s0 s3 s2 s1
```

4.2.15 Transformation semigroup records

Transformation Semigroups are implemented as records. Such a transformation semigroup record has the following components:

`isDomain`, `isSemigroup`
: these two are always `true` for a transformation semigroup.

`isTransformationSemigroup`
: this is bound and `true` only for transformation semigroups.

`generators`
: this holds the set of generators of a transformation semigroup.

`multiplication`
: this record field contains a function that represents the binary operation of the semigroup that can be performed on the elements of the semigroup. For transformation semigroups this equals of course, composition. Example:

```
gap> elms := Elements( s );
[ [ 1, 1, 1 ] , [ 1, 1, 2 ] , [ 1, 2, 2 ] , [ 2, 2, 2 ] ,
  [ 2, 2, 3 ] , [ 2, 3, 3 ] , [ 3, 3, 3 ]  ]
gap> s.multiplication( elms[5], elms[2] );
[ 2, 2, 2 ]
```

`operations`
: this is the operations record of the semigroup.

`size`, `elements`, `rank`, `smallestIdeal`, `IsFinite`, `identity`
: these entries become bound if the according functions have been performed on the semigroup.

`GreenL`, `GreenR`, `GreenD`, `GreenJ`, `GreenH`
: these are entries according to calls of the function `Green` with the corresponding parameters.

4.3 Nearrings

In section 4.1 we introduced transformations on sets and groups. We used set transformations together with composition `*` to construct transformation semigroups in section 4.2. In section 4.1 we also introduced the operation of pointwise addition `+` for group transformations. Now we are able to use these group transformations together with pointwise addition `+` and composition `*` to construct (right) nearrings.

A **(right) nearring** is a nonempty set N together with two binary operations on N, $+$ and $\cdot$ s.t. $(N, +)$ is a group, $(N, \cdot)$ is a semigroup, and $\cdot$ is right distributive over $+$, i.e. $\forall n_1, n_2, n_3 \in N : (n_1 + n_2) \cdot n_3 = n_1 \cdot n_3 + n_2 \cdot n_3$.

There are three possibilities to get a nearring in GAP: the constructor function `Nearring` can be used in two different ways or a nearring can be extracted from the nearrings library by using the function `LibraryNearring`. All functions described here were programmed for permutation groups and they also work fine with them; other types of groups (such as AG groups) are not supported.

Nearrings are represented by records that contain the necessary information to identify them and to do computations with them. The functions described in this section can be found in the source file `NR.G`.

4.3.1 IsNrMultiplication

`IsNrMultiplication(` G, *mul* `)`
The arguments of the function `IsNrMultiplication` are a permutation group G and a GAP function *mul* which has two arguments `x` and `y` which must both be elements of the group G and returns an element `z` of G s.t. *mul* defines a binary operation on G.

`IsNrMultiplication` returns `true` (`false`) if *mul* is (is not) a nearring multiplication on G i.e. it checks whether it is well-defined, associative and right distributive over the group operation of G.

```
gap> g := Group( (1,2), (1,2,3) );
Group( (1,2), (1,2,3) )
gap> mul_r := function(x,y) return x; end;
function ( x, y ) ... end
gap> IsNrMultiplication( g, mul_r );
true
gap> mul_l := function( x, y ) return y; end;
function ( x, y ) ... end
gap> IsNrMultiplication( g, mul_l );
specified multiplication is not right distributive.
false
```

4.3.2 Nearring

`Nearring(` G, *mul* `)`
In this form the constructor function `Nearring` returns the nearring defined by the permutation group G and the nearring multiplication *mul*. (For a detailed explanation of *mul*

see 4.3.1). `Nearring` calls `IsNrMultiplication` in order to make sure that *mul* is really a nearring multiplication.

```
gap> g := Group( (1,2,3) );
Group( (1,2,3) )
gap> mul_r := function( x, y ) return x; end;
function ( x, y ) ... end
gap> n := Nearring( g, mul_r );
Nearring( Group( (1,2,3) ), multiplication( x, y ) )
gap> PrintTable( n );
Let:
n0 := ()
n1 := (1,2,3)
n2 := (1,3,2)

  +  | n0 n1 n2
 ---------------
  n0 | n0 n1 n2
  n1 | n1 n2 n0
  n2 | n2 n0 n1

  *  | n0 n1 n2
 ---------------
  n0 | n0 n0 n0
  n1 | n1 n1 n1
  n2 | n2 n2 n2
```

`Nearring(` t_1`, ...,` t_n `)`
`Nearring( [`t_1`, ...,` t_n`] )`
In this form the constructor function `Nearring` returns the nearring generated by the group transformations $t_1, \ldots, t_n$. All of them must be transformations on the same permutation group.

Note that `Nearring` allows also a list of group transformations as argument, which makes it possible to call `Nearrings` e.g. with a list of endomorphisms generated by the function `Endomorphisms` (see 4.4.2), which for a group G allows to compute $E(G)$; `Nearring` called with the list of all inner automorphisms of a group G would return $I(G)$.

```
gap> t := Transformation( Group( (1,2) ), [2,1] );
Transformation on Group( (1,2) ):
  () -> (1,2)
  (1,2) -> ()

gap> n := Nearring( t );
TransformationNearring( Transformation on Group( (1,2) ):
  () -> (1,2)
  (1,2) -> ()
 )
gap> PrintTable( n );
```

```
Step 1: Building sums of full transformations
Step 2: Multiplying full transformations
Step 1: Building sums of full transformations
Let:
n0 := Transformation on Group( (1,2) ):
  () -> ()
  (1,2) -> ()

n1 := Transformation on Group( (1,2) ):
  () -> ()
  (1,2) -> (1,2)

n2 := Transformation on Group( (1,2) ):
  () -> (1,2)
  (1,2) -> ()

n3 := Transformation on Group( (1,2) ):
  () -> (1,2)
  (1,2) -> (1,2)


  +  | n0 n1 n2 n3
 ------------------
  n0 | n0 n1 n2 n3
  n1 | n1 n0 n3 n2
  n2 | n2 n3 n0 n1
  n3 | n3 n2 n1 n0

  *  | n0 n1 n2 n3
 ------------------
  n0 | n0 n0 n0 n0
  n1 | n0 n1 n2 n3
  n2 | n3 n2 n1 n0
  n3 | n3 n3 n3 n3

gap> g := Group( (1,2), (1,2,3) );     # define S3
Group( (1,2), (1,2,3) )
gap> TransformationPrintLevel( "s" );;
gap> e := Endomorphisms( g );
[ [ 1, 1, 1, 1, 1, 1 ] , [ 1, 2, 2, 1, 1, 2 ] , [ 1, 2, 6, 5, 4, 3 ] ,
  [ 1, 3, 2, 5, 4, 6 ] , [ 1, 3, 3, 1, 1, 3 ] , [ 1, 3, 6, 4, 5, 2 ] ,
  [ 1, 6, 2, 4, 5, 3 ] , [ 1, 6, 3, 5, 4, 2 ] , [ 1, 6, 6, 1, 1, 6 ] ,
  [ 1, 2, 3, 4, 5, 6 ]  ]
gap> nr := Nearring( e ); # the nr generated by the endomorphisms of S3
TransformationNearring( [ 1, 1, 1, 1, 1, 1 ] , [ 1, 2, 2, 1, 1, 2 ] ,
[ 1, 2, 3, 4, 5, 6 ] , [ 1, 2, 6, 5, 4, 3 ] , [ 1, 3, 2, 5, 4, 6 ] ,
[ 1, 3, 3, 1, 1, 3 ] , [ 1, 3, 6, 4, 5, 2 ] , [ 1, 6, 2, 4, 5, 3 ] ,
[ 1, 6, 3, 5, 4, 2 ] , [ 1, 6, 6, 1, 1, 6 ]  )
```

```
gap> Size( nr );
Step 1: Building sums of full transformations
Step 2: Multiplying full transformations
54
```

4.3.3 IsNearring

`IsNearring(` *obj* `)`
`IsNearring` returns `true` if the object *obj* is a nearring and `false` otherwise. This function simply checks if the record component *obj*`.isNearring` is bound to the value `true`.

```
gap> n := LibraryNearring( "C3", 4 );
Nearring( C3, multiplication( x, y ) )
gap> IsNearring( n );
true
gap> IsNearring( nr );
true
gap> IsNearring( Integers );
false           # the Integers are a ring record not a nearring record
```

4.3.4 IsTransformationNearring

`IsTransformationNearring(` *obj* `)`
`IsTransformationNearring` returns `true` if the object *obj* is a transformation nearring and `false` otherwise. `IsTransformationNearring` simply checks if the record component *obj*`.isTransformationNearring` is bound to `true`.

```
gap> IsTransformationNearring( nr );
true
gap> IsTransformationNearring( n );
false
```

4.3.5 LibraryNearring

`LibraryNearring(` *grp_name*, *num* `)`
`LibraryNearring` retrieves a nearring from the nearrings library files. *grp_name* must be one of the following strings indicating the name of the according group: `"C2"`, `"C3"`, `"C4"`, `"V4"`, `"C5"`, `"C6"`, `"S3"`, `"C7"`, `"C8"`, `"C2xC4"`, `"C2xC2xC2"`, `"D8"`, `"Q8"`, `"C9"`, `"C3xC3"`, `"C10"`, `"D10"`, `"C11"`, `"C12"`, `"C2xC6"`, `"D12"`, `"A4"`, `"T"`, `"C13"`, `"C14"`, `"D14"`, `"C15"`, *num* must be an integer which indicates the number of the class of nearrings on the specified group. In dependence of *grp_name*, *num* must be one of the following:

grp_name	*num*	*grp_name*	*num*
"C2"	$1 \leq num \leq 3$	"C3xC3"	$1 \leq num \leq 264$
"C3"	$1 \leq num \leq 5$	"C10"	$1 \leq num \leq 329$
"C4"	$1 \leq num \leq 12$	"D10"	$1 \leq num \leq 206$
"V4"	$1 \leq num \leq 23$	"C11"	$1 \leq num \leq 139$
"C5"	$1 \leq num \leq 10$	"C12"	$1 \leq num \leq 1749$
"C6"	$1 \leq num \leq 60$	"C2xC6"	$1 \leq num \leq 3501$
"S3"	$1 \leq num \leq 39$	"D12"	$1 \leq num \leq 48137$
"C7"	$1 \leq num \leq 24$	"A4"	$1 \leq num \leq 483$
"C8"	$1 \leq num \leq 135$	"T"	$1 \leq num \leq 824$
"C2xC4"	$1 \leq num \leq 1159$	"C13"	$1 \leq num \leq 454$
"C2xC2xC2"	$1 \leq num \leq 834$	"C14"	$1 \leq num \leq 2716$
"D8"	$1 \leq num \leq 1447$	"D14"	$1 \leq num \leq 1821$
"Q8"	$1 \leq num \leq 281$	"C15"	$1 \leq num \leq 3817$
"C9"	$1 \leq num \leq 222$		

(*Remark:* due to the large number of nearrings on D_{12}, make sure that you have enough main memory - say at least 32 MB - available if you want to get a library nearring on D_{12}).

```
gap> n := LibraryNearring( "V4", 13 );
Nearring( V4, multiplication( x, y ) )
```

4.3.6 PrintTable

`PrintTable(` *nr* `)`
`PrintTable` prints the additive and multiplicative Cayley tables of the nearring *nr*. This function works the same way as for semigroups; the dispatcher function `PrintTable` calls *nr*`.operations.Table` which performs the actual printing.

```
gap> PrintTable( LibraryNearring( "V4", 22 ) );
Let:
n0 := ()
n1 := (3,4)
n2 := (1,2)
n3 := (1,2)(3,4)

  +  | n0 n1 n2 n3
 ------------------
  n0 | n0 n1 n2 n3
  n1 | n1 n0 n3 n2
  n2 | n2 n3 n0 n1
  n3 | n3 n2 n1 n0

  *  | n0 n1 n2 n3
 ------------------
  n0 | n0 n0 n0 n0
  n1 | n0 n1 n2 n3
  n2 | n2 n2 n2 n2
  n3 | n2 n3 n0 n1
```

4.3.7 Elements

`Elements(` *nr* `)`
The function `Elements` computes the elements of the nearring *nr*. As for semigroups the GAP standard library dispatcher function `Elements` calls *nr*`.operations.Elements` which simply returns the elements of *nr*`.group` if *nr* is not a transformation nearring or - if *nr* is a transformation nearring - performs a simple closure algorithm and returns a set of transformations which are the elements of *nr*.

```
gap> TransformationPrintLevel( "s" );;
gap> t := Transformation( Group( (1,2) ), [2,1] );
[ 2, 1 ]
gap> Elements( Nearring( t ) );
Step 1: Building sums of full transformations
Step 2: Multiplying full transformations
Step 1: Building sums of full transformations
[ [ 1, 1 ] , [ 1, 2 ] , [ 2, 1 ] , [ 2, 2 ]  ]
gap>
gap> Elements( LibraryNearring( "C3", 4 ) );
[ (), (1,2,3), (1,3,2) ]
```

4.3.8 Size

`Size(` *nr* `)`
`Size` returns the number of elements in the nearring *nr*.

```
gap> Size( LibraryNearring( "C3", 4 ) );
3
```

4.3.9 Endomorphisms

`Endomorphisms(` *nr* `)`
`Endomorphisms` computes all the endomorphisms of the nearring *nr*. The endomorphisms are returned as a list of transformations. In fact, the returned list contains those endomorphisms of *nr*`.group` which are also endomorphisms of the nearring *nr*.

```
gap> TransformationPrintLevel( "s" );;
gap> t := Transformation( Group( (1,2) ), [2,1] );
[ 2, 1 ]
gap> nr := Nearring( t );
TransformationNearring( [ 2, 1 ]  )
gap> Endomorphisms( nr );
Step 1: Building sums of full transformations
Step 2: Multiplying full transformations
Step 1: Building sums of full transformations
[ [ 1, 1, 1, 1 ] , [ 1, 2, 2, 1 ] , [ 1, 2, 3, 4 ]  ]
```

4.3.10 Automorphisms

`Automorphisms(` *nr* `)`

`Automorphisms` computes all the automorphisms of the nearring *nr*. The automorphisms are returned as a list of transformations. In fact, the returned list contains those automorphisms of *nr*.`group` which are also automorphisms of the nearring *nr*.

```
gap> TransformationPrintLevel( "s" );;
gap> t := Transformation( Group( (1,2) ), [2,1] );
[ 2, 1 ]
gap> nr := Nearring( t );
TransformationNearring( [ 2, 1 ]  )
gap> Automorphisms( nr );
Step 1: Building sums of full transformations
Step 2: Multiplying full transformations
Step 1: Building sums of full transformations
[ [ 1, 2, 3, 4 ]  ]
```

4.3.11 FindGroup

`FindGroup(` *nr* `)`

For a transformation nearring *nr*, this function computes the additive group of *nr* as a GAP permutation group and stores it in the record component *nr*.`group`.

```
gap> t := Transformation( Group( (1,2) ), [2,1] );
[ 2, 1 ]
gap> n := Nearring( t );
TransformationNearring( [ 2, 1 ]  )
gap> FindGroup( n );
Step 1: Building sums of full transformations
Step 2: Multiplying full transformations
Step 1: Building sums of full transformations
[ (), (1,2)(3,4), (1,3)(2,4), (1,4)(2,3) ]
gap> Elements( n );
[ [ 1, 1 ] , [ 1, 2 ] , [ 2, 1 ] , [ 2, 2 ]  ]
```

4.3.12 NearringIdeals

`NearringIdeals(` *nr* `)`
`NearringIdeals(` *nr*, `"l" )`
`NearringIdeals(` *nr*, `"r" )`

`NearringIdeals` computes all (left) (right) ideals of the nearring *nr*. The return value is a list of subgroups of the additive group of *nr* representing the according ideals. In case that *nr* is a transformation nearring, `FindGroup` is used to determine the additive group of *nr* as a permutation group. If the optional parameters `"l"` or `"r"` are passed, all left resp. right ideals are computed.

```
gap> n := LibraryNearring( "V4", 11 );
Nearring( V4, multiplication( x, y ) )
gap> NearringIdeals( n );
[ Subgroup( V4, [  ] ), Subgroup( V4, [ (3,4) ] ), V4 ]
gap> NearringIdeals( n, "r" );
[ Subgroup( V4, [  ] ), Subgroup( V4, [ (3,4) ] ), V4 ]
gap> NearringIdeals( n, "l" );
[ Subgroup( V4, [  ] ), Subgroup( V4, [ (3,4) ] ),
  Subgroup( V4, [ (1,2) ] ), Subgroup( V4, [ (1,2)(3,4) ] ), V4 ]
```

4.3.13 InvariantSubnearrings

`InvariantSubnearrings(` *nr* `)`
A subnearring $(M,+,\cdot)$ of a nearring $(N,+,\cdot)$ is called an **invariant subnearring** if both, $M\cdot N\subseteq M$ and $N\cdot M\subseteq M$.

The function `InvariantSubnearrings` computes all invariant subnearrings of the nearring *nr*. The function returns a list of nearrings representing the according invariant subnearrings. In case that *nr* is a transformation nearring, `FindGroup` is used to determine the additive group of *nr* as a permutation group.

```
gap> InvariantSubnearrings( LibraryNearring( "V4" , 22 ) );
[ Nearring( Subgroup( V4, [ (1,2) ] ), multiplication( x, y ) ),
  Nearring( V4, multiplication( x, y ) ) ]
```

4.3.14 Subnearrings

`Subnearrings(` *nr* `)`
The function `Subnearrings` computes all subnearrings of the nearring *nr*. The function returns a list of nearrings representing the according subnearrings. In case that *nr* is a transformation nearring, `FindGroup` is used to determine the additive group of *nr* as a permutation group.

```
gap> Subnearrings( LibraryNearring( "V4" , 22 ) );
[ Nearring( Subgroup( V4, [  ] ), multiplication( x, y ) ),
  Nearring( Subgroup( V4, [ (3,4) ] ), multiplication( x, y ) ),
  Nearring( Subgroup( V4, [ (1,2) ] ), multiplication( x, y ) ),
  Nearring( V4, multiplication( x, y ) ) ]
```

4.3.15 Identity

`Identity(` *nr* `)`
The function `Identity` returns a list containing the identity of the multiplicative semigroup of the nearring *nr* if it exists and the empty list `[  ]` otherwise.

```
gap> Identity( LibraryNearring( "V4", 22 ) );
[ (3,4) ]
```

4.3.16 Distributors

`Distributors(` *nr* `)`
An element x of a nearring $(N, +, \cdot)$ is called a **distributor** if $x = n_1 \cdot (n_2 + n_3) - (n_1 \cdot n_2 + n_1 \cdot n_3)$ for some elements n_1, n_2, n_3 of N.

The function `Distributors` returns a list containing the distributors of the nearring *nr*.

```
gap> Distributors( LibraryNearring( "S3", 19 ) );
[ (), (1,2,3), (1,3,2) ]
```

4.3.17 DistributiveElements

`DistributiveElements(` *nr* `)`
An element d of a right nearring $(N, +, \cdot)$ is called a **distributive element** if it is also left distributive over all elements, i.e. $\forall n_1, n_2 \in N : \; d \cdot (n_1 + n_2) = d \cdot n_1 + d \cdot n_2$.

The function `DistributiveElements` returns a list containing the distributive elements of the nearring *nr*.

```
gap> DistributiveElements( LibraryNearring( "S3", 25 ) );
[ (), (1,2,3), (1,3,2) ]
```

4.3.18 IsDistributiveNearring

`IsDistributiveNearring(` *nr* `)`
A right nearring N is called **distributive nearring** if its multiplication is also left distributive.

The function `IsDistributiveNearring` simply checks if all elements are distributive and returns the according boolean value `true` or `false`.

```
gap> IsDistributiveNearring( LibraryNearring( "S3", 25 ) );
false
```

4.3.19 ZeroSymmetricElements

`ZeroSymmetricElements(` *nr* `)`
Let $(N, +, \cdot)$ be a right nearring and denote by 0 the neutral element of $(N, +)$. An element n of N is called a **zero-symmetric element** if $n \cdot 0 = 0$.

Remark: note that in a **right** nearring $0 \cdot n = 0$ is true for all elements n.

The function `ZeroSymmetricElements` returns a list containing the zero-symmetric elements of the nearring *nr*.

```
gap> ZeroSymmetricElements( LibraryNearring( "S3", 25 ) );
[ (), (1,2,3), (1,3,2) ]
```

4.3.20 IsAbstractAffineNearring

`IsAbstractAffineNearring(` *nr* `)`
A right nearring N is called **abstract affine** if its additive group is abelian and its zero-symmetric elements are exactly its distributive elements.

The function `IsAbstractAffineNearring` returns the according boolean value `true` or `false`.

```
gap> IsAbstractAffineNearring( LibraryNearring( "S3", 25 ) );
false
```

4.3.21 IdempotentElements

`IdempotentElements(` *nr* `)`
The function `DistributiveElements` returns a list containing the idempotent elements of the multiplicative semigroup of the nearring *nr*.

```
gap> IdempotentElements( LibraryNearring( "S3", 25 ) );
[ (), (2,3) ]
```

4.3.22 IsBooleanNearring

`IsBooleanNearring(` *nr* `)`
A right nearring N is called **boolean** if all its elements are idempotent with respect to multiplication.

The function `IsBooleanNearring` simply checks if all elements are idempotent and returns the according boolean value `true` or `false`.

```
gap> IsBooleanNearring( LibraryNearring( "S3", 25 ) );
false
```

4.3.23 NilpotentElements

`NilpotentElements(` *nr* `)`
Let $(N, +, \cdot)$ be a nearring with zero 0. An element n of N is called **nilpotent** if there is a positive integer k such that $n^k = 0$.

The function `NilpotentElements` returns a list of sublists of length 2 where the first entry is a nilpotent element n and the second entry is the smallest k such that $n^k = 0$.

```
gap> NilpotentElements( LibraryNearring( "V4", 4 ) );
[ [ (), 1 ], [ (3,4), 2 ], [ (1,2), 3 ], [ (1,2)(3,4), 3 ] ]
```

4.3.24 IsNilNearring

`IsNilNearring(` *nr* `)`
A nearring N is called **nil** if all its elements are nilpotent.

The function `IsNilNearring` checks if all elements are nilpotent and returns the according boolean value `true` or `false`.

```
gap> IsNilNearring( LibraryNearring( "V4", 4 ) );
true
```

4.3.25 IsNilpotentNearring

`IsNilpotentNearring(` *nr* `)`
A nearring N is called **nilpotent** if there is a positive integer k, s.t. $N^k = \{0\}$.

The function `IsNilpotentNearring` tests if the nearring *nr* is nilpotent and returns the according boolean value `true` or `false`.

```
gap> IsNilpotentNearring( LibraryNearring( "V4", 4 ) );
true
```

4.3.26 IsNilpotentFreeNearring

`IsNilpotentFreeNearring(` *nr* `)`
A nearring N is called **nilpotent free** if its only nilpotent element is 0.

The function `IsNilpotentFreeNearring` checks if 0 is the only nilpotent and returns the according boolean value `true` or `false`.

```
gap> IsNilpotentFreeNearring( LibraryNearring( "V4", 22 ) );
true
```

4.3.27 IsCommutative

`IsCommutative(` *nr* `)`
A nearring $(N, +, \cdot)$ is called **commutative** if its multiplicative semigroup is commutative.

The function `IsCommutative` returns the according value `true` or `false`.

```
gap> IsCommutative( LibraryNearring( "C15", 1235 ) );
false
gap> IsCommutative( LibraryNearring( "V4", 13 ) );
true
```

4.3.28 IsDgNearring

`IsDgNearring(` *nr* `)`
A nearring $(N,+,\cdot)$ is called **distributively generated (d.g.)** if $(N,+)$ is generated additively by the distributive elements of the nearring.

The function `IsDgNearring` returns the according value `true` or `false` for a nearring *nr*.

```
gap> IsDgNearring( LibraryNearring( "S3", 25 ) );
false
gap> IsDgNearring( LibraryNearring( "S3", 1 ) );
true
```

4.3.29 IsIntegralNearring

`IsIntegralNearring(` *nr* `)`
A nearring $(N,+,\cdot)$ with zero element 0 is called **integral** if it has no zero divisors, i.e. the condition $\forall n_1, n_2 :\ n_1 \cdot n_2 = 0 \Rightarrow n_1 = 0 \vee n_2 = 0$ holds.

The function `IsIntegralNearring` returns the according value `true` or `false` for a nearring *nr*.

```
gap> IsIntegralNearring( LibraryNearring( "S3", 24 ) );
true
gap> IsIntegralNearring( LibraryNearring( "S3", 25 ) );
false
```

4.3.30 IsPrimeNearring

`IsPrimeNearring(` *nr* `)`
A nearring $(N,+,\cdot)$ with zero element 0 is called **prime** if the ideal $\{0\}$ is a prime ideal.

The function `IsPrimeNearring` checks if *nr* is a prime nearring by using the condition *for all non-zero ideals* $I, J :\ I \cdot J \neq \{0\}$ and returns the according value `true` or `false`.

```
gap> IsPrimeNearring( LibraryNearring( "V4", 11 ) );
false
```

4.3.31 QuasiregularElements

`QuasiregularElements(` *nr* `)`
Let $(N,+,\cdot)$ be a right nearring. For an element $z \in N$, denote the left ideal generated by the set $\{n - n \cdot z | n \in N\}$ by L_z. An element z of N is called **quasiregular** if $z \in L_z$.

The function `QuasiregularElements` returns a list of all quasiregular elements of a nearring *nr*.

```
gap> QuasiregularElements( LibraryNearring( "V4", 4 ) );
[ (), (3,4), (1,2), (1,2)(3,4) ]
```

4.3.32 IsQuasiregularNearring

`IsQuasiregularNearring(` *nr* `)`
A nearring N is called **quasiregular** if all its elements are quasiregular.

The function `IsQuasiregularNearring` simply checks if all elements of the nearring *nr* are quasiregular and returns the according boolean value `true` or `false`.

```
gap> IsQuasiregularNearring( LibraryNearring( "V4", 4 ) );
true
```

4.3.33 RegularElements

`RegularElements(` *nr* `)`
Let $(N, +, \cdot)$ be a nearring. An element n of N is called **regular** if there is an element x such that $n \cdot x \cdot n = n$.

The function `RegularElements` returns a list of all regular elements of a nearring *nr*.

```
gap> RegularElements( LibraryNearring( "V4", 4 ) );
[ () ]
```

4.3.34 IsRegularNearring

`IsRegularNearring(` *nr* `)`
A nearring N is called **regular** if all its elements are regular.

The function `IsRegularNearring` simply checks if all elements of the nearring *nr* are regular and returns the according boolean value `true` or `false`.

```
gap> IsRegularNearring( LibraryNearring( "V4", 4 ) );
false
```

4.3.35 IsPlanarNearring

`IsPlanarNearring(` *nr* `)`
Let $(N, +, \cdot)$ be a right nearring. For $a, b \in N$ define the equivalence relation $\equiv$ by $a \equiv b :\Leftrightarrow \forall n \in N : \; n \cdot a = n \cdot b$. If $a \equiv b$ then a and b are called **equivalent multipliers**. A nearring N is called **planar** if $|N/_{\equiv}| \geq 3$ and if every equation of the form $x \cdot a = x \cdot b + c$ has a unique solution for two non equivalent multipliers a and b.

The function `IsPlanarNearring` returns the according value `true` or `false` for a nearring *nr*.

This function works only for library nearrings, i.e. nearrings which are constructed by using the function `LibraryNearring`.

```
gap> IsPlanarNearring( LibraryNearring( "V4", 22 ) );
false
```

4.3.36 IsNearfield

`IsNearfield(` *nr* `)`
Let $(N,+,\cdot)$ be a nearring with zero 0 and denote by N^* the set $N-\{0\}$. N is a **nearfield** if $(N^*,\cdot)$ is a group.

The function `IsNearfield` tests if *nr* has an identity and if every non-zero element has a multiplicative inverse and returns the according value `true` or `false`.

```
gap> IsNearfield( LibraryNearring( "V4", 16 ) );
true
```

4.3.37 LibraryNearringInfo

`LibraryNearringInfo(` *group_name*, *list*, *string* `)`
This function provides information about the specified library nearrings in a way similar to how nearrings are presented in the appendix of [Pil83]. The parameter *group_name* specifies the name of a group; valid names are exactly those which are also valid for the function `LibraryNearrings` (cf. section 4.3.5).

The parameter *list* must be a non-empty list of integers defining the classes of nearrings to be printed. Naturally, these integers must all fit into the ranges described in section 4.3.5 for the according groups.

The third parameter *string* is optional. *string* must be a string consisting of one or more (or all) of the following characters: `l`, `m`, `i`, `v`, `s`, `e`, `a`. Per default, (i.e. if this parameter is not specified), the output is minimal. According to each specified character, the following is added:

`c`: print the meaning of the letters used in the output.

`m`: print the multiplication tables.

`i`: list the ideals.

`l`: list the left ideals.

`r`: list the right ideals.

`v`: list the invariant subnearrings.

`s`: list the subnearrings.

`e`: list the nearring endomorphisms.

`a`: list the nearring automorphisms.

Examples:

`LibraryNearringInfo( "C3", [ 3 ], "lmivsea" )` will print all available information about the third class of nearrings on the group C_3.

`LibraryNearringInfo( "C4", [ 1..12 ] )` will provide a minimal output for all classes of nearrings on C_4.

`LibraryNearringInfo( "S3", [ 5, 18, 21 ], "mi" )` will print the minimal information plus the multiplication tables plus the ideals for the classes 5, 18, and 21 of nearrings on the group S_3.

4.3.38 Nearring records

The record of a nearring has the following components:

`isDomain`, `isNearring`
: these two are always `true` for a nearring.

`isTransformationNearring`
: this is bound and `true` only for transformation nearrings (i.e. those nearrings that are generated by group transformations by using the constructor function `Nearring` in the second form).

`generators`
: this is bound only for a transformation nearring and holds the set of generators of the transformation nearring.

`group`
: this component holds the additive group of the nearring as permutation group.

`addition`, `subtraction`, `multiplication`
: these record fields contain the functions that represent the binary operations that can be performed with the elements of the nearring on the additive group of the nearring (addition, subtraction) resp. on the multiplicative semigroup of the nearring (multiplication)

```
gap> nr := Nearring( Transformation( Group( (1,2) ) , [2,1] ) );
TransformationNearring( [ 2, 1 ]  )
gap> e := Elements(nr);
Step 1: Building sums of full transformations
Step 2: Multiplying full transformations
Step 1: Building sums of full transformations
[ [ 1, 1 ] , [ 1, 2 ] , [ 2, 1 ] , [ 2, 2 ]  ]
gap> nr.addition( e[2], e[3] );
[ 2, 2 ]
gap> nr.subtraction( e[2], e[3] );
[ 2, 2 ]
gap> nr.multiplication( e[2], e[4] );
[ 2, 2 ]
gap> nr.multiplication( e[2], e[3] );
[ 2, 1 ]
```

`operations`
: this is the operations record of the nearring.

`size`, `elements`, `endomorphisms`, `automorphisms`
: these entries become bound if the according functions have been performed on the nearring.

4.4 Supportive Functions for Groups

In order to support nearring calculations, a few functions for groups had to be added to the standard GAP group library functions.

The functions described here can be found in the source file `G.G`.

4.4.1 PrintTable

`PrintTable(` *group* `)`
`PrintTable` prints the Cayley table of the group *group*. This function works the same way as for semigroups and nearrings: the dispatcher function `PrintTable` calls *group*`.operations.Table` which performs the printing.

```
gap> g := Group( (1,2), (3,4) );      # this is Klein's four group
Group( (1,2), (3,4) )
gap> PrintTable( g );
Let:
g0 := ()
g1 := (3,4)
g2 := (1,2)
g3 := (1,2)(3,4)

  +  | g0 g1 g2 g3
 ------------------
  g0 | g0 g1 g2 g3
  g1 | g1 g0 g3 g2
  g2 | g2 g3 g0 g1
  g3 | g3 g2 g1 g0
```

4.4.2 Endomorphisms

`Endomorphisms(` *group* `)`
`Endomorphisms` computes all the endomorphisms of the group *group*. This function is most essential for computing the nearrings on a group. The endomorphisms are returned as a list of transformations s.t. the identity endomorphism is always the last entry in this list. For each transformation in this list the record component `isGroupEndomorphism` is set to `true` and if such a transformation is in fact an automorphism then in addition the record component `isGroupAutomorphism` is set to `true`.

```
gap> TransformationPrintLevel("l");
gap> g := Group( (1,2,3) );
Group( (1,2,3) )
gap> Endomorphisms( g );
[ Transformation on Group( (1,2,3) ):
      () -> ()
      (1,2,3) -> ()
```

```
    (1,3,2) -> ()
  , Transformation on Group( (1,2,3) ):
    () -> ()
    (1,2,3) -> (1,3,2)
    (1,3,2) -> (1,2,3)
  , Transformation on Group( (1,2,3) ):
    () -> ()
    (1,2,3) -> (1,2,3)
    (1,3,2) -> (1,3,2)
   ]
```

4.4.3 Automorphisms

`Automorphisms(` *group* `)`
`Automorphisms` computes all the automorphisms of the group *group*. The automorphisms are returned as a list of transformations s.t. the identity automorphism is always the last entry in this list. For each transformation in this list the record components **`isGroupEndomorphism`** and **`isGroupAutomorphism`** are both set to **`true`**.

```
gap> TransformationPrintLevel("s");
gap> d8 := Group( (1,2,3,4), (2,4) ); # dihedral group of order 8
Group( (1,2,3,4), (2,4) )
gap> a := Automorphisms( d8 );
[ [ 1, 2, 8, 7, 5, 6, 4, 3 ] , [ 1, 3, 2, 7, 8, 6, 4, 5 ] ,
  [ 1, 3, 5, 4, 8, 6, 7, 2 ] , [ 1, 5, 3, 7, 2, 6, 4, 8 ] ,
  [ 1, 5, 8, 4, 2, 6, 7, 3 ] , [ 1, 8, 2, 4, 3, 6, 7, 5 ] ,
  [ 1, 8, 5, 7, 3, 6, 4, 2 ] , [ 1, 2, 3, 4, 5, 6, 7, 8 ]  ]
```

4.4.4 InnerAutomorphisms

`InnerAutomorphisms(` *group* `)`
`InnerAutomorphisms` computes all the inner automorphisms of the group *group*. The inner automorphisms are returned as a list of transformations s.t. the identity automorphism is always the last entry in this list. For each transformation in this list the record components **`isInnerAutomorphism`**, **`isGroupEndomorphism`**, and **`isGroupAutomorphism`** are all set to **`true`**.

```
gap> i := InnerAutomorphisms( d8 );
  [ [ 1, 2, 8, 7, 5, 6, 4, 3 ] , [ 1, 5, 3, 7, 2, 6, 4, 8 ] ,
    [ 1, 5, 8, 4, 2, 6, 7, 3 ] , [ 1, 2, 3, 4, 5, 6, 7, 8 ]  ]
```

4.4.5 SmallestGeneratingSystem

`SmallestGeneratingSystem(` *group* `)`
`SmallestGeneratingSystem` computes a smallest generating system of the group *group* i.e. a smallest subset of the elements of the group s.t. the group is generated by this subset.

Remark: there is a GAP standard library function `SmallestGenerators` for permutation groups. This function computes a generating set of a given group such that its elements are smallest possible permutations (w.r.t the GAP internal sorting of permutations).

```
gap> s5 := SymmetricGroup( 5 );
Group( (1,5), (2,5), (3,5), (4,5) )
gap> SmallestGenerators( s5 );
[ (4,5), (3,4), (2,3), (1,2) ]
gap> SmallestGeneratingSystem( s5 );
[ (1,3,5)(2,4), (1,2)(3,4,5) ]
```

4.4.6 IsIsomorphicGroup

`IsIsomorphicGroup(` *g1*, *g2* `)`
`IsIsomorphicGroup` determines if the groups *g1* and *g2* are isomorphic and if so, returns a group homomorphism that is an isomorphism between these two groups and `false` otherwise.

```
gap> g1 := Group( (1,2,3) );  # cyclic group of order 3
Group( (1,2,3) )
gap> g2 := Group( (7,8,9) );  # again cyclic group of order 3
Group( (7,8,9) )
gap> g1 = g2;
false
gap> IsIsomorphicGroup( g1, g2 );
GroupHomomorphismByImages( Group( (1,2,3) ), Group( (7,8,9) ),
[ (1,2,3) ], [ (7,8,9) ] )
```

4.4.7 Predefined groups

The following variables are predefined as according permutation groups with a default smallest set of generators: `C2`, `C3`, `C4`, `V4`, `C5`, `C6`, `S3`, `C7`, `C8`, `C2xC4`, `C2xC2xC2`, `D8`, `Q8`, `C9`, `C3xC3`, `C10`, `D10`, `C11`, `C12`, `C2xC6`, `D12`, `A4`, `T`, `C13`, `C14`, `D14`, `C15`.

```
gap> S3;
S3
gap> Elements(S3);
[ (), (2,3), (1,2), (1,2,3), (1,3,2), (1,3) ]
gap> IsPermGroup(S3);
true
gap> S3.generators;
[ (1,2), (1,2,3) ]
```

4.5 A Note on Installing the Files

There are two compressed files available which contain all source files as well as the nearring and semigroup library files: `nrsg.zip` is the *pkzipped* DOS version and `nrsg.tar.gz` is the *tarred* and *gzipped* UNIX version.

To uncompress the files we suggest that you move `nrsg.zip` (resp. `nrsg.tar.gz`) into your GAP directory and

- in the DOS case: use the command `pkunzip -d nrsg.zip`.
- in the UNIX case: first use `gunzip nrsg.tar.gz` (which will get you the file `nrsg.tar`) and then use `tar xvf nrsg.tar`. (both steps can also be performed with the single command `gunzip -c nrsg.tar.gz | tar xvf -`).

In both cases the directory `nrsg` with two subdirectories (`nr` and `sg`) will be created. `nrsg` contains the source files:

```
cnr.g, csg.g, g.g, initnrsg.g, nr.g, sg.g, smallgps.g, t.g
```

`nrsg/nr` contains the nearring library files:

```
nr10_1.nr, nr10_2.nr, nr11_1.nr, nr12_1.nr, nr12_2.nr,
nr12_3.nr, nr12_4.nr, nr12_5.nr, nr13_1.nr, nr14_1.nr,
nr14_2.nr, nr15_1.nr, nr8_1.nr,  nr8_2.nr,  nr8_3.nr,
nr8_4.nr,  nr8_5.nr,  nr9_1.nr,  nr9_2.nr,  nr_2_7.nr,
```

and `nrsg/sg` contains the semigroup library files:

```
sg1_4.sg, sg5.sg
```

In case you did not uncompress the files in your default GAP directory, edit the file `initnrsg.g` (cf. appendix B.1) and set three path variables: set `PROG_PATH_DIR` to the full directory name where the source files can be found, set `NR_PATH_DIR` to the full directory name where the nearring library files can be found, set `SG_PATH_DIR` to the full directory name where the semigroup library files can be found.

Example:

```
PROG_PATH_DIR := "c:/my/own/dir/nrsg/";
NR_PATH_DIR := "c:/my/own/dir/nrsg/nr/";
SG_PATH_DIR := "c:/my/own/dir/nrsg/sg/";
```

Make sure not to omit the last "/".

Now you can initialize the installed files by just using the `Read` function to read the file `initnrsg.g`. (Be sure to specify the correct path).

Another possibility would be to add a line like
`Read( "`*PATH_NAME*`/initnrsg.g" );`
to your `gap.rc` (resp. `.gaprc`) file which would make the functions available every time you start GAP up.

Appendix A

Examples

A.1 How to find nearrings with (or without) certain properties

The nearring library files can be used to systematically search for nearrings with (or without) certain properties.

For instance, the function `LibraryNearring` can be integrated into a loop or occur as a part of the `Filtered` or the `First` function to get all numbers of classes (or just the first class) of nearrings on a specified group which have the specified properties.

In what follows, we shall give a few examples how this can be accomplished:

To get the number of the first class of nearrings on the group C_6 which have an identity, one could use a command like:

```
gap> First( [1..60], i ->
>           Identity( LibraryNearring( "C6", i ) ) <> [ ] );
28
```

If we try the same with S_3, we will get an error message, indicating that there is no nearring with identity on S_3:

```
gap> First( [1..39], i -> Identity( LibraryNearring( "S3", i) ) <> [ ] );
Error, at least one element of <list> must fulfill <func> in
First( [ 1 .. 39 ], function ( i ) ... end ) called from
main loop
brk> gap>
```

To get all seven classes of nearrings with identity on the dihedral group D_8 of order 8, something like the following will work fine:

```
gap> l := Filtered( [1..1447], i ->
>                   Identity( LibraryNearring( "D8", i ) ) <> [ ] );
```

```
[ 842, 844, 848, 849, 1094, 1096, 1097 ]
gap> time;
435280
```

Note that a search like this may take a few minutes.

Here is another example that provides the class numbers of the four boolean nearrings on D_8:

```
gap> l := Filtered( [1..1447], i ->
>                   IsBooleanNearring( LibraryNearring( "D8", i ) ) );
[ 1314, 1380, 1446, 1447 ]
```

The search for class numbers of nearrings can also be accomplished in a loop like the following:

```
gap> l:=[ ];;
gap> for i in [1..1447] do
>   n := LibraryNearring( "D8", i );
>   if IsDgNearring( n ) and
>   not IsDistributiveNearring( n ) then
>     Add( l, i );
>   fi;
> od;
gap> time;
956690
gap> l;
[ 765, 1094, 1098, 1306 ]
```

This provides a list of those class numbers of nearrings on D_8 which are distributively generated but not distributive.

Two or more conditions for library nearrings can also be combined with "or":

```
gap> l := [ ];;
gap> for i in [1..1447] do
>   n := LibraryNearring( "D8", i );
>   if Size( ZeroSymmetricElements( n ) ) < 8 or
>      Identity( n ) <> [ ] or
>      IsIntegralNearring( n ) then
>        Add( l, i );
>   fi;
> od;
gap> time;
338180
gap> l;
[ 842, 844, 848, 849, 1094, 1096, 1097, 1314, 1315, 1316, 1317, 1318,
  1319, 1320, 1321, 1322, 1323, 1324, 1325, 1326, 1327, 1328, 1329,
  1330, 1331, 1332, 1333, 1334, 1335, 1336, 1337, 1338, 1339, 1340,
```

```
  1341, 1342, 1343, 1344, 1345, 1346, 1347, 1348, 1349, 1350, 1351,
  1352, 1353, 1354, 1355, 1356, 1357, 1358, 1359, 1360, 1361, 1362,
  1363, 1364, 1365, 1366, 1367, 1368, 1369, 1370, 1371, 1372, 1373,
  1374, 1375, 1376, 1377, 1378, 1379, 1380, 1381, 1382, 1383, 1384,
  1385, 1386, 1387, 1388, 1389, 1390, 1391, 1392, 1393, 1394, 1395,
  1396, 1397, 1398, 1399, 1400, 1401, 1402, 1403, 1404, 1405, 1406,
  1407, 1408, 1409, 1410, 1411, 1412, 1413, 1414, 1415, 1416, 1417,
  1418, 1419, 1420, 1421, 1422, 1423, 1424, 1425, 1426, 1427, 1428,
  1429, 1430, 1431, 1432, 1433, 1434, 1435, 1436, 1437, 1438, 1439,
  1440, 1441, 1442, 1443, 1444, 1445, 1446, 1447 ]
gap> Length( l );
141
```

This provides a list of all 141 class numbers of nearrings on D_8 which are non-zerosymmetric or have an identity or are integral. (cf. [Pil83], pp. 416ff).

The following loop does the same for the nearrings on the quaternion group of order 8, Q_8:

```
gap> l := [ ];;
gap> for i in [1..281] do
>   n := LibraryNearring( "Q8", i );
>   if Size( ZeroSymmetricElements( n ) ) < 8 or
>      Identity( n ) <> [ ] or
>      IsIntegralNearring( n ) then
>        Add( l, i );
>   fi;
> od;
gap> time;
49920
gap> l;
[ 280, 281 ]
```

Once a list of class numbers has been computed (in this case here: `l`), e.g. the function `LibraryNearringInfo` can be used to print some information about these nearrings:

```
gap> LibraryNearringInfo( "Q8", l );
--------------------------------------------------------------------------
>>> GROUP: Q8
elements: [ n0, n1, n2, n3, n4, n5, n6, n7 ]
addition table:

  +  | n0 n1 n2 n3 n4 n5 n6 n7
 ----------------------------
  n0 | n0 n1 n2 n3 n4 n5 n6 n7
  n1 | n1 n2 n3 n0 n7 n4 n5 n6
  n2 | n2 n3 n0 n1 n6 n7 n4 n5
  n3 | n3 n0 n1 n2 n5 n6 n7 n4
  n4 | n4 n5 n6 n7 n2 n3 n0 n1
  n5 | n5 n6 n7 n4 n1 n2 n3 n0
```

```
  n6 | n6 n7 n4 n5 n0 n1 n2 n3
  n7 | n7 n4 n5 n6 n3 n0 n1 n2

group endomorphisms:
1:   [ n0, n0, n0, n0, n0, n0, n0, n0 ]
2:   [ n0, n0, n0, n0, n2, n2, n2, n2 ]
3:   [ n0, n1, n2, n3, n5, n6, n7, n4 ]
4:   [ n0, n1, n2, n3, n6, n7, n4, n5 ]
5:   [ n0, n1, n2, n3, n7, n4, n5, n6 ]
6:   [ n0, n2, n0, n2, n0, n2, n0, n2 ]
7:   [ n0, n2, n0, n2, n2, n0, n2, n0 ]
8:   [ n0, n3, n2, n1, n4, n7, n6, n5 ]
9:   [ n0, n3, n2, n1, n5, n4, n7, n6 ]
10:  [ n0, n3, n2, n1, n6, n5, n4, n7 ]
11:  [ n0, n3, n2, n1, n7, n6, n5, n4 ]
12:  [ n0, n4, n2, n6, n1, n7, n3, n5 ]
13:  [ n0, n4, n2, n6, n3, n5, n1, n7 ]
14:  [ n0, n4, n2, n6, n5, n1, n7, n3 ]
15:  [ n0, n4, n2, n6, n7, n3, n5, n1 ]
16:  [ n0, n5, n2, n7, n1, n4, n3, n6 ]
17:  [ n0, n5, n2, n7, n3, n6, n1, n4 ]
18:  [ n0, n5, n2, n7, n4, n3, n6, n1 ]
19:  [ n0, n5, n2, n7, n6, n1, n4, n3 ]
20:  [ n0, n6, n2, n4, n1, n5, n3, n7 ]
21:  [ n0, n6, n2, n4, n3, n7, n1, n5 ]
22:  [ n0, n6, n2, n4, n5, n3, n7, n1 ]
23:  [ n0, n6, n2, n4, n7, n1, n5, n3 ]
24:  [ n0, n7, n2, n5, n1, n6, n3, n4 ]
25:  [ n0, n7, n2, n5, n3, n4, n1, n6 ]
26:  [ n0, n7, n2, n5, n4, n1, n6, n3 ]
27:  [ n0, n7, n2, n5, n6, n3, n4, n1 ]
28:  [ n0, n1, n2, n3, n4, n5, n6, n7 ]

NEARRINGS:
----------------------------------------------------------------------
280)  phi: [ 1, 28, 28, 28, 28, 28, 28, 28 ];  28;  -B----I--P-RW
----------------------------------------------------------------------
281)  phi: [ 28, 28, 28, 28, 28, 28, 28, 28 ];  28;  -B----I--P-RW
----------------------------------------------------------------------
```

A.2 How to input a nearring with known multiplication table

Suppose that for a given group g the multiplication table of a binary operation $*$ on the elements of g is known such that $*$ is a nearring multiplication on g. Then the constructor function **Nearring** can be used to input the nearring specified by g and $*$.

An example shall illustrate a possibility how this could be accomplished: Take the group S_3, which in GAP can be defined e.g. by

```
gap> g := Group( (1,2), (1,2,3) );
Group( (1,2), (1,2,3) )
```

This group has the following list of elements:

```
gap> Elements( g );
[ (), (2,3), (1,2), (1,2,3), (1,3,2), (1,3) ]
```

Let '1' stand for the first element in this list, '2' for the second, and so on up to '6' which stands for the sixth element in the following multiplication table:

*	1	2	3	4	5	6
1	1	1	1	1	1	1
2	2	2	2	2	2	2
3	2	2	6	3	6	3
4	1	1	5	4	5	4
5	1	1	4	5	4	5
6	2	2	3	6	3	6

A nearring on `g` with this multiplication can be input by first defining a multiplication function, say `m` in the following way:

```
gap> m := function( x, y )
>   local elms, table;
>   elms := Elements( g );
>   table := [ [ 1, 1, 1, 1, 1, 1 ],
>              [ 2, 2, 2, 2, 2, 2 ],
>              [ 2, 2, 6, 3, 6, 3 ],
>              [ 1, 1, 5, 4, 5, 4 ],
>              [ 1, 1, 4, 5, 4, 5 ],
>              [ 2, 2, 3, 6, 3, 6 ] ];
>
>   return elms[ table[Position( elms, x )][Position( elms, y )] ];
> end;
function ( x, y ) ... end
```

Then the nearring can be constructed by calling `Nearring` with the parameters `g` and `m`:

```
gap> n := Nearring( g, m );
Nearring( Group( (1,2), (1,2,3) ), multiplication( x, y ) )
```

Appendix B

Source Code

B.1 The Source File initnrsg.g

```
################################################################################
##  File: INITNRSG.G                                                         ##
################################################################################
##  Use the following three variables to explicitly specify where the
##  program files, the nearring library files, and the semigroup library
##  files can be found.
##  The default path names are: PROG_PATH_NAME = "%GAP_DIR%/nrsg/",
##                              NR_PATH_NAME   = "%GAP_DIR%/nrsg/nr/", and
##                              SG_PATH_NAME   = "%GAP_DIR%/nrsg/sg/".
##  Make sure that the last "/" is there.
################################################################################
## PROG_PATH_NAME := "c:/gap/nrsg/";
## NR_PATH_NAME   := "c:/gap/nrsg/nr/";
## SG_PATH_NAME   := "c:/gap/nrsg/sg/";

if not IsBound( PROG_PATH_NAME ) then
  PROG_PATH_NAME := ReplacedString( LIBNAME, "lib", "nrsg" );
fi;

ReadProg := function ( name )
  if not ReadPath( PROG_PATH_NAME, name, ".g", "ReadProg" )  then
    Error( "the program file '",
           name, ".g' must exist and be readable");
  fi;
end;

Print( "Setting up GAP ...\n" );

Print( "Reading t.g\n" );
ReadProg( "t" );

Print( "Reading g.g\n" );
```

```
ReadProg( "g" );

Print( "Reading smallgps.g (defining the groups C2,C3,C4,V4,C5,C6,S3,C7,\n",
  "                    C8,C2xC4,C2xC2xC2,Q8,D8,C9,C3xC3,C10,D10,C11,\n",
  "                    C12,C2xC6,D12,A4,T,C13,C14,D14,C15)\n" );
ReadProg( "smallgps" );

Print( "Reading sg.g\n" );
ReadProg( "sg" );

Print( "Reading nr.g\n" );
ReadProg( "nr" );
```

B.2 The Source File `smallgps.g`

```
################################################################################
##  File: GROUPS.DEF                                                          ##
################################################################################
##  This defines all groups of low order without using any library.
################################################################################
##
C2             := Group( (1,2) );
C2.name        := "C2";  # cyclic group order 2

C3             := Group( (1,2,3) );
C3.name        := "C3";  # cyclic group order 3

C4             := Group( (1,2,3,4) );
C4.name        := "C4";  # cyclic group order 4
V4             := DirectProduct( C2, C2 );
V4.name        := "V4";  # Klein's four group

C5             := Group( (1,2,3,4,5) );
C5.name        := "C5";  # cyclic group order 5

C6             := Group( (1,2,3,4,5,6) );
C6.name        := "C6";  # cyclic group order 6
S3             := Group( (1,2), (1,2,3) );
S3.name        := "S3";  # S3

C7             := Group( (1,2,3,4,5,6,7) );
C7.name        := "C7";  # cyclic group order 7

C8             := Group( (1,2,3,4,5,6,7,8) );
C8.name        := "C8";  # cyclic group order 8
C2xC4          := DirectProduct( C2, C4 );
C2xC4.name     := "C2xC4";
C4xC2          := DirectProduct( C4, C2 );
C4xC2.name     := "C4xC2";
C2xC2xC2       := DirectProduct( C2, C2, C2 );
```

```
C2xC2xC2.name := "C2xC2xC2";
D8            := Group( (1,2,3,4), (2,4) );
D8.name       := "D8";  # dihedral group of order 8
Q8            := Group( (1,2,3,4)(5,6,7,8), (1,5,3,7)(2,8,4,6) );
Q8.name       := "Q8";  # quaternion group of order 8

C9            := Group( (1,2,3,4,5,6,7,8,9) );
C9.name       := "C9";  # cyclic group of order 9
C3xC3         := DirectProduct( C3, C3 );
C3xC3.name    := "C3xC3";

C10           := Group( (1,2,3,4,5,6,7,8,9,10) );
C10.name      := "C10"; # cyclic group of order 10
D10           := Group( (1,2,3,4,5), (2,5)(3,4) );
D10.name      := "D10"; # dihedral group of order 10

C11           := Group( (1,2,3,4,5,6,7,8,9,10,11) );
C11.name      := "C11"; # cyclic group of order 11

C12           := Group( (1,2,3,4,5,6,7,8,9,10,11,12) );
C12.name      := "C12"; # cyclic group of order 12
C2xC6         := DirectProduct( C2, C6 );
C2xC6.name    := "C2xC6";
D12           := Group( (1,2,3,4,5,6), (2,6)(3,5) );
D12.name      := "D12";
A4            := Group( (1,2,4), (2,3,4) );
A4.name       := "A4";
T             := Group( ( 1, 2, 4, 7, 9,12)( 3, 6,10,11, 8, 5),
                        ( 1, 3, 7,11)( 2, 5, 9,10)( 4, 8,12, 6) );
T.name        := "T";  # ( ZS-metacyclic )

C13           := Group( (1,2,3,4,5,6,7,8,9,10,11,12,13) );
C13.name      := "C13";

C14           := Group( (1,2,3,4,5,6,7,8,9,10,11,12,13,14) );
C14.name      := "C14";
D14           := Group( (1,2,3,4,5,6,7), (2,7)(3,6)(4,5) );
D14.name      := "D14";

C15           := Group( (1,2,3,4,5,6,7,8,9,10,11,12,13,14,15) );
C15.name      := "C15";
```

B.3 The Source File `t.g`

```
#############################################################################
##  File: T.G                                                              ##
#############################################################################
##
#############################################################################
##
```

```
#V  TRANSFORMATION_PRINT_LEVEL: this global variable determines how
##                              detailed transformations are printed.
##
TRANSFORMATION_PRINT_LEVEL := -1; # set default: print detailed

#############################################################################
##
#F  TransformationPrintLevel( <arg> ). . set print detail for transformations
##
TransformationPrintLevel := function( arg )
  if Length( arg ) > 1 or
     ( Length( arg ) = 1 and not IsString( arg[1] ) ) then
    Error( "Usage: TransformationPrintLevel() or \n",
           "TransformationPrintLevel( <string> ) where <string> must be\n",
           "\"long\" or \"short\" or \"l\" or \"s\"" );
  fi;

  if Length( arg ) = 0 then
    if TRANSFORMATION_PRINT_LEVEL = 0 then
      return "short";
    else
      return "long";
    fi;
  else
    if arg[1][1] = 'l' then
      TRANSFORMATION_PRINT_LEVEL := -1;
    else
      TRANSFORMATION_PRINT_LEVEL := 0;
    fi;
  fi;
  return;
end;

TPL := TransformationPrintLevel;

#############################################################################
##
#F  Parent( <G>, ... )  . . . . . . . . . . . . . . . . . . common parent group
##
##  overlay the original Parent function thus fixing some
##  incompatibilities within GAP itself.
##
Parent := function ( arg )
    local   G;

    # check the number of arguments
    if Length( arg ) = 0  then
        Error( "usage: Parent( <G>, ... )" );
    fi;

    if IsSet( arg[1] ) then
      return arg[1];
```

```
    fi;

    # compute the parent group
    G := arg[1].operations.Parent( arg );

    # return the parent group
    return G;

end;

#############################################################################
##
#F  IsTransformation( <obj> ) . . . . . . . test if <obj> is a transformation
##
IsTransformation := function( obj )
  return IsRec( obj )
    and IsBound( obj.isTransformation ) and obj.isTransformation;
end;

#############################################################################
##
#F  IsSetTransformation( <obj> ). . . . test if <obj> is a set transformation
##
IsSetTransformation := function( obj )
  return IsRec( obj )
    and IsBound( obj.isSetTransformation ) and obj.isSetTransformation;
end;

#############################################################################
##
#F  IsGroupTransformation( <obj> ). . test if <obj> is a group transformation
##
IsGroupTransformation := function( obj )
  return IsRec( obj )
    and IsBound( obj.isGroupTransformation ) and obj.isGroupTransformation;
end;

#############################################################################
##
#V  TransformationOps . . . . . . . . . operations record for transformations
##
##  'TransformationOps' is the operations record for transformations.  This
##  is initially a copy of 'MappingOps'.  This way all the default methods
##  for mappings are inherited.
##
SetTransformationOps   := Copy( MappingOps );
GroupTransformationOps := rec();

#############################################################################
##
#F  Ker( <t> ) . . . . . . . . . . . . . . . . . . Ker of a full transformation
##
```

```
##  dispatcher function for computing the equivalence relation Ker(t)
##  where t must be a valid full transformation.
##
Ker := function( t )

  if IsTransformation( t ) then
    if not IsBound( t.ker ) then
      t.ker := t.operations.Ker( t );
    fi;
  else
    Error( "Usage: Ker( <t> ) where <t> must be a transformation" );
  fi;

  return t.ker;
end;

#############################################################################
##
#F  AsTransformation( <T> ). . . . . . . make a mapping into a transformation
##
##  Function for making any mapping with the same range and source into a
##  Transformation.
##
AsTransformation := function( T )
  local t,l;
  if not ( IsRec( T ) and IsMapping( T ) and ( T.source = T.range ) and
           ( IsGroup( T.source ) or IsSet( T.range ) ) ) then
    Error( "Usage: AsTransformation( <T> ) where <T> must be a mapping with",
           "\nthe equal source and range which must be a set or a ",
           "group" );
  fi;

  # make the transformation
  t := Copy( T );
  # enter the tag components
  t.isTransformation       := true;
  if IsSet( T.source ) then
    t.isSetTransformation   := true;
    t.elements              := T.source;
  elif IsGroup( T.source ) then
    t.isGroupTransformation := true;
    t.isGroupElement        := true;
    t.elements              := Elements( T.source );
  fi;
  # make the transformation list
  l := List( t.elements, e -> Position( t.elements, Image( T, e ) ) );
  t.tfl                    := l; # a list which represents the transformation
                                 # in the obvious way. ('TransFormationList')
  # enter the operations record
  if IsSet( T.source ) then
    t.operations           := SetTransformationOps;
  elif IsGroup( T.source ) then
```

```
    t.operations            := GroupTransformationOps;
  fi;

  return t;
end;

#############################################################################
##
#F  IdentityTransformation( <D> ). . . . . create the identity transformation
##
##  Constructor function for the identity Transformation on <D> where <D>
##  is either a set or a group.
##
IdentityTransformation := function( D )
  local t,l;
  # check on the validity of the input
  if not ( IsSet( D ) or IsGroup( D ) ) then
    Error( "Usage: IdentityTransformation( <D> ) where <D> must be a set\n",
           "or a group" );
  fi;

  l := List( [1..Size( D )], i -> i );
  # make the transformation
  t := rec();
  # enter the tag components
  t.isGeneralMapping       := true;
  t.domain                 := Mappings;
  t.isMapping              := true;
  t.isTransformation       := true;
  if IsSet( D ) then
    t.isSetTransformation   := true;
    t.elements              := D;
  elif IsGroup( D ) then
    t.isGroupTransformation := true;
    t.isGroupElement        := true;
    t.elements              := Elements( D );
  fi;
  # enter source and range
  t.source                 := D;
  t.range                  := D;
  t.tfl                    := l; # a list which represents the transformation
                                 # in the obvious way. ('TransFormationList')
  # enter the operations record
  if IsSet( D ) then
    t.operations            := SetTransformationOps;
  elif IsGroup( D ) then
    t.operations            := GroupTransformationOps;
  fi;
  # return the transformation
  return t;
end;
```

```
#############################################################################
##
#F  Transformation( <D>, <l> ). . . . . . . . . . . . create a transformation
##
##  Constructor function for a Transformation on <D> where <D> is either a
##  set or a group
##
Transformation := function( D, l )
  local t;
  # check on the validity of the input
  if not ( ( IsSet( D ) or IsGroup( D ) ) and IsVector( l ) and
           ForAll( l, x -> x in [1..Size( D )] ) and
           Length( l ) = Size( D )
         ) then
    Error( "Usage: Transformation( <obj>, <tfl> ) where <obj> must be a ",
           "set\nor a group and <tfl> must be a valid ",
           "transformation list for <obj>" );
  fi;

  # make the transformation
  t := rec();
  # enter the tag components
  t.isGeneralMapping       := true;
  t.domain                 := Mappings;
  t.isMapping              := true;
  t.isTransformation       := true;
  if IsSet( D ) then
    t.isSetTransformation   := true;
    t.elements             := D;
  elif IsGroup( D ) then
    t.isGroupTransformation := true;
    t.isGroupElement       := true;
    t.elements             := Elements( D );
  fi;
  # enter source and range and the transformation list
  t.source                 := D;
  t.range                  := D;
  t.tfl                    := l; # a list which represents the transformation
                                 # in the obvious way. ('TransFormationList')
  # enter the operations record
  if IsSet( D ) then
    t.operations           := SetTransformationOps;
  elif IsGroup( D ) then
    t.operations           := GroupTransformationOps;
    t.operations.ImageRepresentative :=
                 ConjugationGroupHomomorphismOps.ImageRepresentative;
    t.operations.PreImageRepresentative :=
                 ConjugationGroupHomomorphismOps.PreImageRepresentative;
  fi;

  # return the transformation
  return t;
```

```
end;

SetTransformationOps.ImageElm := function( t, elm )
  return t.elements[ t.tfl[ Position( t.elements, elm ) ] ];
end;

SetTransformationOps.ImagesElm := function( t, elm )
  return [ t.elements[ t.tfl[ Position( t.elements, elm ) ] ] ];
end;

SetTransformationOps.ImagesRepresentative := function( t, elm )
  return t.elements[ t.tfl[ Position( t.elements, elm ) ] ];
end;

SetTransformationOps.ImagesSource := function( t )
  return Set( Filtered( t.elements,
              elm -> Position( t.elements, elm ) in t.tfl ) );
end;

SetTransformationOps.Print := function( t )
  local elm;
  if TRANSFORMATION_PRINT_LEVEL <> 0 then
    if IsSetTransformation( t ) then
      Print( "Transformation on ", t.elements, ":\n" );
      for elm in t.elements do
        Print( "  ", elm, " -> ", t.tfl[elm], "\n" );
      od;
    else
      Print( "Transformation on ", t.source, ":\n" );
      for elm in t.elements do
        Print( "  ", elm, " -> ", Image( t, elm ), "\n" );
      od;
    fi;
  else
    Print( t.tfl, " " );
  fi;
end;

SetTransformationOps.\* := function( t1, t2 )
  local prd;
  # product of two transformations ( IF both arguments are transformations )
  if IsRec( t1 ) and IsRec( t2 ) and
    IsTransformation( t1 ) and IsTransformation( t2 ) and
    t1.source = t2.range then
      prd := Transformation( t1.source, t1.tfl{ t2.tfl } );
  else
    # delegate to MappingOps.\*
    prd := MappingOps.\* ( t1, t2 );
  fi;
  return prd;
end;
```

```
SetTransformationOps.Ker := function( t )
  local ker, # help variable: a list in which the eq.classes of the
             # relation Ker are put together
        elm; # help variable
  ker := [];
  for elm in Image( t ) do
    AddSet( ker, Filtered( t.elements, e -> Image( t, e ) = elm ) );
  od;
  return ker;
end;

SetTransformationOps.Rank := function( t )
  return Length( Image( t ) );
end;

GroupTransformationOps := Copy( SetTransformationOps );

GroupTransformationOps.\+ := function( t1, t2 )
  local tfl,  # the tfl of the resulting transformation
        elms, # a list of the elements involved
        l;    # number of elements
  # sum of two group transformations ( IF both arguments are group tf's )
  if IsRec( t1 ) and IsRec( t2 ) and
    IsGroupTransformation( t1 ) and IsGroupTransformation( t2 ) and
    t1.source = t2.source and t1.range = t2.range then
      elms := t1.elements;
      l    := Length( elms );
      tfl  := List( [1..l], i ->
                    Position( elms, elms[ t1.tfl[i] ] * elms[ t2.tfl[i] ] )
                  );
  else
    Error( "Usage: <t1> + <t2> where <t1> and <t2> must both be group\n",
           "transformations on the same group" );
  fi;
  return Transformation( t1.source, tfl );
end;

GroupTransformationOps.\- := function( t1, t2 )
  local tfl,  # the tfl of the resulting transformation
        elms, # a list of the elements involved
        l;    # number of elements
  # difference of two group tf's ( IF both arguments are group tf's )
  if IsRec( t1 ) and IsRec( t2 ) and
    IsGroupTransformation( t1 ) and IsGroupTransformation( t2 ) and
    t1.source = t2.source and t1.range = t2.range then
      elms := t1.elements;
      l    := Length( elms );
      tfl  := List( [1..l], i ->
                    Position( elms, elms[ t1.tfl[i] ] * elms[ t2.tfl[i] ]^-1 )
                  );
  else
    Error( "Usage: <t1> - <t2> where <t1> and <t2> must both be group\n",
```

```
          "transformations on the same group" );
  fi;
  return Transformation( t1.source, tfl );
end;
```

B.4 The Source File sg.g

```
#############################################################################
##  File: SG.G                                                             ##
#############################################################################
##
#############################################################################
##
#V  MAX_SET_SIZE: set the maximum size for for the set S on which the
##               semigroup of all transformations on S can be defined.
##  (this semigroup has then, of course, size MAX_SET_SIZE^MAX_SET_SIZE).
##
MAX_SET_SIZE := 5;

#############################################################################
##

if not IsBound( SG_PATH_NAME ) then
  SG_PATH_NAME := ReplacedString( LIBNAME, "lib", "nrsg/sg" );
fi;

ReadSg := function ( name )
  if not ReadPath( SG_PATH_NAME, name, ".sg", "ReadSg" )  then
    Error( "the semigroup library file '",
           name, ".sg' must exist and be readable");
  fi;
end;

AUTO( ReadSg( "sg1_4" ), SG1, SG2, SG3, SG4 );

AUTO( ReadSg( "sg5" ),   SG5 );

#############################################################################
##
#V  SgOps. . . . . . . . . . . . . . . . . . . operations record for semigroups
##
##  'SgOps' is the operations record for semigroups.  This is initially a
##  copy of 'DomainOps'.  This way all the default methods for domains are
##  inherited.
##
SgOps := Copy( DomainOps );

#############################################################################
##
#F  IsSemigroup( <S> ) . . . . . . . . . . . . . . . test if <S> is a semigroup
```

```
##
IsSemigroup := function( S )
  return IsRec( S ) and IsBound( S.isSemigroup ) and S.isSemigroup;
end;

#############################################################################
##
#F  IsTransformationSemigroup( <S> ). . . . . . test if <S> is a tf semigroup
##
IsTransformationSemigroup := function( S )
  return IsRec( S ) and
    IsBound( S.isTransformationSemigroup ) and S.isTransformationSemigroup;
end;

#############################################################################
##
#F  AllFunctions( <n> ) . create all f's from the set {1,2,...,n} into itself
##
##  Those functions are returned as lex. ordered list with [1,...,1] first,
##  and with the constant function [n,n,...,n] last.
##
AllFunctions := function( n )
  local af,i,j,k,done;

  i := [];
  for j in [1..n] do i[j] := 1; od;
  k := -1;
  af := [];

  while k <= n do
    AddSet( af, Reversed( i ) );
    k := 1; done := false;
    while not done and k <= n do
      if i[k] = n then  i[k] := 1; k := k+1;
      else  i[k] := i[k]+1; done := true;
      fi;
    od;
  od;

  return af;
end;

#############################################################################
##
#F  TransformationSemigroup(<gen>,...). . . . . . . . . . .create a semigroup
##
##  Constructor function for transformation semigroups
##
TransformationSemigroup := function( arg )

  local S,    # the semigroup to be returned
        t,    # transformations
```

```
        set,  # help variable: a set
        gens; # generators of the semigroup

  arg := Flat( arg );
  S := rec();

  if Length( arg ) = 1 and IsInt( arg[1] ) and arg[1] > 0 then

    if arg[1] > MAX_SET_SIZE then
      Error( "warning: maximum size of the set on which the semigroup of all",
          "\ntransformations can be defined exceeded. Increase MAX_SET_SIZE",
          "\nin the file SG.G to allow bigger sets" );
    else
      gens := []; set := List( [1..arg[1]], i -> i );
      for t in AllFunctions( arg[1] ) do
        AddSet( gens, Transformation( set, t ) );
      od;
      S.elements := gens;
    fi;

  else

    # check the validity of the input
    if not ForAll( arg, t -> IsTransformation( t ) and
                       t.source = arg[1].source and t.range = arg[1].range
               ) then
      Error( "Usage: TransformationSemigroup( <t1>, <t2>, ... ) where all ",
             "arguments\nmust be transformations on the same set or \n",
             "TransformationSemigroup( <n> ) where <n> must be a positive ",
             "integer" );
    fi;

    # make sure that all sources and ranges of the generators are not only
    # equal but identical.
    gens := [];
    for t in arg do
      t.source := arg[1].source; t.range := arg[1].source;
      AddSet( gens, t );
    od;

  fi;

  # make the domain
  S.isDomain                  := true;
  S.isSemigroup               := true;
  S.isTransformationSemigroup := true;
  S.generators                := gens;
  S.multiplication            := function( x, y ) return x * y; end;
  S.operations                := SgOps;

  return S;
end;
```

```
###############################################################################
##
##  SgOps.\=( <S1>, <S2> ). . . . . . . compare two transformation semigroups
##
SgOps.\= := function( S1, S2 )
  if IsRec( S1 ) and IsRec( S2 ) and
    S1.isTransformationSemigroup and S2.isTransformationSemigroup then
      return S1.generators = S2.generators;
  else
    return DomainOps.\= ( S1, S2 );
  fi;
end;

###############################################################################
##
##  SgOps.Print( <S> ) . . . . . . . . . . . print a transformation semigroup
##
SgOps.Print := function( S )
  local i;
  Print( "TransformationSemigroup( " );
  for i in [ 1..Length( S.generators ) - 1 ] do
    Print( S.generators[i] );
    if TRANSFORMATION_PRINT_LEVEL = 0 then Print( ", " ); fi;
  od;
  Print( S.generators[ Length( S.generators )], " ) " );
end;

###############################################################################
##
#F  SgOps.Elements( <S> ). compute the elements of a transformation semigroup
##
SgOps.Elements := function ( S )
  local tset,             # set of transformation lists
        previously_added, # set of tf's added in the previous loop execution
        newly_added,      # set of tf's added in the actual loop execution
        t1,t2,t;          # help variables: transformation lists

  if IsBound( S.elements ) then
    return S.elements;
  else
    # perform a simple closure algorithm
    tset := Set( List( S.generators, t -> t.tfl ) );
    previously_added := Copy( tset );

    repeat
      newly_added := [];
      for t1 in tset do
        for t2 in previously_added do
          t := t1{ t2 };
          if not t in tset and not t in previously_added
            and not t in newly_added
```

```
          then
            AddSet( newly_added, t );      # Print( "Adding ", t, "\n" );
          fi;
          if t1 <> t2 then
            t := t2{ t1 };
            if not t in tset and not t in previously_added
              and not t in newly_added
            then
              AddSet( newly_added, t );    # Print( "Adding ", t, "\n" );
            fi;
          fi;
        od;
      od;
      for t1 in previously_added do
        for t2 in previously_added do
          t := t1{ t2 };
          if not t in tset and not t in previously_added
            and not t in newly_added
          then
            AddSet( newly_added, t );      # Print( "Adding ", t, "\n" );
          fi;
        od;
      od;
      UniteSet( tset, previously_added );
      previously_added := Copy( newly_added );

    until newly_added = [];

    if not IsBound( S.size ) then S.size := Length( tset ); fi;

    S.elements := [];
    for t in tset do
      AddSet( S.elements, Transformation( S.generators[1].source, t ) );
    od;

    if not IsBound( S.rank ) then S.rank := S.operations.Rank( S ); fi;

    return S.elements;
  fi;
end;

#############################################################################
##
#F  Rank( <obj> ) . . . . . . . . . . . . Rank of a transformation or a tf sg
##
##  Dispatcher function for computing th rank of a transformation or a
##  transformation semigroup.
##
Rank := function( obj )
  if not ( IsTransformation( obj ) or IsTransformationSemigroup( obj ) ) then
    Error( "Usage: Rank( <obj> ) where <obj> must be a ",
           "\ntransformation or a transformation semigroup" );
```

```
  fi;

  if not IsBound( obj.rank ) then
    obj.rank := obj.operations.Rank( obj );
  fi;

  return obj.rank;
end;

#############################################################################
##
#F  SgOps.Rank( <S> ). . . . . . . . . . . Rank of a transformation semigroup
##
##  This function returns the rank of a transformation semigroup S,i.e.
##  min(rank(t)) for all transformations t in S (see Lallement p. 20)
##
SgOps.Rank := function ( S )
  return Minimum( List( Elements( S ), t -> Length( Set( t.tfl ) ) ) );
end;

#############################################################################
##
#F  SmallestIdeal( <S> ) . . . . smallest ideal of a transformation semigroup
##  Dispatcher function
##
SmallestIdeal := function( S )

  if not IsTransformationSemigroup( S ) then
    Error( "Usage: SmallestIdeal( <S> ) where <S> must be a ",
           "\ntransformation semigroup" );
  fi;

  if not IsBound( S.smallestIdeal ) then
    S.smallestIdeal := S.operations.SmallestIdeal( S );
  fi;

  return S.smallestIdeal;
end;

#############################################################################
##
#F  SgOps.IsSimple( <S> ) . check if a transformation semigroup <S> is simple
##
SgOps.IsSimple := function( S )

  return S = SmallestIdeal( S );

end;

#############################################################################
##
#F  SgOps.SmallestIdeal( <S> ) . smallest ideal of a transformation semigroup
```

```
##
##  this function computes the smallest ideal of a transformation semigroup S.
##  Take all the transformations of minimal rank.
##
SgOps.SmallestIdeal := function ( S )
  local r,     # help variable: rank of the semigroup S
        minid; # help variable: the smallest ideal

  r     := S.operations.Rank( S );
  minid := Filtered( Elements( S ), t -> Length( Set( t.tfl ) ) = r );
  return minid;
end;

#############################################################################
##
#F  SgOps.PrincipalLeftIdeal( <S>, <t> ) . . . . . . . . . . . . . . . . . .
##
##  Compute the principal left ideal of a transformation semigroup generated
##  by the element <t>.
##
SgOps.PrincipalLeftIdeal := function( S, t )
  local elms,elm,pli,PLI;

  elms := Elements( S );
  pli  := [ t.tfl ];

  for elm in elms do
    AddSet( pli, elm.tfl{ t.tfl } );
  od;

  PLI := [];
  for elm in pli do
    AddSet( PLI, Transformation( S.generators[1].source, elm ) );
  od;
  return PLI;
end;

#############################################################################
##
#F  SgOps.PrincipalRightIdeal( <S>, <t> ) . . . . . . . . . . . . . . . . . .
##
##  Compute the principal right ideal of a transformation semigroup generated
##  by the element <t>.
##
SgOps.PrincipalRightIdeal := function( S, t )
  local elms,elm,pri,PRI;

  elms := Elements( S );
  pri  := [ t.tfl ];

  for elm in elms do
    AddSet( pri, t.tfl{ elm.tfl } );
```

```
  od;

  PRI := [];
  for elm in pri do
    AddSet( PRI, Transformation( S.generators[1].source, elm ) );
  od;
  return PRI;
end;

#############################################################################
##
#F  SgOps.PrincipalIdeal( <S>, <t> ). . . . . . . . . . . . . . . . . . . . .
##
##  Compute the principal ideal of a transformation semigroup generated
##  by the element <t>.
##
SgOps.PrincipalIdeal := function( S, t )
  local elm,PI,tl;

  PI  := [ t ];

  tl := S.operations.PrincipalLeftIdeal( S, t );
  for elm in tl do
    UniteSet( PI, S.operations.PrincipalRightIdeal( S, elm ) );
  od;

  return PI;
end;

#############################################################################
##
#F  Green( <S>, <type> ) . . . . . compute Green's relations for a tf sg <S>
##
##  Dispatcher function for all of Green's relations
##
Green := function( S, type )

  if not ( IsTransformationSemigroup( S ) and
           type in [ "L","l","R","r","J","j","D","d","H","h" ] ) then
    Error( "Usage: Green( <S>, <type> ) where <S> must be a transformation",
           "\nsemigroup and <type> must be one of the following: ",
           "\"l\",\"r\",\"j\",\"d\", or \"h\" ");
  fi;

  if type in [ "l","L" ] then
    if not IsBound( S.greenL ) then
      S.greenL := S.operations.GreenL( S );
    fi;
    return S.greenL;
  elif type in [ "r","R" ] then
    if not IsBound( S.greenR ) then
      S.greenR := S.operations.GreenR( S );
```

```
    fi;
    return S.greenR;
  elif type in [ "j","J" ] then
    if not IsBound( S.greenJ ) then
      S.greenJ := S.operations.GreenJ( S );
    fi;
    return S.greenJ;
  elif type in [ "d","D" ] then
    if not IsBound( S.greenD ) then
      S.greenD := S.operations.GreenD( S );
    fi;
    return S.greenD;
  elif type in [ "h","H" ] then
    if not IsBound ( S.greenH ) then
      S.greenH := S.operations.GreenH( S );
    fi;
    return S.greenH;
  else
    Error( "panic, should never come here" );
  fi;

  return -1;
end;

#############################################################################
##
#F  SgOps.GreenL( <S> ). . . . . Green's left relation of a transformation Sg
##
SgOps.GreenL := function( S )
  local l,       # this holds the equivalence classes of Greens left relation
        tf,      # help variable: a transformation in S
        classes, # set of classes
        class,cl,# one equivalence class of Green's left relation
        elms;    # holds the elements of S

  elms := Elements( S );

  # make a pre - classification
  classes := [];
  for tf in elms do
    AddSet( classes, Filtered( elms, t -> Ker( t ) = Ker( tf ) ) );
  od;
  l := Copy( classes );

  # refine the equivalence relation if necessary
  cl := Length( S.generators[1].source );
  if Size( S ) < cl^cl then
    l := [];
    for class in classes do
      elms := Copy( class );
      repeat
        tf := elms[1];
```

```
        cl := Filtered( elms, t -> S.operations.PrincipalLeftIdeal( S, t ) =
                                   S.operations.PrincipalLeftIdeal( S, tf ) );
        AddSet( l, cl );
        SubtractSet( elms, cl );
      until elms = [];
    od;
  fi;

  return l;
end;

#############################################################################
##
#F  SgOps.GreenR( <S> ) . . . . Green's right relation of a transformation Sg
##
SgOps.GreenR := function( S )
  local r,       # this holds the equivalence classes of Greens right relation
        tf,      # help variable: a transformation in S
        classes, # set of classes
        class,cl,# one equivalence class of Green's right relation
        elms;    # holds the elements of S

  elms := Elements( S );

  # make a pre - classification
  classes := [];
  for tf in elms do
    AddSet( classes, Filtered( elms, t -> Image( t ) = Image( tf ) ) );
  od;
  r := Copy( classes );

  # refine the equivalence relation if necessary
  cl := Length( S.generators[1].source );
  if Size( S ) < cl^cl then
    r := [];
    for class in classes do
      elms := Copy( class );
      repeat
        tf := elms[1];
        cl := Filtered( elms, t -> S.operations.PrincipalRightIdeal( S, t ) =
                                   S.operations.PrincipalRightIdeal( S, tf )
                      );
        AddSet( r, cl );
        SubtractSet( elms, cl );
      until elms = [];
    od;
  fi;

  return r;
end;

#############################################################################
```

```
##
#F  SgOps.GreenJ( <S> ). . . . . . . join of Green's left and right relations
##
SgOps.GreenJ := function( S )
  local j,       # this holds the equivalence classes of Greens D relation
        l,r,     # Green's L resp. R relations
        class,   # a class of Green's left relation
        cl,      # a class of Green's J relation
        elm;     # help variable, an element

  l := Green( S, "L" );
  r := Green( S, "R" );

  j := [];
  for class in l do
    cl := [];
    for elm in class do
      UniteSet( cl, First( r, c -> elm in c ) );
    od;
    AddSet( j, cl );
  od;

  return j;
end;

#############################################################################
##
#F  SgOps.GreenD( <S> ). . . . . . . . . . . . . . . . . . Green's D relation
##  D = J in a finite semigroup!
##
SgOps.GreenD := SgOps.GreenJ;

#############################################################################
##
#F  SgOps.GreenH( <S> ) . . . . Intersection of Green's left and right rel's.
##
SgOps.GreenH := function( S )
  local h,        # this holds the equivalence classes of Greens H relation
        l,r,      # Green's L resp. R relations
        cl,       # a class of Green's h relation
        cl_left,  # a class of Green's L relation
        cl_right, # a class of Green's right relation
        elm,      # help variable, an element
        elms;     # holds the elements of S

  elms := Copy( Elements( S ) );
  l := Green( S, "L" );
  r := Green( S, "R" );

  h := [];
  repeat
    elm := elms[1];
```

```
    cl_left  := First( l, cl -> elm in cl );
    cl_right := First( r, cl -> elm in cl );
    cl := IntersectionSet( cl_left, cl_right );
    AddSet( h, cl );
    SubtractSet( elms, cl );
  until elms = [];

  return h;
end;

#############################################################################
##
#F  SgOps.PrintTable( <S> ) . . . . print a Cayley table of the semigroup <S>
##
SgOps.PrintTable := function( S )

  local elms,    # the elements of the semigroup
        n,       # the size of the semigroup
        symbols, # a list of the symbols for the elements of the semigroup
        tw,      # the width of a table
        spc,     # local function which prints the right number of spaces
        bar,     # local function for printing the right length of the bar
        ind,     # help variable, an index
        i,j;     # loop variables

  elms    := Elements( S );
  n       := Size( elms );
  symbols := [ AbstractGenerator( "s0" ) ];
  if n > 1 then
    symbols := Concatenation( [ AbstractGenerator( "s0" ) ],
                              AbstractGenerators( "s" , n-1 ) );
  fi;
  # compute the number of characters per line required for the table
  if n < 11 then tw := 3*n + 6; else tw := 4*n + 8; fi;
  spc     := function( i, max )
               if max < 11 then return " "; fi;
               if i < 11 then return "  "; else return " "; fi;
             end;
  bar     := function( max )
               if max < 11 then return "---"; else return "----"; fi;
             end;

  if SizeScreen()[1] - 3 < tw then
    Print( "The table of a semigroup of order ", n, " will not ",
           "look\ngood on a screen with ", SizeScreen()[1], " characters per ",
           "line.\nHowever, you may want to set your line length to a ",
           "greater\nvalue by using the GAP function 'SizeScreen'.\n" );
    return;
  fi;
  Print( "Let:\n" );
  for i in [1..n] do Print( symbols[i], " := ", elms[i], "\n" ); od;
```

```
  # print the multiplication table
  Print( "\n\n  * ", spc( 0, n ), "| " );
    for i in [1..n] do Print( symbols[i], spc( i, n ) ); od;
  Print( "\n ---", bar( n ) ); for i in [1..n] do Print( bar( n ) ); od;
  for i in [1..n] do
    Print( "\n  ", symbols[i], spc( i, n ), "| " );
    for j in [1..n] do
      ind := Position( elms, S.multiplication( elms[i], elms[j] ) );
      Print( symbols[ ind ], spc( ind, n ) );
    od;
  od;

  Print( "\n\n" );
end;

#############################################################################
##
#F  SgOps.Identity( <S> ) . . . . . . . . compute identity of a semigroup <S>
##
SgOps.Identity := function( S )
  local id, elms;

  id := IdentityTransformation( S.generators[1].source );
  if id in Elements( S ) then
    return [ id ];
  else
    elms := Elements( S );
    for id in elms do
      if ForAll( elms, e -> e*id = e and id*e = e ) then
        return [ id ];
      fi;
    od;
  fi;

  return [ ];
end;

#############################################################################
##
#F  SgOps.IsCommutative( <S> ). . . . . . . . . . . test commutativity of <S>
##
SgOps.IsCommutative := function( S )
  local mul, elms;

  mul  := S.multiplication;
  elms := Elements( S );

  return ForAll( elms, c -> ForAll( elms, e -> mul( c, e ) = mul( e, c ) ) );
end;

#############################################################################
##
```

```
#F  SgOps.IdempotentElements( <S> ). . . . . . . . . . idempotent elms of <S>
##
SgOps.IdempotentElements := function( S )
  local mul;

  mul  := S.multiplication;

  return Filtered( Elements( S ), i -> mul( i, i ) = i );

end;

#############################################################################
##
##  num2fun:
##  convert a number representing a function as occurring in the semigroup
##  library SGLIB.G into the corresponding function.
##
num2fun := function( num, b )
  local f, remainder, i;
  f := []; num := num - 1 ;
  for i in [1..b] do
    remainder := num mod b;
    num := ( num - remainder ) / b;
    Add( f, remainder + 1 );
  od;
  return Reversed( f );
end;

#############################################################################
##
#F  LibrarySemigroup: function extracts a semigroup from the semigroups
##                    library file SG.LIB
##  V1.1 1.3.95.
##
LibrarySemigroup := function( order, type )
  local i,j,          # loop variables
        clmax,        # the total number of isomorphism classes in a sg
        gens,         # the generators (=elements) of the semigroup
        f,            # the function representing the semigroup
        rho,          # a transformation being constructed
        af,           # help variable: containing the relevant functions
        sg;           # help variable: containing semigroup data

  if not ( IsInt( order ) and IsInt( type ) and order > 0 and type > 0 ) then
    Error( "Usage: LibrarySemigroup( <order>, <type> ) where <order> and\n",
    "      <type> must both be positive integers" );
  fi;

  if   ( order = 1 ) then
    sg := SG1;
  elif ( order = 2 ) then
    sg := SG2;
```

```
elif ( order = 3 ) then
  sg := SG3;
elif ( order = 4 ) then
  sg := SG4;
elif ( order = 5 ) then
  sg := SG5;
else
  Print( "There are only orders 1 to 5 in the semigroups library.\n" );
  return;
fi;

clmax := Length( RecFields( sg ) );
if type > clmax then
  Print( "There are only ", clmax,
         " isomorphism classes of semigroups of order ", order, ".\n" );
  return;
fi;

f    := sg.(type).phi;
gens := [];
af   := [];

for i in [1..order] do
  if not IsBound( af[f[i]] ) then
    af[f[i]] := num2fun( f[i], order );
  fi;
od;

# check if the semigroup has an identity
i := First( [1..order+1], j -> j = order+1 or
            ( af[f[j]] = [1..order] and
              ForAll( [1..order], x -> af[f[x]][j] = x ) )
          );

if i <= order then

  # the case if the sg has an identity

  for i in [1..order] do
    rho := [];
    for j in [1..order] do
      Add( rho, af[f[j]][i] );
    od;
    Add( gens, Transformation( [1..order], rho ) );
  od;

else

  # the case if the sg has no identity

  for i in [1..order] do
    rho := [];
```

```
      for j in [1..order] do
        Add( rho, af[f[j]][i] );
      od;
      Add( rho, i );
      Add( gens, Transformation( [1..order+1], rho ) );
    od;

  fi;

  return TransformationSemigroup( gens );
end;

#############################################################################
##
#F  AllLibrarySemigroups: this function extracts all semigroups of a
##  specified class from the semigroups library file SG.LIB
##  V1.0 14.2.95.
##
AllLibrarySemigroups := function( order, type )
  local i,j,          # loop variables
        clmax,        # the total number of isomorphism classes in a sg
        gens,         # the generators (=elements) of the semigroup
        f,            # the function representing the semigroup
        autos,a,      # automorphisms
        rho,          # a transformation being constructed
        af,           # help variable: containing the relevant functions
        sgps,         # return value: a list of semigroups
        sg;           # help variable: containing semigroup data

  if not ( IsInt( order ) and IsInt( type ) and order > 0 and type > 0 ) then
    Error( "Usage: AllLibrarySemigroups( <order>, <type> ) where <order>",
           "and\n      <type> must both be positive integers" );
  fi;

  if   ( order = 1 ) then
    sg := SG1;
  elif ( order = 2 ) then
    sg := SG2;
  elif ( order = 3 ) then
    sg := SG3;
  elif ( order = 4 ) then
    sg := SG4;
  elif ( order = 5 ) then
    sg := SG5;
  else
    Print( "There are only orders 1 to 5 in the semigroups library.\n" );
    return;
  fi;

  clmax := Length( RecFields( sg ) );
  if type > clmax then
    Print( "There are only ", clmax,
```

```
        " isomorphism classes of semigroups of order ", order, ".\n" );
  return;
fi;

f     := sg.(type).phi;
af    := [];
sgps  := [];
autos := [];
for a in sg.(type).bijs_yielding_iso_sgps do
  Add( autos, num2fun( a, order ) );
od;

for i in [1..order] do
  if not IsBound( af[f[i]] ) then
    af[f[i]] := num2fun( f[i], order );
  fi;
od;

# check if the semigroup has an identity
i := First( [1..order+1], j -> j = order+1 or
            ( af[f[j]] = [1..order] and
              ForAll( [1..order], x -> af[f[x]][j] = x ) ) );

if i <= order then

  # the case if the sg has an identity

  for a in autos do
    gens := [];
    for i in [1..order] do
      rho := [];
      for j in [1..order] do
        rho[a[j]] := a[ af[f[j]][i] ];
      od;
      Add( gens, Transformation( [1..order], rho ) );
    od;
    AddSet( sgps, TransformationSemigroup( gens ) );
  od;

else

  # the case if the sg has no identity

  for a in autos do
    gens := [];
    for i in [1..order] do
      rho := [];
      for j in [1..order] do
        rho[a[j]] := a[ af[f[j]][i] ];
      od;
      Add( rho, a[i] );
      Add( gens, Transformation( [1..order+1], rho ) );
```

```
      od;
      AddSet( sgps, TransformationSemigroup( gens ) );
    od;

  fi;

  return sgps;
end;
```

B.5 The Source File `nr.g`

```
################################################################################
##  File: NR.G                                                                ##
################################################################################
##
################################################################################
##  Provide some GAP library functions for nearrings
################################################################################
##

if not IsBound( NR_PATH_NAME ) then
  NR_PATH_NAME := ReplacedString( LIBNAME, "lib", "nrsg/nr" );
fi;

ReadNr := function ( name )
  if not ReadPath( NR_PATH_NAME, name, ".nr", "ReadNr" )  then
    Error( "the nearring library file '",
           name, ".nr' must exist and be readable");
  fi;
end;

AUTO( ReadNr( "nr_2_7" ),
      NR_C2, NR_C3, NR_C4, NR_V4, NR_C5, NR_C6, NR_S3, NR_C7 );

AUTO( ReadNr( "nr8_1" ), NR_C8 );
AUTO( ReadNr( "nr8_2" ), NR_C2xC4 );
AUTO( ReadNr( "nr8_3" ), NR_C2xC2xC2 );
AUTO( ReadNr( "nr8_4" ), NR_D8 );
AUTO( ReadNr( "nr8_5" ), NR_Q8 );

AUTO( ReadNr( "nr9_1" ), NR_C9 );
AUTO( ReadNr( "nr9_2" ), NR_C3xC3 );

AUTO( ReadNr( "nr10_1" ), NR_C10 );
AUTO( ReadNr( "nr10_2" ), NR_D10 );

AUTO( ReadNr( "nr11_1" ), NR_C11 );

AUTO( ReadNr( "nr12_1" ), NR_C12 );
AUTO( ReadNr( "nr12_2" ), NR_C2xC6 );
```

```
AUTO( ReadNr( "nr12_3" ), NR_D12 );
AUTO( ReadNr( "nr12_4" ), NR_A4 );
AUTO( ReadNr( "nr12_5" ), NR_T );

AUTO( ReadNr( "nr13_1" ), NR_C13 );

AUTO( ReadNr( "nr14_1" ), NR_C14 );
AUTO( ReadNr( "nr14_2" ), NR_D14 );

AUTO( ReadNr( "nr15_1" ), NR_C15 );

#############################################################################
##
#V  NearringOps . . . . . . . . . . . . . . . operations record for nearrings
##
##  'NearringOps' is the operation record for nearrings. This is initially
##  a copy of 'DomainOps'. This way all the default methods for domains are
##  inherited.
##
NearringOps := Copy( DomainOps );

#############################################################################
##
##  IsNearring( <obj> ) . . . . . . . . . . . . . test if <obj> is a nearring
##
IsNearring := function( obj )
  return IsRec( obj ) and IsBound( obj.isNearring ) and obj.isNearring;
end;

#############################################################################
##
##  IsTransformationNearring( <obj> ). . . . . test if <obj> is a tf nearring
##
IsTransformationNearring := function( obj )
  return IsRec( obj ) and
    IsBound( obj.isTransformationNearring ) and obj.isTransformationNearring;
end;

#############################################################################
##
#F  PrintTable( <D> ) . . . . . . nicely print a Cayley table of a domain <D>
##
##  Dispatcher function for printing a Cayley table of <D>.
##  Works for <D> = a semigroup or a group or a nearring.
##
PrintTable := function( D )
  if IsGroup( D ) or IsNearring( D ) or IsSemigroup( D ) then
    D.operations.PrintTable( D );
  else
    Error( "Usage: PrintTable( <D> ) where <D> must be a semigroup or a ",
           "group\nor a nearring" );
  fi;
```

```
end;

################################################################################
##
#F  Identity( <D> ). . . . . . . . . . . . . compute identity of a domain <D>
##
##  Dispatcher function to compute the identity of a domain <D> where <D> is
##  a nearring or a semigroup.
##
Identity := function( D )

  if not ( IsTransformationSemigroup( D ) or IsNearring( D ) ) then
    Error( "Usage: Identity( <D> ) where <D> must be a",
           " nearring or a\ntransformation semigroup" );
  fi;

  if not IsBound( D.identity ) then
    D.identity := D.operations.Identity( D );
  fi;

  return D.identity;
end;

################################################################################
##
#F  IsCommutative( <D> ) . . . . . . . . . test commutativity of a domain <D>
##
##  Dispatcher function to test if a domain <D> is commutative where <D> is
##  a nearring or a semigroup.
##
IsCommutative := function( D )

  if not ( IsTransformationSemigroup( D ) or IsNearring( D ) ) then
    Error( "Usage: IsCommutative( <D> ) where <D> must be a",
           " nearring or a\ntransformation semigroup" );
  fi;

  if not IsBound( D.isCommutative ) then
    D.isCommutative := D.operations.IsCommutative( D );
  fi;

  return D.isCommutative;
end;

################################################################################
##
#F  Endomorphisms( <D> ). . . . . . . . . . create all endo's on a domain <D>
##  V1.0  20.2.95
##  Dispatcher function for computing all endomorphisms on <D>.
##  Works for <D> = a group or a nearring.
##
Endomorphisms := function( D )
```

```
  if IsGroup( D ) or IsNearring( D ) then
    if not IsBound( D.endomorphisms ) then
      D.endomorphisms := D.operations.Endomorphisms( D );
    fi;
  else
    Error( "Usage:Endomorphisms( <D> ) where <D> must be a group or a ",
           "nearring" );
  fi;

  return D.endomorphisms;
end;

#############################################################################
##
#F  Automorphisms( <D> ). . . . . . . . . . create all auto's on a domain <D>
##  V1.0  20.2.95
##  Dispatcher function for computing all automorphisms on <D>.
##  Works for <D> = a group or a nearring.
##
Automorphisms := function( D )

  if IsGroup( D ) or IsNearring( D ) then
    if not IsBound( D.automorphisms ) then
      D.automorphisms := D.operations.Automorphisms( D );
    fi;
  else
    Error( "Usage:Automorphisms( <D> ) where <D> must be a group or a ",
           "nearring" );
  fi;

  return D.automorphisms;
end;

#############################################################################
##
##  IsNrMultiplication( <G>, <mul> ). . . . . . . test if <mul> is a nearring
##                                                multiplication on <G>
##
##  This function tests if a specified multiplication function <mul> is a
##  nearring multiplication on the group <G>.
##
IsNrMultiplication := function( G, mul )
  local elms;
  elms := Elements( G );

  # check that the arguments are really a group and a WELL-DEFINED function
  if not ( IsGroup( G ) and IsFunc( mul ) and
           ForAll( elms, n1 -> ForAll( elms, n2 -> mul( n1, n2 ) in G ) )
         ) then
    Error( "Usage: IsNrMultiplication( <G>, <mul> ) where <G> must be a ",
           "group\nand <mul> must be a function <G> x <G> -> <G>" );
```

```
  fi;

  # check if mul is ASSOCIATIVE
  if not ForAll( elms, n1 -> ForAll( elms, n2 -> ForAll( elms, n3 ->
    mul( n1, mul( n2, n3 ) ) = mul( mul( n1, n2 ), n3 ) ) ) )
  then
    Print( "specified multiplication is not associative.\n" );
    return false;
  fi;

  # check if mul is RIGHT DISTRIBUTIVE
  # (note that the addition is denoted by '*' )
  if not ForAll( elms, n1 -> ForAll( elms, n2 -> ForAll( elms, n3 ->
    mul( n1 * n2 , n3 ) =  mul( n1, n3 ) * mul( n2, n3 )  ) ) )
  then
    Print( "specified multiplication is not right distributive.\n" );
    return false;
  fi;

  return true;
end;

#############################################################################
##
#F  Nearring( <arg> ) . . . . . . . . . . . . . . . . . . . . create a nearring
##
##  Constructor function for a nearring
##  So far there are two possibilities to construct a nearring:
##  1.) enter a group and a nearring multiplication on this group.
##  2.) enter a few group transformations and consider the generated
##      nearring. ( in this case Elements(.) is the most important function )
##
Nearring := function( arg )
  local G,    # the additive group of a nearring to be defined
        mul,  # a multiplication which makes G into a nearring
        gens, # generators of a nearring
        gt,   # a group transformation
        NR;   # the nearring to be returned

  arg := Flat( arg );
  if  Length( arg ) in [ 2, 3 ]  and IsGroup( arg[1] ) then

    if Length( arg ) = 2 then
      if not IsNrMultiplication( arg[1], arg[2] ) then return; fi;
    fi;
    G   := arg[1];
    mul := arg[2];
    NR  := rec();
    # enter category components
    NR.isDomain       := true;
    NR.isNearring     := true;
    # enter identification components
```

```
    NR.group          := G;
    NR.addition       := function( x, y ) return x * y;    end;
    NR.subtraction    := function( x, y ) return x * y^-1; end;
    NR.multiplication := mul;
    # enter operations record
    NR.operations     := NearringOps;

  elif ForAll( arg, gt -> IsGroupTransformation( gt ) and
             gt.source = arg[1].source and gt.range = arg[1].range
           ) then
    # make sure that all sources and ranges of the generators are not only
    # equal but identical.
    gens := [];
    for gt in arg do
      gt.source := arg[1].source; gt.range := arg[1].source;
      AddSet( gens, gt );
    od;
    NR  := rec();
    # enter category components
    NR.isDomain                 := true;
    NR.isNearring               := true;
    NR.isTransformationNearring := true;
    # enter identification components
    NR.generators               := gens;
    NR.group                    := "?";
    NR.addition                 := function( x, y ) return x + y; end;
    NR.subtraction              := function( x, y ) return x - y; end;
    NR.multiplication           := function( x, y ) return x * y; end;
    # enter operations record
    NR.operations     := NearringOps;

  else
    Error( "Usage: Nearring( <G>, <mul> ) where <G> must be a group\n",
           "and <mul> must be a valid function <G> x <G> -> <G> or",
           "\nNearring( <t1>, <t2>, ... ) where all arguments must ",
           "be\ntransformations on the same group" );
  fi;
  return NR;
end;

#############################################################################
##
#F  NearringOps.Elements( <N> ). . . compute the elements of the nearring <N>
##
NearringOps.Elements := function( N )
  local closure,       # the constructed closure
        elms,          # the elements of the group on which the tf's operate
        l,             # the number of the elements
        firstrun,      # help var: indicates if first loop run
        tfl,tfl1,tfl2, # help vars: transformation lists
        elmset,        # help var: contains the elms curr. in the closure
        changed_elmset, # help var: indicates if elmset has changed
```

```
    done;          # help var: indicates, when it is time to stop

if IsTransformationNearring( N ) then

  # perform a simple closure algorithm
  closure  := Set( List( N.generators, g -> g.tfl ) );
  elms     := N.generators[1].elements;
  l        := Length( elms );
  firstrun := true;

  repeat
    # step 1: add all sums
    Print( "Step 1: Building sums of full transformations\n" );
    elmset         := closure;
    changed_elmset := true;
    done           := true;
    while changed_elmset do
      closure        := Copy( elmset );
      changed_elmset := false;
      for tfl1 in closure do
        for tfl2 in closure do
          tfl := List( [1..l], i ->
                   Position( elms, elms[ tfl1[i] ] * elms[ tfl2[i] ] ) );
          if not tfl in elmset then
            AddSet( elmset, tfl );
            changed_elmset := true;
            done           := false;
          fi;
        od;
      od;
    od;
    if not done or firstrun then
      # step 2 build the multiplicative closure
      Print( "Step 2: Multiplying full transformations\n" );
      firstrun       := false;
      elmset         := closure;
      changed_elmset := true;
      done           := true;
      while changed_elmset do
        closure        := Copy( elmset );
        changed_elmset := false;
        for tfl1 in closure do
          for tfl2 in closure do
            tfl := tfl1{ tfl2 };
            if not tfl in elmset then
              AddSet( elmset, tfl );
              changed_elmset := true;
              done           := false;
            fi;
          od;
        od;
      od;
```

```
      fi;
    until done;
    N.elements := [];
    for tfl in closure do
      AddSet( N.elements, Transformation( N.generators[1].source, tfl ) );
    od;

  else

    N.elements := Elements( N.group );

  fi;

  return N.elements;
end;

#############################################################################
##
#F  NearringOps.Print( <N> ) . . . . . . . . . . . . . . . . print a nearring
##
NearringOps.Print := function( N )
  local i;
  if IsTransformationNearring( N ) then
    Print( "TransformationNearring( " );
    for i in [ 1..Length( N.generators ) - 1 ] do
      Print( N.generators[i] );
      if TRANSFORMATION_PRINT_LEVEL = 0 then Print( ", " ); fi;
    od;
    Print( N.generators[ Length( N.generators )], " ) " );
  else
    Print( "Nearring( ", N.group, ", multiplication( x, y ) )" );
  fi;
end;

#############################################################################
##
#F  FindGroup( <N> ). . . determine the additive group of a transformation nr
##                        <N> as a GAP permutation group.
##
##  Find the embedding which maps the additive group of transformations of
##  a transformation nearring <N> into the symmetric group Sn of all
##  permutations on {1..n} where n is the size of the nearring.
##  The record fields 'group' and 'PHI' will be added to N.
##  The list PHI is a list of the permutations of the subgroup of Sn
##  such that there is a 1-1 correspondence between the list
##  elms := Elements( N ) ( which is identical to N.elements ) and the
##  list PHI s.t. PHI[1] = elms[1], PHI[2] = elms[2], ... ,PHI[n] = elms[n].
##
FindGroup := function( N )
  local TFL,   # the list of the transformation lists of the tf's of N
        telms, # the elms of the group a tf of N works on ( "transf'd elms" )
        s,     # the size of the group a tf in N works on
```

```
        PHI,   # the list of all permutations, return value
        tfl1,  # a transformation list of a transformation in N
        phi;   # the permutation derived for one fixed transformation in N

  if not IsTransformationNearring( N ) then
    Error( "Usage: FindGroup( <N> ) where <N> must be a transformation ",
           "nearring" );
  fi;
  if not ( IsBound( N.group ) and IsBound( N.PHI ) ) then

    TFL   := List( Elements( N ), e -> e.tfl );
    telms := Elements( N.generators[1].source );
    s     := Size( telms );
    PHI   := [];

    for tfl1 in TFL do
      phi := PermList( List( TFL, tfl2 ->
              Position( TFL, List( [1..s], i ->
                Position( telms, telms[ tfl1[i] ] * telms[ tfl2[i] ] )
                     ))    )     );
      Add( PHI, phi );
    od;

    N.group := Group( SmallestGeneratingSystem( Group( PHI, () ) ), () );
    N.PHI   := PHI;
  fi;
  return N.group;
end;

#############################################################################
##
#F  NearringOps.Identity( <N> ) . . . . . . . . . . . compute identity of <N>
##
NearringOps.Identity := function( N )
  local mul, elms;

  mul  := N.multiplication;
  elms := Elements( N );

  return Filtered( elms, i -> ForAll( elms, e -> mul( i, e ) = e and
                                                 mul( e, i ) = e ) );
end;

#############################################################################
##
#F  NearringOps.IsCommutative( <N> ). . . . . . . . test commutativity of <N>
##
NearringOps.IsCommutative := function( N )
  local mul, elms;

  mul  := N.multiplication;
  elms := Elements( N );
```

```
  return ForAll( elms, c -> ForAll( elms, e -> mul( c, e ) = mul( e, c ) ) );
end;

#############################################################################
##
#F  NearringOps.Endomorphisms( <N> ). . . create all endo's on a nearring <N>
##
NearringOps.Endomorphisms := function( N )
  local elms,endos;
  if IsTransformationNearring( N ) then
    FindGroup( N );
  fi;
  elms  := Elements( N );
  endos := Filtered( Endomorphisms( N.group ), e ->
    ForAll( elms, x -> ForAll( elms, y ->
      elms[ e.tfl[ Position( elms, N.multiplication( x, y ) ) ]  ] =
      N.multiplication( elms[ e.tfl[ Position( elms, x ) ] ],
                        elms[ e.tfl[ Position( elms, y ) ] ] ) ) ) );
  return endos;
end;

#############################################################################
##
#F  NearringOps.Automorphisms( <N> ). . . create all auto's on a nearring <N>
##
NearringOps.Automorphisms := function( N )
  local elms,autos;
  if IsTransformationNearring( N ) then
    FindGroup( N );
  fi;
  elms  := Elements( N );
  autos := Filtered( Automorphisms( N.group ), e ->
    ForAll( elms, x -> ForAll( elms, y ->
      elms[ e.tfl[ Position( elms, N.multiplication( x, y ) ) ]  ] =
      N.multiplication( elms[ e.tfl[ Position( elms, x ) ] ],
                        elms[ e.tfl[ Position( elms, y ) ] ] ) ) ) );
  return autos;
end;

#############################################################################
##
#F  NearringOps.PrintTable( <arg> ). . . . . . . print a Cayley table of a nr
##
NearringOps.PrintTable := function( arg )

  local N,       # the nearring
        elms,    # the elements of the nearring
        n,       # the size of the nearring
        symbols, # a list of the symbols for the elements of the nearring
        tw,      # the width of a table
        spc,     # local function which prints the right number of spaces
```

```
      bar,     # local function for printing the right length of the bar
      ind,     # help variable, an index
      print_addition, print_multiplication, # status variables
      i,j;     # loop variables

print_addition := true; print_multiplication := true;
if Length( arg ) = 2 then
  if arg[ 2 ] = "a" or arg[ 2 ] = "A" then
    print_multiplication := false;
  else
    print_addition := false;
  fi;
fi;
N       := arg[ 1 ];
elms    := Elements( N );
n       := Size( elms );
symbols := [ AbstractGenerator( "n0" ) ];
if n > 1 then
  symbols := Concatenation( [ AbstractGenerator( "n0" ) ],
                            AbstractGenerators( "n" , n-1 ) );
fi;
# compute the number of characters per line required for the table
if n < 11 then tw := 3*n + 6; else tw := 4*n + 8; fi;
spc     := function( i, max )
             if max < 11 then return " "; fi;
             if i < 11 then return "  "; else return " "; fi;
           end;
bar     := function( max )
             if max < 11 then return "---"; else return "----"; fi;
           end;

if SizeScreen()[1] - 3 < tw then
  Print( "The table of a nearring of order ", n, " will not look\n",
         "good on a screen with ", SizeScreen()[1], " characters per ",
         "line.\nHowever, you may want to set your line length to a ",
         "greater\nvalue by using the GAP function 'SizeScreen'.\n" );
  return;
fi;
if IsTransformationNearring( N ) then FindGroup( N ); fi;
if print_addition and print_multiplication then
  Print( "Let:\n" );
  for i in [1..n] do
    Print( symbols[i], " := ", elms[i] );
    if IsTransformationNearring( N ) and TRANSFORMATION_PRINT_LEVEL = 0 then
      Print( " ( <-> ", Elements( N.group )[i], " )" );
    fi;
    Print( "\n" );
  od;
fi;

if print_addition then
  # print the addition table
```

```
    Print( "\n  + ", spc( 0, n ), "| " );
      for i in [1..n] do Print( symbols[i], spc( i, n ) ); od;
    Print( "\n ---", bar( n ) ); for i in [1..n] do Print( bar( n ) ); od;
    for i in [1..n] do
      Print( "\n  ", symbols[i], spc( i, n ), "| " );
      for j in [1..n] do
        ind := Position( elms, N.addition( elms[i], elms[j] ) );
        Print( symbols[ ind ], spc( ind, n ) );
      od;
    od;
  fi;

  if print_multiplication then
    # print the multiplication table
    Print( "\n\n  * ", spc( 0, n ), "| " );
      for i in [1..n] do Print( symbols[i], spc( i, n ) ); od;
    Print( "\n ---", bar( n ) ); for i in [1..n] do Print( bar( n ) ); od;
    for i in [1..n] do
      Print( "\n  ", symbols[i], spc( i, n ), "| " );
      for j in [1..n] do
        ind := Position( elms, N.multiplication( elms[i], elms[j] ) );
        Print( symbols[ ind ], spc( ind, n ) );
      od;
    od;
  fi;

  Print( "\n\n" );
end;

#############################################################################
##
##  NearringOps.IsNrIdeal( <N>, <I> ) . . . . check if <I> is an ideal of <N>
##
NearringOps.IsNrIdeal := function( N, I )

  local isid,  # return value: a record with the boolean record fields:
               #               isLeftIdeal, isRightIdeal, isIdeal.
        ielms; # help var: the elements of  the subgroup being considered

  if not IsNearring( N ) then
    Error( N, " must be a nearring" );
  fi;
  if IsTransformationNearring( N ) then
    FindGroup( N );
  fi;
  if not IsSubgroup( N.group, I ) then
    Error( I, " must be a subgroup of ", N, ".group" );
  fi;
  if not IsNormal( N.group, I ) then
    Error( I, " must be a normal subgroup of ", N, ".group" );
  fi;
```

```
  # in case of a transformation nearring take those elements
  # (=transformations) of N which form the subgroup
  if IsTransformationNearring( N ) then
    ielms := Filtered( Elements( N ), n ->
      Elements( N.group )[ Position( Elements( N ), n ) ] in Elements( I ) );
  else
    ielms := Elements( I );
  fi;

  isid := rec(
    isLeftIdeal := false,
    isRightIdeal := false,
    isIdeal := false );

  if ForAll( Elements( N ), n -> ForAll( ielms, i ->
             N.multiplication( i, n ) in ielms ) )
  then
    isid.isRightIdeal := true;
  fi;

  if ForAll( Elements( N ), m -> ForAll( Elements( N ), n ->
     ForAll( ielms, i ->
             N.subtraction(
             N.multiplication( m, N.addition( n, i) ),
             N.multiplication( m, n ) ) in ielms ) ) )
  then
    isid.isLeftIdeal := true;
  fi;

  if isid.isLeftIdeal and isid.isRightIdeal then
    isid.isIdeal := true;
  fi;

  return isid;
end;

#############################################################################
##
##  NearringIdeals( <arg> ). . . . . . . . . . . . compute all ideals of a nr
##
NearringIdeals := function( arg )

  local N,          # the nearring
        L,          # the lattice of subgroups of the add. group
        Rep,        # representative of the <i>-th class
        normalizer, # normalizer of <I> in <N.group>
        reps,       # transversal of <normalizer> in <N.group>
        I,          # the subgroup being considered
        elms_I,     # the elements of the subgroup M
        elms_N,     # the elements of the (group of the) nearring
        ideals,     # the list of ideals
        add,
```

```
      sub,
      mul,        # the nearring multiplication
      right_only,
      left_only,
      i,k;        # loop variables

if not (
         Length( arg ) = 1 and IsNearring( arg[1] ) or (
         Length( arg ) = 2 and
         ( arg[2] = "l" or arg[2] = "L" or arg[2] = "r" or arg[2] = "R" ) )
       )
         then
  Error( "Usage: NearringIdeals( <N> ) or NearringIdeals( <N>, <\"r\"> )\n",
         "or NearringIdeals( <N>, <\"l\"> ) where <N> must be a nearring" );
fi;

N := arg[ 1 ]; right_only := false; left_only := false;
if Length( arg ) = 2 then
  if arg[2] = "r" or arg[2] = "R" then
    right_only := true;
  else
    left_only := true;
  fi;
fi;

if IsTransformationNearring( N ) then FindGroup( N ); fi;

add    := N.addition;
sub    := N.subtraction;
mul    := N.multiplication;
ideals := [ Subgroup( N.group, [ ] ) ];
elms_N := Elements( N );
L      := Lattice( N.group );

for i in [ 2..Length( L.classes )-1 ] do

  Rep := L.classes[ i ].representative;
  # get the transversal
  normalizer := Normalizer( L.group, ShallowCopyNoSC(Rep) );
  reps := RightTransversal( L.group, ShallowCopyNoSC(normalizer) );

  # consider all normal subgroups of N.group
  #   for k in [ 1..Length( reps ) ] do
  if Length( reps ) = 1 then
  #     I := Rep^reps[ k ];
    I := Rep;
    if IsNormal( N.group, I ) then

      elms_I := List( Elements( I ), e ->
                elms_N[ Position( Elements( N.group ), e ) ] );

      # this is the check for the (right) (left) ideal condition
```

```
      if right_only then
        if ForAll( elms_I, i -> ForAll( elms_N, n ->
           mul( i, n ) in elms_I ) ) then
          Add( ideals, I );
        fi;
      elif left_only then
        if ForAll( elms_N, n -> ForAll( elms_N, m -> ForAll( elms_I, i ->
           sub( mul( n, add( m, i ) ), mul( n, m ) ) in elms_I ) ) ) then
          Add( ideals, I );
        fi;
      else
        if ForAll( elms_I, i -> ForAll( elms_N, n ->
                   mul( i, n ) in elms_I ) ) and
          ForAll( elms_N, n -> ForAll( elms_N, m -> ForAll( elms_I, i ->
          sub( mul( n, add( m, i ) ), mul( n, m ) ) in elms_I ) ) ) then
          Add( ideals, I );
        fi;
      fi;
    fi;

  #   od;
  fi;

  od;

  Add( ideals, N.group );

  return ideals;
end;

#############################################################################
##
##  InvariantSubnearrings( <N> ). . . . . . . compute all inv subnr's of <N>.
##
InvariantSubnearrings := function( N )

  local L,          # the lattice of subgroups of the add. group
        Rep,        # representative of the <i>-th class
        normalizer, # normalizer of <I> in <N.group>
        reps,       # transversal of <normalizer> in <N.group>
        M,          # the subgroup being considered
        elms_M,     # the elements of the subgroup M
        elms_N,     # the elements of the (group of the) nearring
        inv_sub_nrs,# the list of invariant subnearrings
        mul,        # the nearring multiplication
        i,k;        # loop variables

  if not IsNearring( N ) then
    Error( "Usage: InvariantSubnearrings( <N> ) where <N> must be a ",
           "nearring" );
  fi;
```

```
  if IsTransformationNearring( N ) then FindGroup( N ); fi;

  mul         := N.multiplication;
  inv_sub_nrs := [];
  elms_N      := Elements( N );
  L           := Lattice( N.group );

  for i in [1..Length(L.classes)] do

    Rep := L.classes[i].representative;
    # get the transversal
    normalizer := Normalizer( L.group, ShallowCopyNoSC(Rep) );
    reps := RightTransversal( L.group, ShallowCopyNoSC(normalizer) );

    # consider all subgroups of N.group
    for k in [1..Length(reps)] do

      M      := Rep^reps[ k ];
      elms_M := List( Elements( M ), e ->
                elms_N[ Position( Elements( N.group ), e ) ] );

      # this is the check for the invariant subnr condition
      if ForAll( elms_N, n -> ForAll( elms_M, m ->
        mul( m, n ) in elms_M and mul( n, m ) in elms_M ) ) then
        Add( inv_sub_nrs, M );
      fi;

    od;

  od;

  if IsTransformationNearring( N ) then
    return List( inv_sub_nrs, i -> Nearring( List( Elements( i ), e ->
      Elements( N )[ Position( Elements( N.group ), e ) ] ) ) );
  else
    return List( inv_sub_nrs, i -> Nearring( i, mul, "n" ) );
  fi;
end;

#############################################################################
##
##  Subnearrings( <N> ) . . . . . . . . . . . . . compute all subnr's of <N>.
##
Subnearrings := function( N )

  local L,          # the lattice of subgroups of the add. group
        Rep,        # representative of the <i>-th class
        normalizer, # normalizer of <I> in <N.group>
        reps,       # transversal of <normalizer> in <N.group>
        M,          # the subgroup being considered
        elms_M,     # the elements of the subgroup M
        sub_nrs,    # the list of subnearrings
```

```
      mul,       # the nearring multiplication
      i,k;       # loop variables

  if not IsNearring( N ) then
    Error( "Usage: Subnearrings( <N> ) where <N> must be a nearring" );
  fi;

  if IsTransformationNearring( N ) then FindGroup( N ); fi;

  mul     := N.multiplication;
  sub_nrs := [];
  L       := Lattice( N.group );

  for i in [1..Length(L.classes)] do

    Rep := L.classes[i].representative;
    # get the transversal
    normalizer := Normalizer( L.group, ShallowCopyNoSC(Rep) );
    reps := RightTransversal( L.group, ShallowCopyNoSC(normalizer) );

    # consider all subgroups of N.group
    for k in [1..Length(reps)] do

      M      := Rep^reps[ k ];
      elms_M := List( Elements( M ), e ->
                Elements( N )[ Position( Elements( N.group ), e ) ] );

      # this is the check for the subnr condition
      if ForAll( elms_M, n -> ForAll( elms_M, m -> mul( m, n ) in elms_M ) )
        then Add( sub_nrs, M );
      fi;

    od;

  od;

  if IsTransformationNearring( N ) then
    return List( sub_nrs, i -> Nearring( List( Elements( i ), e ->
      Elements( N )[ Position( Elements( N.group ), e ) ] ) ) );
  else
    return List( sub_nrs, i -> Nearring( i, mul, "n" ) );
  fi;
end;

#############################################################################
##
#F  LibraryNearring( <name>, <num> ). . . . . . . . get a nr from the library
##
##  This function 'extracts' a nearring from the nearring library files.
##
LibraryNearring := function( name, num )
```

```
local n,     # help var: a nearring
      clmax, # the maximal number of equivalence classes of nearrings
      G,     # the additive group of the nr to be returned
      NR,    # the nearring to be returned
      elms,  # help var: the elements of G
      i,     # help var: a loop variable
      tfle,  # help var: the record that holds the tfl's of the group endos
      f,     # help var: a valid function that represents a class of nr's
      vf,endos,g,a,a_inv,h,compute_all,
      mul;   # local function: the multiplication of the nearring

# check the arguments
if not ( IsString( name ) and IsInt( num ) and num > 0 ) then
  Error( "Usage: LibraryNearring( <name>, <num> ) where <name> must be ",
         "the\n       name of a group and <num> must be a positive ",
         "integer which\ndetermines an isomorphism class" );
fi;

if   ( name = "C2" ) then
  n := NR_C2;
elif ( name = "C3" ) then
  n := NR_C3;
elif ( name = "C4" ) then
  n := NR_C4;
elif ( name = "V4" ) then
  n := NR_V4;
elif ( name = "C5" ) then
  n := NR_C5;
elif ( name = "C6" ) then
  n := NR_C6;
elif ( name = "S3" ) then
  n := NR_S3;
elif ( name = "C7" ) then
  n := NR_C7;
elif ( name = "C8" ) then
  n := NR_C8;
elif ( name = "C2xC4" ) then
  n := NR_C2xC4;
elif ( name = "C2xC2xC2" ) then
  n := NR_C2xC2xC2;
elif ( name = "D8" ) then
  n := NR_D8;
elif ( name = "Q8" ) then
  n := NR_Q8;
elif ( name = "C9" ) then
  n := NR_C9;
elif ( name = "C3xC3" ) then
  n := NR_C3xC3;
elif ( name = "C10" ) then
  n := NR_C10;
elif ( name = "D10" ) then
  n := NR_D10;
```

```
elif ( name = "C11" ) then
  n := NR_C11;
elif ( name = "C12" ) then
  n := NR_C12;
elif ( name = "C2xC6" ) then
  n := NR_C2xC6;
elif ( name = "D12" ) then
  n := NR_D12;
elif ( name = "A4" ) then
  n := NR_A4;
elif ( name = "T" ) then
  n := NR_T;
elif ( name = "C13" ) then
  n := NR_C13;
elif ( name = "C14" ) then
  n := NR_C14;
elif ( name = "D14" ) then
  n := NR_D14;
elif ( name = "C15" ) then
  n := NR_C15;
else
  Print( "There is no group name '", name,
         "' in the nearrings library.\n" );
  return;
fi;

clmax := Length( RecFields( n.classes ) );
if num > clmax then
  Print( "There are only ", clmax, " isomorphism classes of nearrings ",
         "on the group ", name, ".\n" );
  return;
fi;

# put the group of the nearring together and define a few help variables
G      := Group( n.group_generators, () );
G.name := n.group_name;
elms   := Elements( G );
tfle   := n.group_endomorphisms;
f      := n.classes.(num).phi;
G.phi  := f;
G.a_y_i_nrs := n.classes.(num).autos_yielding_iso_nrs;
# retrieve the group endomorphisms from the Nearrings record
if not IsBound( G.endomorphisms ) then
  i := 1; G.endomorphisms := [];         # convert the endomorphism record
  while IsBound( tfle.(i) ) do           # into a list of endomorphisms
    Add( G.endomorphisms, Transformation( G, tfle.(i) ) );
    i := i + 1;
  od;
fi;

compute_all := false;
vf := [ f ];
```

```
  if compute_all then
    endos := []; g:= [];
    for i in [1..Length( RecFields( tfle ) )] do
      Add( endos, tfle.(i) );
    od;
    for a in n.classes.(num).autos_yielding_iso_nrs do
      a := endos[a]; a_inv := [];
      for i in [1..Length( a )] do a_inv[a[i]] := i; od;
      for i in [1..Length( a )] do
        h := a{endos[f[i]]};
        g[ a[i] ] := Position( endos, h{a_inv} );
      od;
      AddSet( vf, Copy( g ) );
    od;
  fi;

  # define a RIGHT distributive multiplication
  mul := function( x, y )
    return elms[ tfle.(f[ Position( elms, y ) ]) [ Position( elms, x ) ] ];
  end;
# define a LEFT distributive multiplication
#  mul := function( x, y )
#    return elms[ tfle.(f[ Position( elms, x ) ]) [ Position( elms, y ) ] ];
#  end;

  # put the nearring together
  NR  := rec();
  # enter category components
  NR.isDomain       := true;
  NR.isNearring     := true;
  NR.isLibraryNearring := true;
  # enter identification components
  NR.group          := G;
  NR.addition       := function( x, y ) return x * y;    end;
  NR.subtraction    := function( x, y ) return x * y^-1; end;
  NR.multiplication := mul;
  # enter operations record
  NR.operations     := NearringOps;

  if compute_all then
    NR.vf_of_iso_nrs := vf;
  fi;

  return NR;
end;

#############################################################################
##
#F  Distributors( <N> ) . . compute the set of distributors on a nearring <N>
##
##  Dispatcher function for computing the distributors of <N>.
##
```

```
Distributors := function( N )
  if IsNearring( N ) then
    if not IsBound( N.distributors ) then
      N.distributors := N.operations.Distributors( N );
    fi;
  else
    Error( "Usage: Distributors( <N> ) where <N> must be a nearring" );
  fi;

  return N.distributors;
end;

#############################################################################
##
#F  DistributiveElements( <N> ). . . compute the distributive elements of <N>
##
##  Dispatcher function for computing the distributive elements of <N>.
##
DistributiveElements := function( N )
  if IsNearring( N ) then
    if not IsBound( N.distributiveElements ) then
      N.distributiveElements := N.operations.DistributiveElements( N );
    fi;
  else
    Error( "Usage: DistributiveElements( <N> ) where <N> must be a nearring");
  fi;

  return N.distributiveElements;
end;

#############################################################################
##
#F  ZeroSymmetricElements( <N> ) . compute the zero-symmetric elements of <N>
##
##  Dispatcher function for computing the zero-symmetric elements of <N>,
##  i.e. all elements n s.t. n0 = 0 (note: 0n = 0 is always true)
##
ZeroSymmetricElements := function( N )
  if IsNearring( N ) then
    if not IsBound( N.zeroSymmetricElements ) then
      N.zeroSymmetricElements := N.operations.ZeroSymmetricElements( N );
    fi;
  else
    Error("Usage: ZeroSymmetricElements( <N> ) where <N> must be a nearring");
  fi;

  return N.zeroSymmetricElements;
end;

#############################################################################
##
#F  IdempotentElements( <D> ). . . . . compute the idempotent elements of <D>
```

```
##
##  Dispatcher function for computing the idempotent elements of <D>.
##
IdempotentElements := function( D )
  if IsNearring( D ) or IsSemigroup( D ) then
    if not IsBound( D.idempotentElements ) then
      D.idempotentElements := D.operations.IdempotentElements( D );
    fi;
  else
    Error( "Usage: IdempotentElements( <D> ) where <D> must be a ",
           "nearring or a semigroup" );
  fi;

  return D.idempotentElements;
end;

#############################################################################
##
#F  NilpotentElements( <N> ). . . . . . . . compute the nilpotent elms of <N>
##
##  Dispatcher function to compute the nilpotent elements of a nearring <N>
##
NilpotentElements := function( N )

  if not IsNearring( N )  then
    Error( "Usage: NilpotentElements( <N> ) where <N> must be a nearring" );
  fi;

  if not IsBound( N.nilpotentElements ) then
    N.nilpotentElements := N.operations.NilpotentElements( N );
  fi;

  return N.nilpotentElements;
end;

#############################################################################
##
#F  QuasiregularElements( <N> ). . . . . compute the quasiregular elms of <N>
##
##  Dispatcher function to compute the quasiregular elements of a nr <N>.
##
QuasiregularElements := function( N )

  if not IsNearring( N )  then
    Error( "Usage: QuasiregularElements( <N> ) where <N> must be a ",
           "nearring" );
  fi;

  if not IsBound( N.quasiregularElements ) then
    N.quasiregularElements := N.operations.QuasiregularElements( N );
  fi;
```

```
  return N.quasiregularElements;
end;

#############################################################################
##
#F  RegularElements( <N> ). . . . . . . . . . compute the regular elms of <N>
##
##  Dispatcher function to compute the regular elements of a nr <N>.
##
RegularElements := function( N )

  if not IsNearring( N )  then
    Error( "Usage: RegularElements( <N> ) where <N> must be a nearring" );
  fi;

  if not IsBound( N.regularElements ) then
    N.regularElements := N.operations.RegularElements( N );
  fi;

  return N.regularElements;
end;

#############################################################################
##
#F  IsAbstractAffineNearring( <N> ) . . . . . . . . . . . test if <N> is a.a.
##
IsAbstractAffineNearring := function( N )
  if IsNearring( N ) then
    if not IsBound( N.isAbstractAffine ) then
      if IsTransformationNearring( N ) then FindGroup( N ); fi;
      N.isAbstractAffine := IsAbelian( N.group ) and
                ( ZeroSymmetricElements( N ) = DistributiveElements( N ) );
    fi;
  else
    Error( "Usage: IsAbstractAffineNearring( <N> ) where <N> must be a ",
           "nearring" );
  fi;

  return N.isAbstractAffine;
end;

#############################################################################
##
#F  IsDistributiveNearring( <N> ) . . . . . . . . test if <N> is distributive
##
IsDistributiveNearring := function( N )
  if IsNearring( N ) then
    if not IsBound( N.isDistributive ) then
      N.isDistributive :=
        Size( DistributiveElements( N ) ) = Size( Elements( N ) );
    fi;
  else
```

```
    Error( "Usage: IsDistributiveNearring( <N> ) where <N> must be a ",
           "nearring" );
  fi;

  return N.isDistributive;
end;

#############################################################################
##
#F  IsBooleanNearring( <N> ) . . . . . . . . . . . . . test if <N> is boolean
##
IsBooleanNearring := function( N )
  if IsNearring( N ) then
    if not IsBound( N.isBoolean ) then
      N.isBoolean :=
        Size( IdempotentElements( N ) ) = Size( Elements( N ) );
    fi;
  else
    Error( "Usage: IsBooleanNearring( <N> ) where <N> must be a nearring" );
  fi;

  return N.isBoolean;
end;

#############################################################################
##
#F  IsDgNearring( <N> ) . . . . . . . test if <N> is distributively generated
##
IsDgNearring := function( N )
  local Nd, elms;

  if IsNearring( N ) then
    if not IsBound( N.isDg ) then
      elms := Elements( N );
      if IsTransformationNearring( N ) then
        FindGroup( N );
        Nd := List( DistributiveElements( N ), de ->
                    Elements( N.group)[ Position( elms, de ) ] );
      else
        Nd := DistributiveElements( N );
      fi;
      N.isDg := Group( Nd, () ) = N.group;
    fi;
  else
    Error( "Usage: IsDgNearring( <N> ) where <N> must be a nearring" );
  fi;

  return N.isDg;
end;

#############################################################################
##
```

```
#F  IsIntegralNearring( <N> ) . . . . . . . . . . . . test if <N> is integral
##
##  A nr is called integral if it has no zero divisors
##
IsIntegralNearring := function( N )
  local mul, elms, zero, non_zero_elms;

  if IsNearring( N ) then
    if not IsBound( N.isIntegral ) then
      mul  := N.multiplication;
      elms := Elements( N );
      zero := elms[ 1 ];  ## the first element is always the zero!
      non_zero_elms := Copy( elms ); RemoveSet( non_zero_elms, zero );
      N.isIntegral := ForAll( non_zero_elms, x ->
                      ForAll( non_zero_elms, y -> mul( x, y ) <> zero ) );
    fi;
  else
    Error( "Usage: IsIntegralNearring( <N> ) where <N> must be a nearring" );
  fi;

  return N.isIntegral;
end;

#############################################################################
##
#F  IsNilNearring( <N> ) . . . . . . . . . . . . . . . . . . test if <N> is nil
##
IsNilNearring := function( N )
  if IsNearring( N ) then
    if not IsBound( N.isNil ) then
      N.isNil :=
        Length( NilpotentElements( N ) ) = Size( Elements( N ) );
    fi;
  else
    Error( "Usage: IsNilNearring( <N> ) where <N> must be a nearring" );
  fi;

  return N.isNil;
end;

#############################################################################
##
#F  IsNilpotentNearring( <N> ) . . . . . . . . . . . test if <N> is nilpotent
##
IsNilpotentNearring := function( N )
  local mul, elms, prod, previous_prod, m, n;

  if IsNearring( N ) then
    if not IsBound( N.isNilpotent ) then
      if ( Size( IdempotentElements( N ) ) > 1 ) or
         ( not IsNilNearring( N ) ) then
          N.isNilpotent := false;
```

```
      else

        mul := N.multiplication;
        elms := Elements( N );
        previous_prod := Copy( elms );

        repeat

          prod := [];
          for m in previous_prod do
            for n in elms do
              AddSet( prod, mul( m, n ) );
            od;
          od;
          if prod = previous_prod then
            N.isNilpotent := false;
          elif Size( prod ) = 1 then
            N.isNilpotent := true;
          else
            previous_prod := Copy( prod );
          fi;
        until IsBound( N.isNilpotent );

      fi;
    fi;
  else
    Error( "Usage: IsNilpotentNearring( <N> ) where <N> must be a nearring" );
  fi;

  return N.isNilpotent;
end;

#############################################################################
##
#F  IsPrimeNearring( <N> ) . . . . . . . . . . . . . . . test if <N> is prime
##
##  A nearring N is called prime if { 0 } is a prime ideal
##
IsPrimeNearring := function( N )
  local mul, ideals, elms, elms_G, zero;

  if IsNearring( N ) then
    if not IsBound( N.isPrime ) then

      if IsIntegralNearring( N ) then
        N.isPrime := true;
      else
        ideals := Copy( NearringIdeals( N ) ); Unbind( ideals[ 1 ] );
        mul    := N.multiplication;
        if IsTransformationNearring( N ) then
          FindGroup( N );
          elms   := Elements( N );
```

```
          elms_G := Elements( N.group );
          zero   := elms[ 1 ];
          N.isPrime := ForAll( ideals, I -> ForAll( ideals, J ->
                     ForAny( Elements( I ), i -> ForAny( Elements( J ), j ->
                     mul( elms[ Position( elms_G, i ) ],
                          elms[ Position( elms_G, j ) ] ) <> zero  ) ) ) ) );
        else
          N.isPrime := ForAll( ideals, I -> ForAll( ideals, J ->
                       ForAny( Elements( I ), i -> ForAny( Elements( J ), j ->
                       mul( i, j ) <> ()  ) ) ) ) );
        fi;
      fi;

    fi;

  else
    Error( "Usage: IsPrimeNearring( <N> ) where <N> must be a nearring" );
  fi;

  return N.isPrime;
end;

#############################################################################
##
#F  IsQuasiregularNearring( <N> ) . . . . . . . . . . . . . test if <N> is qr
##
IsQuasiregularNearring := function( N )
  if IsNearring( N ) then
    if not IsBound( N.isQuasiregular ) then
      N.isQuasiregular :=
        Size( QuasiregularElements( N ) ) = Size( Elements( N ) );
    fi;
  else
    Error( "Usage: IsQuasiregularNearring( <N> ) where <N> must be a ",
           "nearring" );
  fi;

  return N.isQuasiregular;
end;

#############################################################################
##
#F  IsRegularNearring( <N> ) . . . . . . . . . . . . . test if <N> is regular
##
IsRegularNearring := function( N )
  if IsNearring( N ) then
    if not IsBound( N.isRegular ) then
      N.isRegular :=
        Size( RegularElements( N ) ) = Size( Elements( N ) );
    fi;
  else
    Error( "Usage: IsRegularNearring( <N> ) where <N> must be a nearring" );
```

```
  fi;

  return N.isRegular;
end;

#############################################################################
##
#F  IsNilpotentFreeNearring( <N> ) . . test if <N> is w/o non-zero nilpotents
##
IsNilpotentFreeNearring := function( N )
  if IsNearring( N ) then
    if not IsBound( N.isNilpotentFree ) then
      N.isNilpotentFree := Length( NilpotentElements( N ) ) = 1;
    fi;
  else
    Error( "Usage: IsNilpotentFreeNearring( <N> ) where <N> must be a ",
           "nearring" );
  fi;

  return N.isNilpotentFree;
end;

#############################################################################
##
#F  IsPlanarNearring( <N> ) . . . . . . . . . . . . . . test if <N> is planar
##
IsPlanarNearring := function( N )
  local phi, size, endos;

  if IsBound( N.isLibraryNearring ) then
    if not IsBound( N.isPlanar ) then
      phi   := Set( N.group.phi );
      size  := Size( N.group );
      endos := Endomorphisms( N.group );
      N.isPlanar := Size( phi ) >= 3 and
        ForAll( phi, p -> p = 1 or p = Length( endos ) or
          ( Size( Set( endos[p].tfl ) ) = size and
            ForAll( [2..size], i -> endos[p].tfl[i] <> i ) ) );
    fi;
  else
    Error( "Usage: IsPlanarNearring( <N> ) where <N> must be a ",
           "library nearring" );
  fi;

  return N.isPlanar;
end;

#############################################################################
##
#F  IsNearfield( <N> ) . . . . . . . . . . . . . . test if <N> is a nearfield
##
IsNearfield := function( N )
```

```
  local mul, id, elms;

  if IsNearring( N ) then
    if not IsBound( N.isNearfield ) then
      mul  := N.multiplication;
      elms := Elements( N );
      id   := Identity( N );
      N.isNearfield := id <> [ ] and
        Size(
              Filtered( elms, e -> ForAny( elms, x -> mul( e, x ) = id[1] ) )
            ) = Size( elms ) - 1;
    fi;
  else
    Error( "Usage: IsNearfield( <N> ) where <N> must be a nearring" );
  fi;

  return N.isNearfield;
end;

#############################################################################
##
#F  NearringOps.Distributors( <N> ). . compute distributors of a nearring <N>
##
NearringOps.Distributors := function( N )
  local elms, a, b, c, add, sub, mul, dbs;

  add := N.addition;
  sub := N.subtraction;
  mul := N.multiplication;
  dbs := [];
  elms := Elements( N );
  for a in elms do
    for b in elms do
      for c in elms do
        AddSet( dbs, sub( mul( a, add( b, c ) ),
                         add( mul( a, b ), mul( a, c ) ) ) );
      od;
    od;
  od;

  return dbs;
end;

#############################################################################
##
#F  NearringOps.DistributiveElements( <N> ). . . . . distributive elms of <N>
##
##  Note: this function works only for RIGHT nearrings, i.e. it computes
##        the LEFT distributive elements.
##
NearringOps.DistributiveElements := function( N )
  local add, mul, elms;
```

```
  add  := N.addition;
  mul  := N.multiplication;
  elms := Elements( N );

  return Filtered( elms, d -> ForAll( elms, a -> ForAll( elms, b ->
            mul( d, add( a, b ) ) = add( mul( d, a ), mul( d, b ) ) ) ) );

end;

#############################################################################
##
#F  NearringOps.ZeroSymmetricElements( <N> ) . . . zero-symmetric elms of <N>
##
##  Note: this function works only for RIGHT nearrings, i.e. it computes
##        all elements n s.t n0 = 0. (Note that in a RIGHT nearring
##        0n = 0 is always true).
##
NearringOps.ZeroSymmetricElements := function( N )
  local mul, elms, zero;

  mul  := N.multiplication;
  elms := Elements( N );
  zero := elms[ 1 ];  # the first element is the zero element!

  return Filtered( elms, n -> mul( n, zero ) = zero );

end;

#############################################################################
##
#F  NearringOps.IdempotentElements( <N> ). . . . . . . idempotent elms of <N>
##
NearringOps.IdempotentElements := function( N )
  local mul;

  mul  := N.multiplication;

  return Filtered( Elements( N ), i -> mul( i, i ) = i );

end;

#############################################################################
##
#F  NearringOps.NilpotentElements( <N> ). . compute nilpotent elements of <N>
##
NearringOps.NilpotentElements := function( N )
  local mul, elms, elm, e, size, zero, npelms, k, old_e;

  mul    := N.multiplication;
  elms   := Elements( N );
  size   := Size( elms );
```

```
  zero   := elms[ 1 ];   ## the first element is the zero!
  npelms := [ ];

  for elm in elms do
    k := 1; e := Copy( elm );
    old_e := zero;
    while e <> zero and e <> old_e and k < size do
      k := k + 1;
      old_e := Copy( e );
      e := mul( e, elm );
    od;
    if e = zero then Add( npelms, [ elm, k ] ); fi;
  od;

  return npelms;
end;

#############################################################################
##
#F  NearringOps.QuasiregularElements( <N> ). . . . compute qr elements of <N>
##
NearringOps.QuasiregularElements := function( N )
  local mul, sub, elms, z, elms_to_test, qr_elms, A, li, Lz;

  mul     := N.multiplication;
  sub     := N.subtraction;
  elms    := Elements( N );
  qr_elms := List( NilpotentElements( N ), n -> n[1] );
  elms_to_test := Copy( elms );
  SubtractSet( elms_to_test, IdempotentElements( N ) );
  SubtractSet( elms_to_test, qr_elms );

  for z in elms_to_test do
    A := Set( List( elms, n -> sub( n, mul( n, z ) ) ) );
    if IsTransformationNearring( N ) then
      FindGroup( N );
      li := List( NearringIdeals( N, "l" ), i ->
        List( Elements(i), e -> elms[ Position( Elements(N.group), e ) ] ) );
    else
      li := List( NearringIdeals( N, "l" ), i -> Elements( i ) );
    fi;

    Lz := First( li, i -> IsSubset( i, A ) );
    if z in Lz then AddSet( qr_elms, z ); fi;
  od;

  return qr_elms;
end;

#############################################################################
##
#F  NearringOps.RegularElements( <N> ). . . . . . . . . . regular elms of <N>
```

```
##
NearringOps.RegularElements := function( N )
  local mul, elms;

  mul  := N.multiplication;
  elms := Elements( N );

  return
    Filtered( elms, x -> ForAny( elms, y -> mul( x, mul( y, x ) ) = x ) );

end;

#############################################################################
##
#F  LibraryNearringInfo( <name>, <list> ). . . . info about library nearrings
##
LibraryNearringInfo := function( arg )
  local N, elms, n, symbols, help, i, k, name, list, string, letters;

  if not ( Length( arg ) in [ 2, 3 ] and IsString( arg[1] ) and
           IsList( arg[2] ) and ForAll( arg[2], l -> IsInt( l ) ) ) then
    Error( "Usage: LibraryNearringInfo( <name>, <list> ) where <name>",
           " must be a\ngroup name and <list> must be a list of numbers",
           "of classes" );
  fi;

  name := arg[1]; list := arg[2];
  if IsBound( arg[3] ) then letters := arg[3]; else letters := ""; fi;
  if 'C' in letters or 'c' in letters then
    Print( "A ... abstract affine\n" );
    Print( "B ... boolean\n" );
    Print( "C ... commutative\n" );
    Print( "D ... distributive\n" );
    Print( "F ... nearfield\n" );
    Print( "G ... distributively generated\n" );
    Print( "I ... integral\n" );
    Print( "N ... nilpotent\n" );
    Print( "O ... planar\n" );
    Print( "P ... prime\n" );
    Print( "Q ... quasiregular\n" );
    Print( "R ... regular\n" );
    Print( "W ... without non-zero nilpotent elements\n" );
    Print( "---------------------------------------",
           "---------------------------------------\n" );
  fi;

  N := LibraryNearring( name, list[ 1 ] );
  elms    := Elements( N );
  n       := Size( elms );
  symbols := [ AbstractGenerator( "n0" ) ];
  if n > 1 then
    symbols := Concatenation( [ AbstractGenerator( "n0" ) ],
```

```
                              AbstractGenerators( "n" , n-1 ) );
fi;
Print( "--------------------------------------",
       "--------------------------------------" );
Print( "\n>>> GROUP: ", N.group.name, "\nelements: " );
Print( symbols, "\n" );

Print( "addition table:\n" );
NearringOps.PrintTable( N, "a" );

Print( "group endomorphisms:\n" );
for i in [1..Length(Endomorphisms( N.group ))] do
  if i < 10 then
    Print( i, ":   " );
  else
    Print( i, ":  " );
  fi;
  Print( List( Endomorphisms(N.group)[i].tfl, e -> symbols[ e ] ), "\n" );
od;

Print( "\nNEARRINGS:\n" );
Print( "--------------------------------------",
       "--------------------------------------" );

for k in list do
  N := LibraryNearring( name, k );
  Print( "\n", k, ")  phi: ", N.group.phi, ";  " );
  for i in N.group.a_y_i_nrs do Print( i, ";" ); od;
  string := [];
  if IsAbstractAffineNearring( N ) then Add( string, 'A' );
  else Add( string, '-' ); fi;
  if IsBooleanNearring( N ) then Add( string, 'B' );
  else Add( string, '-' ); fi;
  if IsCommutative( N ) then Add( string, 'C' );
  else Add( string, '-' ); fi;
  if IsDistributiveNearring( N ) then Add( string, 'D' );
  else Add( string, '-' ); fi;
  if IsNearfield( N ) then Add( string, 'F' );
  else Add( string, '-' ); fi;
  if IsDgNearring( N ) then Add( string, 'G' );
  else Add( string, '-' ); fi;
  if IsIntegralNearring( N ) then Add( string, 'I' );
  else Add( string, '-' ); fi;
  if IsNilpotentNearring( N ) then Add( string, 'N' );
  else Add( string, '-' ); fi;
  if IsPlanarNearring( N ) then Add( string, 'O' );
  else Add( string, '-' ); fi;
  if IsPrimeNearring( N ) then Add( string, 'P' );
  else Add( string, '-' ); fi;
  if IsQuasiregularNearring( N ) then Add( string, 'Q' );
  else Add( string, '-' ); fi;
  if IsRegularNearring( N ) then Add( string, 'R' );
```

```
else Add( string, '-' ); fi;
if IsNilpotentFreeNearring( N ) then Add( string, 'W' );
else Add( string, '-' ); fi;
Print( "  ", string );
if Identity( N ) <> [] then Print( ";  I = ",
  symbols[ Position( Elements( N.group ), Identity(N)[1] ) ], "\n" );
else
  Print( "\n" );
fi;

if 'M' in letters or 'm' in letters then
  Print( "multiplication table:" );
  NearringOps.PrintTable( N, "m" );
fi;
if 'I' in letters or 'i' in letters then
  Print( "ideals:\n" );
  help := NearringIdeals( N );
  for i in [1..Length(help)] do
    Print( i, ".  ", List( Elements(help[i]),
           elm -> symbols[Position(elms,elm)] ), "\n" );
  od;
fi;
if 'L' in letters or 'l' in letters then
  Print( "left ideals:\n" );
  help := NearringIdeals( N, "l" );
  for i in [1..Length(help)] do
    Print( i, ".  ", List( Elements(help[i]),
           elm -> symbols[Position(elms,elm)] ), "\n" );
  od;
fi;
if 'R' in letters or 'r' in letters then
  Print( "right ideals:\n" );
  help := NearringIdeals( N, "r" );
  for i in [1..Length(help)] do
    Print( i, ".  ", List( Elements(help[i]),
           elm -> symbols[Position(elms,elm)] ), "\n" );
  od;
fi;
if 'V' in letters or 'v' in letters then
  Print( "invariant subnearrings:\n" );
  help := InvariantSubnearrings( N );
  for i in [1..Length(help)] do
    Print( i, ".  ", List( Elements(help[i]),
           elm -> symbols[Position(elms,elm)] ), "\n" );
  od;
fi;
if 'S' in letters or 's' in letters then
  Print( "subnearrings:\n" );
  help := Subnearrings( N );
  for i in [1..Length(help)] do
    Print( i, ".  ", List( Elements(help[i]),
           elm -> symbols[Position(elms,elm)] ), "\n" );
```

```
      od;
    fi;
    if 'E' in letters or 'e' in letters then
      Print( "nearring endomorphisms: " );
      help := Endomorphisms( N );
      for i in List( help, e -> Position( Endomorphisms(N.group), e ) ) do
        Print( i, "; " );
      od; Print( "\n" );
    fi;
    if 'A' in letters or 'a' in letters then
      Print( "nearring automorphisms: " );
      help := Automorphisms( N );
      for i in List( help, e -> Position( Endomorphisms(N.group), e ) ) do
        Print( i, "; " );
      od; Print( "\n" );
    fi;
    Print( "---------------------------------------",
           "---------------------------------------" );

  od;
  Print( "\n" );
  return;
end;
```

B.6 The Source File g.g

```
#############################################################################
##  File: G.G                                                              ##
#############################################################################
##  Add some support functions for Groups.
#############################################################################
##
#############################################################################
##
#F  InnerAutomorphisms( <G> ) . . . . . create all innerauto's on a group <G>
##  V1.0  24.2.95
##  Dispatcher function for computing all inner automorphisms on <G>.
##
InnerAutomorphisms := function( G )

  if IsGroup( G ) then
    if not IsBound( G.innerAutomorphisms ) then
      G.innerAutomorphisms := G.operations.InnerAutomorphisms( G );
    fi;
  else
    Error( "Usage: InnerAutomorphisms( <G> ) where <G> must be a group" );
  fi;

  return G.innerAutomorphisms;
end;
```

```
################################################################################
##
##  GroupOps.Endomorphisms( <G> ). . . . compute the endomorphisms of a group
##
GroupOps.Endomorphisms := function( G )

  local elms,        # the elements of G
        gens,        # the generators of G
        m,           # the number of generators of G
        n,           # the number of elements of G
        k,           # a loop variable
        TFL,         # a list of transformation lists ( tfl's )
        im,          # the image of an endomorphism candidate
        orders_gens, # a list of the orders of the generators of G
        orders_elms, # a list of the orders of the elements of G
        tfl,         # a transformation list
        E,           # the list of endomorphisms on G
        h,           # an endomorphism candidate
        done;        # a flag variable

  elms        := Elements( G );
  gens        := G.generators;
  m           := Length( gens );
  n           := Size( elms );
  k           := -1;
  TFL         := [];
  im          := List( [1..m], j -> 1 );
  orders_gens := List( gens, gen -> Order( G, gen ) );
  orders_elms := List( elms, elm -> Order( G, elm ) );

  # consider all functions: f: gens -> elms
  #                         s.t. Order( f(gen) ) divides Order( gen )
  # ( f represented as Length(gens)-tuples "im" of elements in elms )
  while k <= m do

    if ForAll( [1..m], j -> RemInt( orders_gens[j], orders_elms[im[j]]) = 0 )
    then

      h := GroupHomomorphismByImages( G, G, gens, List( im, j -> elms[j] ) );
      if IsGroupHomomorphism( h ) then
      # NOTE: this additional "if" may indeed seem a little awkward, but
      # "IsGroupHomomorphism( h )" alone won't work in GAP 3.2.
        if MappingOps.IsGroupHomomorphism( h ) then
          AddSet( TFL,
                  List( elms, elm -> Position( elms, Image( h, elm ) ) ) );
        fi;
      fi;
    fi;
    k := 1; done := false;
    while not done and k <= m do
      if im[k] = n then im[k] := 1; k := k+1;
```

```
      else im[k] := im[k]+1; done := true;
      fi;
    od;
  od;

  # Put I in the last position
  TFL := Filtered( TFL, e -> e <> [1..n] );
  Add( TFL, List( [1..n], j -> j ) );

  # return the result as group transformations
  E := [];
  for tfl in TFL do
    h := Transformation( G, tfl );
    h.isGroupEndomorphism := true;
    if Size( Set( tfl ) ) = n then h.isGroupAutomorphism := true; fi;
    Add( E, h );
  od;

  return E;
end;

#############################################################################
##
##  GroupOps.Automorphisms( <G> ). . . . compute the automorphisms of a group
##
GroupOps.Automorphisms := function( G )

  local elms,        # the elements of G
        gens,        # the generators of G
        m,           # the number of generators of G
        n,           # the number of elements of G
        k,           # a loop variable
        TFL,         # a list of transformation lists ( tfl's )
        im,          # the image of an endomorphism candidate
        orders_gens, # a list of the orders of the generators of G
        orders_elms, # a list of the orders of the elements of G
        tfl,         # a transformation list
        E,           # the list of endomorphisms on G
        h,           # an endomorphism candidate
        done;        # a flag variable

  elms        := Elements( G );
  gens        := G.generators;
  m           := Length( gens );
  n           := Size( elms );
  k           := -1;
  TFL         := [];
  im          := List( [1..m], j -> 2 );
  orders_gens := List( gens, gen -> Order( G, gen ) );
  orders_elms := List( elms, elm -> Order( G, elm ) );
  # consider all functions: f: gens -> elms
  #                         s.t. Order( f(gen) ) divides Order( gen )
```

```
  # ( f represented as Length(gens)-tuples "im" of elements in elms )
  while k <= m do
    if ( Size( Set( im ) ) = m and
         ForAll( [1..m], j -> orders_gens[j] = orders_elms[im[j]] ) ) then
      h := GroupHomomorphismByImages( G, G, gens, List( im, j -> elms[j] ) );
      if IsBijection( h ) then
      # NOTE: this additional "if" may indeed seem a little awkward, but
      # "IsGroupHomomorphism( h )" alone won't work in GAP 3.2.
        if MappingOps.IsGroupHomomorphism( h ) then
          AddSet( TFL,
                  List( elms, elm -> Position( elms, Image( h, elm ) ) ) );
        fi;
      fi;
    fi;
    k := 1; done := false;
    while not done and k <= m do
      if im[k] = n then im[k] := 2; k := k+1;
      else im[k] := im[k]+1; done := true;
      fi;
    od;
  od;

  # Put I in the last position
  TFL := Filtered( TFL, e -> e <> [1..n] );
  Add( TFL, List( [1..n], j -> j ) );

  # return the result as group transformations
  E := [];
  for tfl in TFL do
    h := Transformation( G, tfl );
    h.isGroupEndomorphism := true;
    h.isGroupAutomorphism := true;
    Add( E, h );
  od;

  return E;
end;

#############################################################################
##
##  GroupOps.InnerAutomorphisms( <G> ). . . . compute inner auto's of a group
##
GroupOps.InnerAutomorphisms := function( G )
  local I,E,g,id,i;

  I := []; E := [];
  for g in Elements( G ) do
    AddSet( I, InnerAutomorphism( G, g ) );
  od;

  for i in I do
    i := AsTransformation( i );
```

```
    i.isGroupEndomorphism := true;
    i.isGroupAutomorphism := true;
    i.isInnerAutomorphism := true;
    if i <> IdentityTransformation( G ) then
      AddSet( E, i );
    else
      id := Copy( i );
    fi;
  od;
  Add( E, id );
  return E;
end;

#############################################################################
##
##  GroupOps.Table( <G> ) . . . . . . . . print a Cayley table of a group <G>
##
GroupOps.Table := function( G )

  local elms,    # the elements of the group
        n,       # the size of the group
        symbols, # a list of the symbols for the elements of the group
        tw,      # the width of a table
        spc,     # local function which prints the right number of spaces
        bar,     # local function for printing the right length of the bar
        ind,     # help variable, an index
        i,j;     # loop variables

  elms    := Elements( G );
  n       := Length( elms );
  symbols := [ AbstractGenerator( "g0" ) ];
  if n > 1 then
    symbols := Concatenation( [ AbstractGenerator( "g0" ) ],
                              AbstractGenerators( "g" , n-1 ) );
  fi;
  # compute the number of characters per line required for the table
  if n < 11 then tw := 3*n + 6; else tw := 4*n + 8; fi;
  spc     := function( i, max )
               if max < 11 then return " "; fi;
               if i < 11 then return "  "; else return " "; fi;
             end;
  bar     := function( max )
               if max < 11 then return "---"; else return "----"; fi;
             end;

  if SizeScreen()[1] - 3 < tw then
    Print( "The table of a group of order ", n, " will not ",
           "look\ngood on a screen with ", SizeScreen()[1], " characters per ",
           "line.\nHowever, you may want to set your line length to a ",
           "greater\nvalue by using the GAP function 'SizeScreen'.\n" );
    return;
  fi;
```

```
  Print( "Let:\n" );
  for i in [1..n] do Print( symbols[i], " := ", elms[i], "\n" ); od;

  # print the addition table
  Print( "\n  + ", spc( 0, n ), "| " );
    for i in [1..n] do Print( symbols[i], spc( i, n ) ); od;
  Print( "\n ---", bar( n ) ); for i in [1..n] do Print( bar( n ) ); od;
  for i in [1..n] do
    Print( "\n  ", symbols[i], spc( i, n ), "| " );
    for j in [1..n] do
      ind := Position( elms, elms[i] * elms[j] );
      Print( symbols[ ind ], spc( ind, n ) );
    od;
  od;

  Print( "\n\n" );
end;

#############################################################################
##
#F  SmallestGeneratingSystem(<G>) . . . smallest generating system of a group
##
SmallestGeneratingSystem := function ( G )
  local   gens,       # smallest generating system of <G>, result
          gen,        # one generator of <gens>
          elms,       # the list of elements sorted by decreasing order
          H;          # subgroup generated by <gens> so far

  if not IsGroup( G ) then
    Error( "Usage: SmallestGeneratingSystem( <G> ) where <G> must be a ",
           "group" );
  fi;
  # start with the empty generating system and the trivial subgroup
  gens := [];
  H := TrivialSubgroup( G );
  elms := Copy( Elements( G ) );
  if Size( elms ) = 1 then return elms; fi;
  Sort( elms, function( x, y ) return Order( G, x ) > Order( G, y ); end );

  # loop over the elements of <G> in their decreasing order
  for gen  in elms do

    # add the element not lying in the subgroup generated by the previous
    if not gen in H  then
      Add( gens, gen );
      H := Closure( H, gen );

      # it is important to know when to stop
      if Size( H ) = Size( G )  then
        return gens;
      fi;
```

```
    fi;

  od;

  # well we should never come here
  Error("panic, <G> not generated by its elements");
end;

#############################################################################
##
#F  IsIsomorphicGroup( <G>, <H> ) . . Check if two groups G, H are isomorphic
##  V0.2 3.10.94
##  this version works even with the error in GAP 3.2.
##  The return value is 'false' if G and H are not isomorphic and an
##  isomorphism between G and H if they are isomorphic.
##
IsIsomorphicGroup := function( G, H )

  local imageset, # all the poss. im's of the f's G.generators -> H.elements
        image,    # one fixed image in imageset
        h;        # help variable: an isomorphism candidate

  if not ( IsGroup( G ) and IsGroup( H ) ) then
    Error( "Usage: IsIsomorphicGroup( <G>, <H> ) where <G> and <H> both ",
           "must be\ngroups" );
  fi;
  if Size( G ) <> Size( H ) then return false; fi;

  # build all functions G.generators -> H.elements and consider them
  imageset := Tuples( Elements( H ), Length( G.generators ) );

  for image in imageset do
    h := GroupHomomorphismByImages( G, H, G.generators, image );
    if IsGroupHomomorphism( h ) then
    # NOTE: this additional "if" may indeed seem a little awkward, but
    # "IsGroupHomomorphism( h )" alone won't work in GAP 3.2.
      if MappingOps.IsGroupHomomorphism( h ) then
        if IsBijection( h ) then
          return h;
        fi;
      fi;
    fi;
  od;
  return false;
end;

#############################################################################
##  This function here is necessary to be compatible with GAP versions
##  lower than 3.4.
##
ShallowCopyNoSC := function ( G )
    local   S;
```

```
    S := ShallowCopy( G );
    Unbind( S.orbit );
    Unbind( S.transversal );
    Unbind( S.stabilizer );
    Unbind( S.stabChain );
    return S;
end;
```

B.7 The Source File cnr.g

```
#############################################################################
##  File: CNR.G                                                            ##
#############################################################################
##
#############################################################################
##  This source file contains the stuff to compute and classify nearrings.
#############################################################################
##
#############################################################################
##  The first function is an implementation of Clay's method to generate
##  nearrings as Yearby describes it on pp. 7 - 10 of his dissertation.
#############################################################################
#############################################################################
##
#F  ValidFunctionsNr( <G> ). . Determine all functions f: G -> E ( = End(G) )
##                            with f(f(a)(b)) = f(a) o f(b)
##  V1.0 23.2.95  V0.9 6.8.94
##  input parameter: G........a group
##  return value:    valid_f..a list of all functions with the above property
##
##  the format of valid_f is a list of lists s.t. each of those lists
##  represents a function f: G -> E
##
ValidFunctionsNr := function( G )

  local TFL,     # the list of the tf lists of the tf's of the group
        m,       # help var: the number of transformed elements
        n,       # help var: total number of endomorphisms
        i,       # help var: a list of indices
        tuples,  # help var: all tuples out of [1..m]
        valid_f, # a list of the functions with the above property
        idpot,   # number of idempotent endomorphisms
        total,   # total number of functions to be considered
        k,       # a loop variable
        count,   # the number of the function currently being considered
        done,    # help var: indicates when to stop the loop
        f;       # help var: a function to be considered

  Print( "computing endomorphisms...\n" );
  TFL     := List( Endomorphisms( G ), e -> e.tfl );
```

```
  m       := Length( TFL[1] );
  n       := Length( TFL );
  i       := List( [1..m], k -> 1 );
  tuples  := Tuples( [1..m], 2 );
  valid_f := [];
  idpot   := 0;
  # determine the number of idempotent endomorphisms in TFL.
  for k in TFL do if k{ k } = k then idpot := idpot + 1; fi; od;
  # compute the number of loop executions
  total     := (idpot-1) * n^(m-1);
  Print( "total number of endomorphisms on ", G, ": ", n, "\n" );
  Print( "total number of idempotent endomorphisms on ", G, ": ",
         idpot, "\n" );
  Print( "computing valid functions...\n" );

  k     := -1;
  count :=  0;
  while k <= m and i[m] < n do
    count := count + 1;
    Print( "  considering function ", count, " of ", total, "...\r" );
    f := Sublist( TFL, Reversed( i ) );
    if ForAll( tuples, t -> f[ f[t[1]][t[2]] ] = f[t[1]] { f[t[2]] } ) then
      Add( valid_f, Reversed( i ) );
    fi;
    k := 1;
    done := false;
    while not done and k <= m do
      if i[k] = n then i[k] := 1; k := k+1;
      else
        if k = m then
          repeat
            i[k] := i[k]+1;
          # until the next idempotent or I is found
          until TFL[i[k]]{ TFL[i[k]] } = TFL[i[k]] or TFL[i[k]] = [1..m];
          done := true;
        else
          i[k] := i[k]+1;
          done := true;
        fi;
      fi;
    od;
  od;

  Print( "\n" );
  # Add [I,...,I] to the list of valid functions
  Add( valid_f, List( [1..m], k -> Position( TFL, [1..m] ) ) );
  return valid_f;

end;

##############################################################################
##
```

```
#F  ClassifyNr( <valid_f>, <E> ) . . . . . . . . determine iso classes of nrs
##  V1.0 23.2.95  V0.15 7.10.94
##  input parameters: valid_f...a list of lists representing
##                              valid functions G -> E
##                    E.........a list of endomorphisms on G
##  return value:     classes...a record of iso classes
##
ClassifyNr := function( valid_f, E )

  local TFL,     # the list of the tf lists of the tf's of the group
        R,       # a list, the entries stand for the elements of the group
        A,       # a list of the tfl's that represent automorphisms
        vf,      # help var: the set of lists representing valid functions,
                 # will be dynamically reduced in each loop execution
        noc,     # contains the number of the current class
        classes, # return value: record which contains the computed classes
        vfcount, # help var: counts how many functions remain
        f,g,     # lists, representing functions in vf
        a,       # a tfl, representing an automorphism in A
        loa;     # a list of automorphisms ( represented as lists )

  TFL     := List( E, e -> e.tfl );
  R       := [1..Length( TFL[1] )];
  Print( "computing automorphisms...\n" );
  A       := Filtered( TFL, e -> Size( Set( e ) ) = Size( R ) );
             Add( A, [] );
  vf      := Set( valid_f );
  noc     := 0;
  classes := rec();
  vfcount := Length( vf );

  Print( "classifying...\n" );
  for f in vf do
    noc := noc + 1;
    classes.(noc) := rec( phi := f );
    # initialize loa with the identity
    loa := [ Length( E ) ]; # ( note that I must be the last entry in E )
    Print( "functions to go: ", vfcount, "   \r" );
    Unbind( vf[ Position( vf, f ) ] ); vfcount := vfcount - 1;
    for g in vf do
      a := First( A, a -> a = []
              # this is isomorphisms the right way
              or ForAll( R, x -> a{ TFL[ f[x] ] } = TFL[ g[a[x]] ]{ a } )
#              # this is anti-isomorphisms
#              or ForAll( R, x -> ForAll( R, y ->
#                                   a[TFL[f[y]][x]] = TFL[g[a[x]]][a[y]] ))
               );
      if a <> [] then
        AddSet( loa, Position( TFL, a ) );
        Unbind( vf[Position( vf, g )] ); vfcount := vfcount - 1;
      fi;
    od;
```

```
    classes.(noc).autos_yielding_iso_nrs := loa;
  od;

  Print( "\n" );
  return classes;
end;

#############################################################################
##
#F  ConstructNr( <G>, <classes> ) . . . . . . . . . construct nearring record
##  V1.0 23.2.95
##  input parameters: G.........a group
##                    classes...a record of nr classes on G as
##                              constructed by ClassifyNr
##  return value:     rec.......a nearring record with all neccessary info
##
ConstructNr := function( G, classes )

  local E, elms, endos, i, NR;

  E := Endomorphisms( G );
  elms  := rec();
  endos := rec();
  for i in [1..Size( G )] do elms.(i) := Elements( G )[i]; od;
  for i in [1..Length( E )] do endos.(i) := E[i].tfl; od;

  for i in [1..Length( RecFields( classes ) )] do
    Sort( classes.(i).autos_yielding_iso_nrs );
  od;

  NR := rec(
    group_name          := G.name,
    group_generators    := G.generators,
    elements            := elms,
    group_endomorphisms := endos,
    classes             := classes
  );

  return NR;
end;

#############################################################################
##  The two following functions are an implementation of Yearby's method
##  to construct nearrings as he describes it on pp 74 - 91 of his
##  dissertation.
#############################################################################
#############################################################################
##
##  Extend( <S>, <f> ) . . . . . . . . . . . . extend partial multiplications
##
##  input parameters: S: a list of the numbers of elements that are mapped
##                    f: a list of tfl's ( maybe with holes )
```

```
##  Example: S = [ 3, 5 ], f = [ ,,[...],,[...] ]
##
Extend := function( S, f )

  local f1, L1, L2, S2, ST, s, t, c, f2, C, pairs;

  f1 := Copy( f ); L1 := Copy( S ); C := [];
  while true do
    # generate L2 and S2
    L2 := Copy( L1 ); S2 := []; ST := [];
    for s in L1 do
      for t in L1 do
        c := f1[s][t];
        AddSet( L2, c );
        # construct the set C for the compatibility check
        if c in L1 then
          if [ s, t ] in C then
            RemoveSet( C, [s,t] );
          else
            AddSet( C, [s,t] );
          fi;
        fi;
        # store the pair [ s, t ] for later use
        if not ( c in L1 ) and not ( c in S2 ) then
          AddSet( S2, c );
          ST[c] := [ s, t ];
        fi;
      od;
    od;
    # perform compatibility check
    if not ForAll( C, pairs -> f1[ f1[pairs[1]][pairs[2]] ] =
                               f1[pairs[1]]{ f1[pairs[2]] } ) then
      return [ false, [] ];
    fi;
    if S2 = [] then return [ f1, L1 ]; fi;
    # compute f2
    f2 := [];
    for c in L2 do
      if c in L1 then
        f2[c] := f1[c];
      else
        s := ST[c][1]; t := ST[c][2];
        f2[c] := f1[s]{ f1[t] };
      fi;
    od;
    # reassign f1 and L1 and restart the loop
    f1 := f2; L1 := L2;
  od;
  return;
end;

##############################################################################
```

```
##
#F  ValidFunctionsYearby( <E> ). . . . . . Determine all functions f: S -> E
##                                         with f(f(a)(b)) = f(a) o f(b)
##  V1.0 28.2.95  V0.9 9.11.94
##  input parameter: E....a set s.t. ( E, gamma ) is a Yearby pair w.r.t. S
##  return value:    valid_f...a list of all valid functions
##
##  Examples for Yearby pairs: G a group: ( End(G), Aut(G) ),
##                             S a set :  ( T(G), Sym(G) ).
##
##  the format of valid_f is a list of lists s.t. each of those lists
##  represents a function f:S -> E
##
##  This is an implementation of the method Yearby introduced in chapter IV
##  of his dissertation.
##
ValidFunctionsYearby := function( E )
  local TFL, m, n, G, rec_set, valid_f, done, l, f, S, ext, elm, i,
        f1, L, diff, count, f2;

  TFL := List( E, e -> e.tfl );
  m   := Length( TFL[1] );
  n   := Length( TFL );
  G   := [1..m];
  rec_set := [ [ [ TFL[1] ], 1 ] ];
  valid_f := [];
  done    := false;
  count   := 0;

  repeat

    l    := Length( rec_set );
    f    := rec_set[l][1];
    S    := Filtered( [1..Length(f)], i -> IsBound(f[i]) );
    ext  := Extend( S, f );
    elm  := rec_set[l][2];
    i    := Position( TFL, f[elm] );
    f1   := ext[1];
    L    := ext[2];
    diff := Difference( G, L );

    if f1 = false or diff = [] then
      if diff = [] then
        f2 := List( f1, k -> Position( TFL, k ) );
        Add( valid_f, f2 );
      fi;
      if l = 1 then
        if i < n then
          rec_set := [ [ [ TFL[i+1] ], 1 ] ];
        else
          done := true;
        fi;
```

```
      else  # if l > 1
        if i < n then
          rec_set[l][1][elm] := TFL[i+1]; # assign a new image
        else
          repeat
            Unbind( rec_set[l] );
            l := l-1;
            elm := rec_set[l][2];
            f := rec_set[l][1];
            i := Position( TFL, f[elm] );
          until i < n or Length( rec_set ) = 1;
          if i < n then
            rec_set[l][1][elm] := TFL[i+1]; # assign a new image
          else
            done := true;
          fi;
        fi;
      fi;
   else  # extension succeeded only partially
      elm := diff[1];
      f1[elm] := TFL[1];
      Add( rec_set, [ f1, elm ] );
    fi;

  count := count + 1;

  until done;

  Print( "\nloop executions :", count, "\n\n" );
  return valid_f;
end;
```

B.8 The Source File csg.g

```
##############################################################################
##  File: CSG.G                                                             ##
##############################################################################
##
##############################################################################
##  This source file contains the stuff to compute and classify semigroups.
##############################################################################
##
##############################################################################
##
#F AllFunctions: create all functions from the set [1..n] into the set [1..n]
##                s.t. they are lex. ordered with [1,...,1] first, and with
##                the constant function [n,n,...,n] last.
##
AllFunctions := function( n )
  local af,i,j,k,success;
```

```
  i := [];
  for j in [1..n] do i[j] := 1; od;
  k := -1;
  af := [];

  while k <= n do
    AddSet( af, Reversed( i ) );
    k := 1; success := false;
    while not success and k <= n do
      if i[k] = n then i[k] := 1; k := k+1;
      else i[k] := i[k]+1; success := true;
      fi;
    od;
  od;

  return af;
end;

#############################################################################
##
#F  ValidFunctionsSg( m ) . . . . . Determine all functions f: S -> T (= S^S)
##                                  with f(f(a)(b)) = f(a) o f(b).
##  V1.0  15.2.95
##  input parameter: m.........a positive integer defining a size
##  return value:    valid_f...a list of all functions with the above property
##
##  the format of valid_f is a list of lists s.t. each of those lists
##  represents a function f: S -> T
##
ValidFunctionsSg := function( m )

  local n,       # help var: the size of T
        i,       # help var: a list of indices
        tuples,  # help var: all tuples out of [1..m]
        valid_f, # a list of the functions with the above property
        total,   # total number of functions to be considered
        k,       # a loop variable
        count,   # the number of the function currently being considered
        done,    # help var: indicates when to stop the loop
        f;       # help var: a function to be considered

  Print( "computing all functions...\n" );
  T       := AllFunctions( m );
  n       := m^m;
  i       := List( [1..m], k -> 1 );
  tuples  := Tuples( [1..m], 2 );
  valid_f := [];
  # compute the number of loop executions
  total     := n^m;
  Print( "computing valid functions...\n" );
```

```
  k     := -1;
  count :=  0;
  while k <= m and i[m] <= n do
    count := count + 1;
    Print( "  considering function ", count, " of ", total, "...\r" );
    f := Sublist( T, Reversed( i ) );
    if ForAll( tuples, t -> f[ f[t[1]][t[2]] ] = f[t[1]] { f[t[2]] } ) then
      Add( valid_f, Reversed( i ) );
    fi;
    k := 1;
    done := false;
    while not done and k <= m do
      if i[k] = n
        then i[k] := 1; k := k+1;
      else
        i[k] := i[k]+1; done := true;
      fi;
    od;
  od;

  Print( "\n" );
  return valid_f;

end;

#############################################################################
##
#F  ClassifySg( valid_f ). . . . . . . . . . . .determine iso classes of sgps
##  V1.0  15.2.95
##  input parameters: valid_f...a list of lists representing
##                              valid functions G -> E
##  return value:     classes...a record of iso classes
##
ClassifySg := function( valid_f )

  local R,       # a list, the entries stand for the elements of the set
        AF,      # the list of all functions
        A,       # a list of the bijections: R -> R
        vf,      # help var: the set of lists representing valid functions,
                 # will be dynamically reduced in each loop execution
        noc,     # contains the number of the current class
        classes, # return value: record which contains the computed classes
        vfcount, # help var: counts how many functions remain
        f,g,     # lists, representing functions in vf
        a,       # a list, representing a bijection
        loa;     # a list of automorphisms ( represented as lists )

  Print( "computing all functions...\n" );
  AF      := AllFunctions( Length( valid_f[1] ) );
  R       := [1..Length( AF[1] )];
  Print( "computing bijections...\n" );
  A       := Filtered( AF, e -> Size( Set( e ) ) = Size( R ) );
```

```
            Add( A, [] );
  vf      := Set( valid_f );
  noc     := 0;
  classes := rec();
  vfcount := Length( vf );

  Print( "classifying...\n" );
  for f in vf do
    noc := noc + 1;
    classes.(noc) := rec( phi := f );
    # initialize loa with the identity
    loa := [ Position( AF, R ) ]; # this adds the identity!

    Print( "functions to go: ", vfcount, "    \r" );

    Unbind( vf[ Position( vf, f ) ] ); vfcount := vfcount - 1;
    for g in vf do
      a := First( A, a -> a = []
               # this is isomorphisms
               or ForAll( R, x -> a{ AF[ f[x] ] } = AF[ g[a[x]] ]{ a } )
#               # this is anti-isomorphisms
#               or ForAll( R, x -> ForAll( R, y ->
#                                    a[AF[f[y]][x]] = AF[g[a[x]]][a[y]] ))
                );
      if a <> [] then
        AddSet( loa, Position( AF, a ) );
        Unbind( vf[Position( vf, g )] ); vfcount := vfcount - 1;
      fi;
    od;
    classes.(noc).bijs_yielding_iso_sgps := loa;
  od;

  Print( "\n" );
  return classes;
end;
```

B.9 The Source File `cl1.c`

```
/***************************************************************************/
/*  File: CL1.C                                                            */
/***************************************************************************/
/*  Classification program for nearrings V1.01 1.5.95                      */
/*  To be compiled with GCC.                                               */
/*  Produces GAP readable output.                                          */
/***************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define ORDER 15
```

```
typedef short FUN[ ORDER ];
static  short order;

long Find_First( FUN *, FUN *, FUN *, FUN *, long );
FUN  *file2mem( char*, short, long* );

main( int argc, char *argv[] ) {
  FILE   *fopen(), *fp_vf, *fp_out;
  FUN    *valid_f,  *f,  *g,  *vf,  *E,  *A;
  long   count_vf = 0, count_e = 0, count_a = 0, count1, count2, noc = 0, a;
  long   vfcount, noa;
  short  i, n, sizevg;
  char   s[2];
  time_t starttime, stoptime;
  vf = (FUN *)malloc( sizeof( FUN ) ); /* initialize vf ! */

  if ( argc != 5 ) {
    printf( "%s%s\n", "Error, Usage: cl1 <valid functions file> ",
    "<endomorphisms file> \n  <automorphisms file> <output file>" );
    return 0;
  }

  printf( "size of <size_t>: %d\n", sizeof( size_t ) );
  printf( "size of <int>   : %d\n", sizeof( int ) );
  printf( "size of <short> : %d\n", sizeof( short ) );

  fp_vf = fopen( argv[1], "rt" );
  if ( fp_vf == NULL ) {
    printf( "Error, cannot open file %s.", argv[1] );
    return 0;
  }
  fscanf( fp_vf, "%1s%1s", &s[0], &s[0] );
  order = 0;
  while( s[0] != ']' ) {
    fscanf( fp_vf, "%hd%1s", &n, &s[0] );
    /* printf( "just scanned: %c\n", s[0] ); */
    order++;
  }
  fclose( fp_vf );

  printf( "%s %hd %s", "order: ", order, "\n" );
  sizevg = ORDER * sizeof( **vf );
  printf( "sizevg: %hd\n", sizevg );

  valid_f = file2mem( argv[1], sizevg, &count_vf );
  if ( valid_f == 0 ) return 0;
  printf( "number of valid functions: %ld\n", count_vf );

  E = file2mem( argv[2], sizevg, &count_e );
  if ( E == 0 ) return 0;
  printf( "number of endomorphisms: %ld\n", count_e );
```

```
A = file2mem( argv[3], sizevg, &count_a );
if ( A == 0 ) return 0;
printf( "number of automorphisms %ld\n", count_a );

time( &starttime );
fp_out = fopen( argv[4], "wt" );
if ( fp_out == NULL ) {
  printf( "Error, cannot open file %s.", argv[4] );
  return 0;
}
fprintf( fp_out, "classes := rec(\n" );

count2 = count_vf; vfcount = count_vf;
for( f = valid_f; count2 > 0; count2--, f++ ) {

  printf( "to go: %ld; ", vfcount );

  /* ignore those functions already considered */
  while( (*f)[0] == 0 && count2 > 1) {
    f++; count2--;
  }

  for( i = 0; i < order; i++ ) {
    (*vf)[i] = (*f)[i];      /* work with a copy of f */
  }

  noc++;

  /* mark the currently considered f */
  (*f)[0] = 0; vfcount--;

  /* print the currently considered vf */
  fprintf( fp_out, "  %ld%s", noc, " := rec(\n    phi := [" );
  for( i = 0; i < order-1; i++ ) {
    fprintf( fp_out, "%hd%s", (*vf)[i], "," );
  }
  fprintf( fp_out, "%hd%s", (*vf)[order-1], "],\n" );
  fprintf( fp_out, "%s", "    autos_yielding_iso_nrs := [" );

  count1 = count_vf; noa = 1;
  for( g = valid_f; count1 > 0; count1--, g++ ) {
    while( (*g)[0] == 0 && count1 > 1) {
      g++; count1--;
    }
    if (  !(((*vf)[0]) == 0) && !(((*g)[0]) == 0) ) {
      a = Find_First( vf, g, E, A, count_a );
      if ( a != 0 ) {
        fprintf( fp_out, "%ld%s", a, "," );
        (*g)[0] = 0; vfcount--; noa++;
      }
    }
```

```
      if ( noa == count_a ) break;
    }
    printf( "count1: %ld\n", count1 );

    if (vfcount > 0)
      fprintf( fp_out, "%ld%s", count_e, "] ),\n" );
    else
      fprintf( fp_out, "%ld%s", count_e, "] )" );

    if (vfcount == 0) break;
  }
  fprintf( fp_out, "%s", ");\n" );
  fclose( fp_out ); free( valid_f ); free( E ); free( A );
  time( &stoptime );

  printf( "computation time in sec.: %.0f \n",
           (double)stoptime - (double)starttime );
  return 1;
}

/****************************************************************************/
long Find_First( FUN  *f, FUN  *g, FUN  *E, FUN  *A, long count_a ) {
/****************************************************************************/

  short x, y, i;
  long  count1;
  FUN   *a, *e;

  count1 = count_a;
  for( a = A; count1; count1--, a++ ) {
    for( x = 0; x < order; x++ ) {
      for( y = 0; y < order; y++ ) {
        if ( (*a)[ E [ (*f)[x]-1 ][y] - 1 ] !=
         E [ (*g)[ (*a)[x]-1 ] - 1 ][ (*a)[y] -1 ] )
          goto cont;
      }
    }

    /* find the right position of the automorphism in E */
    for( e = E, count1 = 1;; count1++, e++ ) {
      for( i = 0; i < order; i++ ) {
        if ( (*a)[i] != (*e)[i] ) goto fail;
      }
      return count1;
      fail:;
    }

    cont:;
  }

  return 0;
}
```

```
/*************************************************************************/
FUN *file2mem( char *filename, short sizevg, long *cnt ) {
/*************************************************************************/

  FILE  *fopen(), *fp;
  FUN   *vf, *f;
  char  c, s[2];
  long  count = 0, count2;
  short i, n;

  fp = fopen( filename, "rt" );
  if ( fp == NULL ) {
    printf( "Error, cannot open file %s.", filename );
    return 0;
  }
  /* determine how many valid functions there are */
  while( ( c = (char)fgetc( fp ) ) != EOF )
    if ( c == '[' ) count++;
  count--;

  /* allocate the memory for all the functions */
  vf = (FUN  *)calloc( (size_t)count, (size_t)sizevg );
  if( vf == NULL ) {
    printf( "Sorry, not enough memory." );
    return 0;
  }

  /* read the valid functions and store them */
  rewind( fp );
  count2 = count;
  fscanf( fp, "%1s", &s[0] );
  for( f = vf; count2; count2--, f++ ) {
    /* read the next valid function */
    fscanf( fp, "%1s", &s[0] );
    for( i = 0; i < order; i++ ) {
      fscanf( fp, "%hd%1s", &((*f)[i]), &s[0] );
      /* (*f)[i] = n;
         printf( "%hd,", (*f)[i] ); */
    }
    fscanf( fp, "%1s", &s[0] );
  }

  fclose( fp );
  *cnt = count;
  return vf;
}
```

Bibliography

[Baa78] Sara Baase. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley Publishing Company, Reading, Massachusetts; Menlo Park, California; London; Amsterdam; Don Mills, Ontario; Sydney, first edition, 1978.

[Cla68] James R. Clay. The near-rings on groups of low order. *Math. Zeitschr.*, 104:364–371, 1968.

[Cla70] James R. Clay. Research in near-ring theory using a digital computer. *BIT*, 10:249–265, 1970.

[Cla92] James R. Clay. *Nearrings: Genesis and Applications*. Oxford University Press, Oxford, 1992.

[CP61] A.H. Clifford and G.B. Preston. *Algebraic Theory of Semigroups*, volume 1. Amer. Math. Soc., Providence, 1961.

[How76] J. M. Howie. *An Introduction to Semigroup Theory*. Academic Press, London, New York, San Francisco, 1976.

[Lal79] Gerard Lallement. *Semigroups and Combinatorial Applications*. John Wiley & Sons, New York, Chichester, Brisbane, Toronto, first edition, 1979.

[Pet73] Mario Petrich. *Introduction to Semigroups*. Charles E. Merrill Publishing Co., Columbus, Ohio, 1973.

[Pil83] Günter Pilz. *Near-Rings*. North-Holland Publishing Company, Amsterdam, New York, Oxford, second edition, 1983.

[Pil89] Günter Pilz. *Algebra - Ein Reiseführer durch die schönsten Gebiete*. Universitätsverlag Rudolf Trauner, Linz, second edition, 1989.

[S+94] Martin Schönert et al. *GAP - Groups, Algorithms and Programming*. Lehrstuhl D für Mathematik, RWTH Aachen, July 1994. GAP manual for version 3 release 4.

[Yea73] Robert Lee Yearby. *A Computer Aided Investigation of Near Rings on Low Order Groups*. PhD Dissertation, University of Southwestern Louisiana, May 1973.