DESIGN AND IMPLEMENTATION OF A PRIMARY MEMORY VERSION OF ALDAT, INCLUDING RECURSIVE RELATIONS

AUGUST 1986

Normand Laliberté
School of Computer Science
McGill University
Montréal, Québec.

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science.

ABSTRACT

This thesis documents the creation of relix: *rel*ational database system on UN*IX*, an interactive tool for exploring the concept of the "relation as a primitive data unit".

Because relix was designed to provide a short response time, relations are assumed to fit in primary memory.

Aldat, the language offered to the user, is easy to use and algebraic in nature. It was designed as a stand-alone language with the relation as the unique unit of data. It offers the full power of the relational and domain algebras, including null values, to operate on relations. Relations can be defined recursively in a natural way. A simple mechanism to evaluate this type of relation is provided.

The work included building a translator which takes Aldat statements as input and produces intermediate code as output and an interpreter which performs the operations indicated by the code.

RESUME

Cette thèse documente la création de relix: système relationnel de base de données sur UNIX (*rel*ational database system on UN*IX*), un outil interactif pour explorer le concept de "la relation comme unité de donnée".

Relix fut conçu de façon à fonctionner avec un court délai de réaction, aussi nous présumons que les relations utilisées peuvent loger en zone primaire de mémoire.

Aldat, le langage offert à l'usager, est facile d'apprentissage, algébrique de nature et utilisé de manière autonome avec la relation comme élément atomique. Il comporte tout le pouvoir des algèbres des relations et des domaines, incluant divers éléments neutres, pour manipuler les relations. Une relation peut etre définie récursivement. Nous présentons un mécanisme simple pour évaluer ce genre de relation.

Notre travail inclut la construction des modules suivants: un programme de traduction qui à partir d'expressions en Aldat produit un code intermédiaire et un programme d'interprétation qui exécute les opérations indiquées par ce dernier code.

ACKNOWLEDGEMENTS

I would like to thank my thesis adviser, Professor T. H. Merrett, for many things: his dedication to my bringing of this work to completion; many hours of challenging discussion about the topics herein presented; his persistent demand for high standard; his support in organizing this thesis and starting it in the first place.

I am also indebted to several people who contributed to this work. To these people, I would like to express my gratitude. Especially, to those who helped me in tapping some of the UNIX system capabilities on various machines: Andrew Adaszkiewicz, Alf Lund, Matthew Odinotski and Charles Snow. To all the secretaries of the School for their encouragement. To Geoff Forbes: I have been honoured with his partnership in several school projects, many of them proving very relevant to this thesis.

Special thoughts to my friend Luc Boulianne for an indefinite number of services ranging from advising on programming style to providing valuable pieces of system software.

How can I not mention this other friend, Ann Chatwin, who proofread the typescript of this thesis and, above all, cheered me up constantly.

I am grateful to the School of Computer Science and, in particular, to Professor Merrett for their financial assistance.

TABLE OF CONTENTS

chapter I

INTRODUCTION

The purpose of this work was to develop a new implementation of Aldat: the *al*gebraic *dat*a language. We wanted to provide the users with an interactive version of this high level programming language for exploring the concept of the relation as the primitive unit of data. As well, we developed a prototype of view evaluation with the particular aim of providing a simple way to evaluate a special type of view: recursively defined relations.

Our main concerns were speed and ease of use. The first concern dictated that the relations on which the user wants to operate can fit into main memory. The second goal required us to redesign Aldat as a stand-alone language obeying a natural syntax. The resulting implementation, named relix, was intended to be highly portable. That is, only minor modifications should be required so that it can work on other machines running under UNIX.

a) A HISTORICAL PERSPECTIVE

Codd proposed relations as models for files and sets of relations as models for data in databases in 1970 [CODD70]. One of his goals was to release the user from the navigation problems entailed by the hierarchical and network models. Since then much research has been carried out in the field of relational database systems. This research comprises, among others, the following areas:

1.- query languages (see below).

2.- implementation techniques: [KIM 79].

3.- knowledge bases and expert systems: [KERS84].

4.- non-formatted data: [GARD84], [BARB85].

5.- distributed databases: [CERI84].

6.- concurrency control: [BERN83b].

7.- theory: [MAIE83b].

A query language allows the user to retrieve or modify the information in a database. Several approaches have been suggested and developed. Among these are:

1.- Tuple at a time: relations are processed tuple by tuple, reminding us of record scanning in earlier data processing. Theseus uses that type of processing [SHOP75].

2.- Algebra oriented: operations are defined to take whole relations as operands and yield a relation as a result. The loops are hidden in the operator. A language using this approach is ISBL (Information System Base Language) on PRTV (Peterlee Relational Test Vehicle) [TODD76].

3.- Calculus oriented: an expression is used to describe the data to be retrieved. The expression may be formulated in a language similar to first order predicate calculus. The system determines the means of finding the data. There are many implementations using this approach.

a) QUEL (QUEry Language) on INGRES (Interactive Graphics and Retrieval System) [STON76].

b) ARIEL (A RetrIEval Language) [MACG85].

c) For DRC (Domain Relational Calculus) and ILL (Intermediate Level Language) domains represent the sets of objects whereas relations are various kinds of association among these objects [LACR77].

We have presented these approaches in order of decreasing procedurality. A language is less procedural than another to the degree that the user can describe the result to be achieved rather than specify the actions to perform in order to achieve it. Calculus oriented languages intend to be less procedural than algebraic languages. There are languages half-way between calculus and algebraic languages. SQL (a version of SEQUEL [CHAM76], itself a version of SQUARE) is such a language [CHAM80]. It has been shown that both algebraic and calculus languages are equivalent in the following sense: a query expressible in one language can also be expressed in the other language (see [CODD71] or [ULLM82]).

In calculus languages the user is still induced to think in terms of tuples. On the contrary, algebraic languages consider the relations as the primitive unit of data and, thus, provide a high level of abstraction. Furthermore, the result of any algebraic operation on relations is a relation. This closure property guarantees that only one type of data need to be considered.

Aldat is such an algebraic language. Since Merrett [MERR77] proposed it as a programming language for which relations are the elements, different implementations have been realized at McGill as parts of what is now known as the Aldat project. Before the proposal of Aldat, the first relational database system developed at McGill was MRDS a data sub-language for PL/I [MERR76]. This system provided the user with project, select and the complete array of set theoretic relational functions called the μ-join. This system was implemented in Pascal as MRDSP [MERR81], with the addition of the σ-join operation, an extension of Codd's division [CODD79].

Up to this point, implementations had been done on main frames. This proved expensive even if these systems were not interactive. Many of the following implementations were carried out on microcomputers. These were believed to provide a less costly environment, suitable to develop that type of software. MRDSA [CHIU82], the UCSD Pascal version was implemented in 1982 as a data sub-language on an Apple II microcomputer. This system, because of its limited resources, simplified data handling as much as possible. MRDSA directly imitates the relational view of data at the storage level. Attribute values are stored as character data, tuple by tuple in a contiguous physical location on the disk. No mechanisms for data compression or optimized retrieval have been implemented. MRDSA's main purpose was to demonstrate the power of the relation as a model for data processing. This system provided the user with the extended set of relational operations, in particular, project, μ-join, σ-join, full screen relational editor and a QT-select function: an extension of the select operation [MERR84a]. These operations are called from a Pascal program via a system of library subroutines. The user is required to specify which attributes to use in the operation. MRDSA does not implement the concept of domain types. That is, each attribute is a character string and any attribute can be used with any other attribute in operations of the relational algebra.

MRDS/FS [VANR83], MRDS with functional syntax, used MRDSA on an IBM PC as the basis for an interactive relation manipulation system. It extended MRDSA by adding a domain algebra facility which allows the definition of new attributes as a function of already existing attributes. In addition, MRDS/FS has released the user from the burden of writing and compiling Pascal programs. This, by creating a system where relational expressions are entered interactively, interpreted and evaluated through calls to MRDSA procedures. It is noteworthy that the relation is the only data structure available to the user. MRDS/FS is an interactive interpreter for relational expressions. It provides the user with a complete set of relational programming functions: relational algebra functions, domain algebra functions, conditional execution functions, branching functions and housekeeping functions. These functions allow the user to create views of the database, including recursive relations. With the conditional execution and branching functions just mentioned the user can build loops to define views. This system comprises two modules. The first one analyses the input relational expressions, detects errors and converts these expressions into MRDSA-procedure calls. The second module executes these procedure calls. MRDS/FS was developed on an IBM PC after appropriate modifications to MRDSA,,the underlying system. This, because the Apple II was taxed to its limits. The size of the relations handled by MRDS/FS is quite large, considering that it is not intended for commercial use. A database is constrained to fit within a set of fifty diskettes.

The moving of MRDS from the main frames to microcomputers did not fully satisfy the need for a highly interactive system. MRDS/FS is judged too slow to be used interactively: response time increases rapidly with the complexity of the operations performed and the size of relations involved in these operations. Moreover, its functional syntax, although theoretically appealing, is not easily mastered and may not seem very intuitive in the context of data processing.

We conclude now our review of the various MRDS implementations. They supported the following view: the relation together with the relational and domain algebras provide the user with a powerful tool to query and modify the information present in a database. However, they did not provide the user with a fast system or an easy to use query language.

The microcomputer implementations were cheaper to develop and use than the previous mainframe implementations, but they are too much slower. Their query language is either embedded in a host programming language or uses a syntax which, at this point, is difficult to exploit even by sophisticated users.

Finally, there is an important consideration of completeness. Merrett [MERR77] observed that the relational algebra was incomplete because it had to be embedded in a programming language with loop structures to solve some kinds of problem, especially least-fixed point problems [AHO 79]. Kamel's implementation of Aldat [KAME80] embedded the relational algebra in a Pascal-like language permitting loops. We saw above that MRDS/FS allowed the user to construct loops. We aim for a version of Aldat which uses recursion to achieve the same end.

These considerations motivate our goal stated above: provide the user with a fast, easy to use implementation of Aldat. The main contributions of this thesis are·

1.- Aldat has been redesigned as a stand-alone programming language, providing means to create views or recursively defined relations. The loop structures needed to evaluate these views are hidden in the implementation. The relation is the unique data structure available to the user.

2.- Aldat has been implemented on a truly portable operating system, namely UNIX, running currently on the following machines: Cadmus, Masscomp and Vax-780. We repeat the following important restriction: relations must be small enough so that the operands, at most two in any case, of any operation of the relational algebra can fit into primary memory. This is in order to produce a system with as short a response time as possible. It is also consistent with the way in which UNIX treats files. Furthermore, relix has been designed so that features of Aldat not supplied by our implementation can be easily added: for example, a relational editor, QT-selectors, the σ-join and others.

Parts of this work constitute extensions to a project done with Geoff Forbes in the course 308-573 on Minicomputers [FORB85].

b) THESIS OUTLINE

The development of the thesis followed the steps listed below.

1.- An unambiguous LALR-1 Aldat grammar was produced. It follows as much as possible the notation and conventions presented in [MERR84a]. The differences are justified by the restrictions imposed by the system on which we were working.

2.- This grammar together with routines to perform semantic checking and code generation were fed into a parser generator. This provided a translator which takes as input Aldat-statements and produces as its output intermediate code.

3.- We built an interpreter which transforms that intermediate code into function or procedure calls in order to perform the operations of the relational or domain algebra. We supplied the routines for the project, select and domain algebra operations. As well, routines are provided to perform error checking and recovery where possible.

4.- On top of the interpreter we added a mechanism to evaluate recursively defined relations.

5.- Producing a relational editor or implementing the join operations were not part of this thesis. Facilities are supplied to overcome the first limitation. In particular, the means we provided to escape to the host operating system permit us to use UNIX editors for relations. Ann T. Chong implemented the μ-join [CHON86].

With respect to the Aldat language described in appendix A, the implementation is complete up to the code generation phase. Past this point, work remains to be done in order to provide the σ-join and operations on domains of type real.

This thesis will outline how the above steps were achieved. It is divided into nine chapters. The first one has stated objectives and placed the work in historical perspective. Chapter II describes the terminology: relations, relational and domain algebra, views

Chapter III constitutes the user's manual It specifies the exact syntax mentioned in step 1. That is, it shows how the user can enter Aldat statements or use the facilities developed in step 5. Chapter IV describes the parser and the semantic analyser required to build the translator mentioned in step 2. It also details the construction of the interpreter of step 3. Chapter V explains the implementation of the domain algebra operations. Chapter VI does the same for the relational algebra operations. Chapter VII describes the view evaluation mechanism of step 4. Error handling is detailed in chapter VIII.

Because this implementation was designed as the basis on which one could develop a more elaborate system, Chapters IV through VII detail at length the implementation. They can be seen as forming a programmer's manual. Chapter IX, the conclusion, indicates some directions for further research.

chapter II

BASIC RELATIONAL CONCEPTS

Many textbooks cover to some extent relational database systems and query languages: [DATE82], [KORT86], [MAIE83b], [MERR84a], [OZKA86] and [ULLM82]. In particular, [MAIE83b] and [MERR84a] deal exclusively with these topics. The aim of this chapter is to provide the user with general definitions of relations and of both the relational and domain algebras.

a) DEFINITION OF RELATION

Definition: a relation on N, not necessarily distinct, sets $S_1,...,S_n$ is a set of N-tuples each of which has its first element from S_1, ..., its N-th from S_n.

In other words, it is a subset of the cartesian product of $S_1,...,S_n$. It can be seen as a table with the following properties:

1.- all rows are distinct and their ordering is immaterial;

2.- each column is assigned a unique name; so, their ordering is immaterial;

3.- all entries in each row and under each column are atomic.

Each row represents a tuple. A column is referred to as a domain or attribute and its underlying set as a domain type. A domain may occur only once in a relation whereas a domain type may be used many times. Atomicity depends on the operations defined on the domain type. The degree of a relation is taken to be the number of domains on which it is defined. A database is a set of time-varying relations.

Throughout this thesis, most of the examples are taken from SCHOOL, a small database containing, among others, the following domains and relations.

DOMAINS

name	*type*	*length*	*description*
NAME	string	26	student name
STUID	"	7	student id number
SEC	"	2	section
YEAR	"	4	current year
A1	integer	11	assignment 1
A2	"	11	assignment 2
MID	"	11	midterm
FIN	"	11	final
FEES	"	11	fees paid
CRED	"	11	credits in current year

RELATIONS

MARKS_420

NAME	STUID	SEC	A1	A2	MID	FIN
arrau, antonina	8192214	A	18	20	9	42
berard, paulette	8314201	C	23	21	11	40
brady, vivian	8230267	A	11	17	8	44
christos, marilou	8215291	B	13	19	11	38
giroux, aline	8314626	A	20	16	12	46
hart, terry	8317112	A	12	11	8	25
jones, raymond	8215174	B	13	17	7	30
king, tam	8328521	C	17	22	12	36
lamontagne, paul	7913295	B	20	20	11	43
rivet, maurice	8214512	C	16	21	9	41

CLASS

NAME	STUID	SEC	FEES
arrau, antonina	8192214	A	200
berard, paulette	8314201	C	452
brady, vivian	8230267	A	117
christos, marilou	8215291	B	398
giroux, aline	8314626	A	200
jones, raymond	8215174	B	-50
king, tam	8328521	C	34
lamontagne, paul	7913295	B	171

DEPT

NAME	YEAR	CRED
brady, vivian	1984	13
brady, vivian	1985	15
jones, raymond	1983	16
jones, raymond	1984	15
jones, raymond	1985	16
rivet, michel	1982	15
rivet, michel	1983	12
rivet, michel	1984	14

b) RELATIONAL ALGEBRA

Relations can be introduced as a new data type in a programming language and an extended relational algebra added as operations on that type. These operations take relations as operands and yield a relation as a result. They include assignment, projection and join on chosen domains, selection of different tuples.

Our examples do not fully illustrate the exact syntax described in chapter III. The complete grammar is found in appendix A.

Notice that a domain name can denote a list of domains. For example, if a relation R is defined on domains A, B, C, D, E then we can say that R is defined on X and Y where X= { A, D} and Y= { B, C, E}.

ASSIGNMENT: this operation assigns a value to a relation name. It acts in the same way that assignment of values to variables acts in programming languages.

TEST <- MARKS_420

TEST

NAME	STUID	SEC	A1	A2	MID	FIN
arrau, antonina	8192214	A	18	20	9	42
berard, paulette	8314201	C	23	21	11	40
brady, vivian	8230267	A	11	17	8	44
christos, marilou	8215291	B	13	19	11	38
giroux, aline	8314626	A	20	16	12	46
hart, terry	8317112	A	12	11	8	25
jones, raymond	8215174	B	13	17	7	30
king, tam	8328521	C	17	22	12	36
lamontagne, paul	7913295	B	20	20	11	43
rivet, maurice	8214512	C	16	21	9	41

PROJECT: project creates a relation which is a vertical subset of the operand relation, that is, a subset of the attributes (columns) from the operand relation are copied to a new relation. Any duplicates created in the process are eliminated. In other words, this operation specifies a subset of the attributes of a relation and the resulting relation is defined on those attributes.

Definition: let R be a relation defined on the domains A and B; the projection of R on A is defined by

$$R\,[A] = \{\ a \mid a \in A \text{ and } (a,b) \in R \text{ for some } b \in B\}.$$

For example, let R be MARKS_420, A= { NAME, MID, FIN} and B= { STUID, SEC, A1, A2}.

MARKS_420 [NAME, MID, FIN]

NAME	MID	FIN
arrau, antonina	9	42
berard, paulette	11	40
brady, vivian	8	44
christos, marilou	11	38
giroux, aline	12	46
hart, terry	8	25
jones, raymond	7	30
king, tam	12	36
lamontagne, paul	11	43
rivet, maurice	9	41

SELECT: select creates a relation which is a subset of the operand relation by including only those tuples which satisfy a given condition. It is required that each tuple of the operand relation contains all the information necessary to decide the truth value of the condition determining membership in the result relation.

Definition: let R be the same relation as above; let σ be a logical expression involving any number of occurrences of the following elements only:

-A, B, constants of the same domain type as A or B

-logical operators: and, or ,not

-comparison operators: $=$, $\neq$, $<$, $>$, $<=$, $>=$;

the select of R based on σ is defined by

$$R\,[\,\sigma] \doteq \{\,(a,b) \mid \sigma \text{ is true}\}.$$

For example, let R be MARKS_420, A= { SEC, FIN} and σ= FIN > 40 and SEC $\neq$ "B"

MARKS_420 [σ]

NAME	STUID	SEC	A1	A2	MID	FIN
arrau, antonina	8192214	A	18	20	9	42
brady, vivian	8230267	A	11	17	8	44
giroux, aline	8314626	A	20	16	12	46
rivet, maurice	8214512	C	16	21	9	41

JOIN: join performs generalized set operations (union, intersection, cartesian product and the like) on pairs of operand relations. In general, the operands have common attributes which are used to determine which of their tuples will be combined to participate in the result. We consider first the μ-join and then the σ-join.

Definition: consider two relations R(A, B) and S(C, D) with B and C defined on common domain types; let DC be a constant representing irrelevant information, "don't care". We define the μ-join of R and S in union mode, denoted ujoin, by

R [B ujoin C] S = left_wing $\cup$ center $\cup$ right_wing

where

left_wing= $\{ (x,y,DC) | (x,y) \in R \text{ and for all } z, (y,z) \notin S \}$

center= $\{ (x,y,z) \mid (x,y) \in R \text{ and } (y, z) \in S \}$

right_wing= $\{ (DC,y,z) | (y,z) \in S \text{ and for all } x, (x,y) \notin R \}$

the other modes of the μ-join of R and S are:

natural	R [B ijoin C] S =	center
left join	R [B ljoin C] S =	left_wing $\cup$ center
right join	R [B rjoin C] S =	center $\cup$ right_wing
symmetric difference join	R [B sjoin C] S =	left_wing $\cup$ right_wing
left difference	R [B dljoin C] S =	left_wing
right differemce	R [B drjoin C] S =	right_wing

MARKS_420 [NAME ijoin NAME] DEPT

NAME	STUID	SEC	A1	A2	MID	FIN	YEAR	CRED
brady, vivian	8230267	A	11	17	8	44	1984	13
brady, vivian	8230267	A	11	17	8	44	1985	15
jones, raymond	8215174	B	13	17	7	30	1983	16
jones, raymond	8215174	B	13	17	7	30	1984	15
jones, raymond	8215174	B	13	17	7	30	1985	16

where ijoin denotes the natural or intersection join.

Definition: let R(A, B) and S(C, D) be as in the previous definition. For a $\bar{\in}$ A let $R(a) = \{ b | (a,b\} \in R\}$. Observe that R(a) is a subset of b; it is the set of values of B associated with a given element of A. Similarly, for d ∈ D let $S(d) = \{ c | (c,d) \in S\}$. We define the σ-join of R and S as an extension of the division first proposed by Codd [CODD71]. This family has six primitive modes based on the following set comparisons:

mode	*description*
eqjoin	equal
ltjoin	proper subset
lejoin	subset
gtjoin	proper superset
gejoin	superset
iejoin	empty intersection

which yield the following:

$$\begin{aligned}
R[\text{ B eqjoin C }]S &= \{ (a,d) | R(a) = S(d)\} \\
R[\text{ B ltjoin C }]S &= \{ (a,d) | R(a) \subset S(d)\} \\
R[\text{ B lejoin C }]S &= \{ (a,d) | R(a) \subseteq S(d)\} \\
R[\text{ B gtjoin C }]S &= \{ (a,d) | R(a) \supset S(d)\} \\
R[\text{ B gejoin C }]S &= \{ (a,d) | R(a) \supseteq S(d)\} \\
R[\text{ B iejoin C }]S &= \{ (a,d) | R(a) \cap S(d) = \emptyset\}
\end{aligned}$$

There are six complementary modes obtained by prefixing each basic mode with a negation. For example:

$$R[\text{ B not eqjoin C }]S = \{ (a,d) | R(a) \neq S(d)\}.$$

In the next section we illustrate the use of icomp, the natural or intersection composition, which is not iejoin. That is,

$$R[\text{ B icomp C }]S \equiv R[\text{ B not iejoin C }]S$$

Although powerful, the relational algebra cannot handle computations across tuples or along domains. Moreover, it is incomplete in the sense that it does not include a loop structure and, hence, can not solve least fixed-point problems.

c) DOMAIN ALGEBRA

The domain algebra is a facility whereby new attributes can be created as some function of existing attributes in a given relation. For example, a domain, say MID_FIN, can be defined as the sum of two other attributes, say MID and FIN. Another domain, say TOTAL_MID_FIN, can be defined as the sum of all the values in the attribute MID_FIN in a given relation.

The contents of a relation can therefore be transformed both horizontally, inside a tuple, and vertically, across tuples to create new attributes which can be used like any other attribute. The creation of a new relation with values for those attributes can be achieved using the project operation.

Functions defining virtual domains are composed of domain operations which fall into two categories: horizontal and vertical. Horizontal domain operations have operands which do not cross tuple boundaries. That is, all the operands for that operation are found in the same tuple. For example,

let MID_FIN be MID + FIN

where both MID and FIN are already defined attributes.

An actual domain is an attribute which exists in a given relation. MID_FIN is said to be virtual because it does not presently exist in any relation. When MID_FIN is actualized, again through a project operation, it will contain in each tuple the sum of the values of MID and FIN for that tuple.

MARKS_420[NAME, MID, FIN, MID_FIN]

NAME	MID	FIN	MID_FIN
arrau, antonina	9	42	51
berard, paulette	11	40	51
brady, vivian	8	44	52
christos, marilou	11	38	49
giroux, aline	12	46	58
hart, terry	8	25	33
jones, raymond	7	30	37
king, tam	12	36	48
lamontagne, paul	11	43	54
rivet, maurice	9	41	50

Assignment of a constant value to a domain is a special type of horizontal domain operation.

let SPECIAL_FEE be 13

creates a virtual domain whose value is 13. A constant domain therefore is a domain which has the same value for all relations, all tuples.

By comparison with horizontal domains, vertical domains have operands which are a result of a function on values from one or more tuples. Four classes of vertical domain operators can be defined.

Reduction (RED) is a class of domain operators which perform some binary operation on an attribute over every tuple in the relation in order to produce a single result. For example,

let TOTAL_MID_FIN be red + of MID_FIN

produces a new attribute which is a sum of all the values in the MID_FIN attribute. Conceptually, TOTAL_MID_FIN will have the same value for each tuple.

MARKS_420[NAME, MID_FIN, TOTAL_MID_FIN]

NAME	MID_FIN	TOTAL_MID_FIN
arrau, antonina	51	483
berard, paulette	51	483
brady, vivian	52	483
christos, marilou	49	483
giroux, aline	58	483
hart, terry	33	483
jones, raymond	37	483
king, tam	48	483
lamontagne, paul	54	483
rivet, maurice	50	483

Equivalence reduction (EQUIV) is a type of reduction by which a relation is first stratified into sets of tuples having the same value for one or more domains. A separate reduction operation is then performed on each stratum or equivalence class.

let FEES_BY_SEC be equiv + of FEES by SEC

defines an attribute which will be calculated by first stratifying the relation by section and then performing the reduction operation on each stratum, that is, the summing up of fees paid by each student.

CLASS[SEC, FEES_BY_SEC]

SEC	FEES_BY_SEC
A	517
B	519
C	486

Functional mapping (FUN) is another class of vertical domain operator. It acts upon a relation on which an ordering can be induced by one or more attributes. This allows the exploration of a relationship between successive tuples. A FUN operation performs some binary operation, but unlike RED which produces a single value for the result attribute, it performs an operation and stores the current result in the resulting attribute. That is,

value of the result attribute = for current tuple	value of attribute for previous tuple	OP	value of operand domain

where OP is some binary operator. If the current tuple is the first tuple, then the value of the attribute for the previous tuple is taken to be the identity element for the operator OP.

It is stressed that the ordering attribute must functionally determine the operand attribute in order for the result to be meaningful. If there is a group of tuples with the same value for the ordering attribute then this group of tuples will be treated as a single tuple. That is, the operation will be performed only once for the group and the result attribute will have the same value for all tuples in the group.

For example, given

let CUM_CRED be fun + of CRED order YEAR

the result attribute represents the cumulative number of credits for each year. All tuples with the same value for YEAR, were there any, would be treated as a single tuple, thereby entering into the operation only once and having the same result value.

Suppose that JONES <- DEPT[NAME = "jones, raymond"].

JONES[YEAR, CRED, CUM_CRED]

YEAR	CRED	CUM_CRED
1983	16	16
1984	15	31
1985	16	47

Partial functional mapping (PAR) is an extension of functional mapping. The relation is first stratified over specified domains. A separate functional mapping is then performed over each of these strata. In other words, PAR is to FUN what EQUIV is to RED. For example,

let CUM_CRED_PER_STUDENT be par + of CRED order YEAR by STUDENT

DEPT[NAME, YEAR, CRED, CUM_CRED_PER_STUDENT]

NAME	YEAR	CRED	CUM_CRED_PER_STUDENT
brady, vivian	1984	13	13
brady, vivian	1985	15	28
jones, raymond	1983	16	16
jones, raymond	1984	15	31
jones, raymond	1985	16	47
rivet, michel	1982	15	15
rivet, michel	1983	12	27
rivet, michel	1984	14	41

d) VIEW DEFINITION AND RECURSIVE RELATIONS

Just as new domains can be defined as a function of previously defined domains, so can new relations. A view of a database is a relation derived from the given relations of the database by some expression using the relational and domain algebra. Among other advantages, views offer the possibility of defining relations recursively which, in turn, allows least fixed-point operations like computing the transitive closure of a graph.

Consider a relation called PARENT and defined on SENIOR and JUNIOR which are domains of type string and length 18. If "edward IV elizabeth of york " is a tuple of PARENT it indicates that edward IV is a parent of elizabeth of york. In order to find for any two persons whether one is a descendant of the other, we compute the transitive closure of PARENT and call the result ANCESTOR defined on SENIOR and JUNIOR. We have the following:

ANCESTOR is PARENT [ujoin]

(ANCESTOR [JUNIOR icomp SENIOR] ANCESTOR)

where icomp is the natural composition and PARENT contains:

PARENT

SENIOR	JUNIOR
edward IV	elizabeth of york
elizabeth of york	henry VIII
elizabeth of york	margaret
henry VII	henry VIII
henry VII	margaret
henry VIII	edward VI
henry VIII	elizabeth I
henry VIII	mary I
james IV stewart	james V stewart
james V stewart	mary stewart
margaret	james V stewart

ANCESTOR

SENIOR	JUNIOR
edward IV	edward VI
edward IV	elizabeth I
edward IV	elizabeth of york
edward IV	henry VIII
edward IV	james V stewart
edward IV	margaret
edward IV	mary I
edward IV	mary stewart
elizabeth of york	edward VI
elizabeth of york	elizabeth I
elizabeth of york	henry VIII
elizabeth of york	james V stewart
elizabeth of york	margaret
elizabeth of york	mary I
elizabeth of york	mary stewart
henry VII	edward VI
henry VII	elizabeth I
henry VII	henry VIII
henry VII	james V stewart
henry VII	margaret
henry VII	mary I
henry VII	mary stewart
henry VIII	edward VI
henry VIII	elizabeth I
henry VIII	mary I
james IV stewart	james V stewart
james IV stewart	mary stewart
james V stewart	mary stewart
margaret	james V stewart
margaret	mary stewart

In this example we used an icomp. Observe that the same result can be obtained by performing an ijoin followed by a project which eliminates the joining attributes.

Indeed, the set of operators described in this chapter is very rich. Some authors like [KORT86], [MAIE83b] and [ULLM82] define a set of elementary, non redundant operators and then express the other operators in terms of the previous. This is interesting from a theoretical point of view. However, our goal is to supply the user with a simple conceptual framework.

The next chapter explains, among other things, the exact syntax used to enter Aldat statements.

chapter III

USER'S MANUAL

Relix is an Aldat interactive emulator offering a way to experiment with both the relational and domain algebras. It is assumed that there is enough room in main memory for the operands and result, join excepted, of any relational operation. It is interactive in the sense that it accepts and executes one statement at a time, as opposed to collecting statements and waiting for a special instruction from the user to start execution. A fair knowledge of the main characteristics of an Aldat-like language, as described in [MERR84a], constitutes a definite asset; also, the user should be decently familiar with UNIX command language: cp, rm, vi, not to mention login, etc. The present manual comprises the following sections:

a) Getting Started Using System Commands
b) Domain Algebra
c) Relational Algebra

a) GETTING STARTED USING THE SYSTEM COMMANDS

Bear in mind the following:

-relix is command driven, as opposed to menu driven

-user input lines are in italic characters

-relix output lines are in bold characters

Suppose you want to create a database named SCHOOL comprising the following relations: MARKS_420 defined on NAME, STUID, SEC, A1, A2, MID and FIN, CLASS on NAME, STUID, SEC and FEES.

The type of the attributes NAME, STUID and others is specified below. When you have the UNIX prompt, say '%', type in

% relix SCHOOL

You soon get the relix prompt: '>'. Let us inspect the current state of dom_table (respectively rel_table and rd_table) with sd! (sr! and srd! respectively). At this point, they contain information about the system relations only and _NULL. This relation is attributeless but may contain one tuple. Among others, the σ-join, to be discussed further, can make use of it.

sdl

Domain Table: SCHOOL

Index	Name	Length	Actual	Type
0	dom_name	20	T	STRG
1	rel_name	20	T	STRG
2	length	11	T	INTG
3	type	11	T	INTG
4	tuple_size	11	T	INTG
5	ntuples	11	T	INTG
6	count	11	T	INTG
7	dom_pos	11	T	INTG
8	sort_rank	11	T	INTG

srl

Relation Table: SCHOOL

Index	Name	Tsize	Ntuples	Arity	Domains
0	DOM	42	16	3	dom_name length type
1	REL	42	6	3	rel_name tuple_size ntuples
2	RD	73	20	5	rel_name dom_name count dom_pos sort_rank
3	_NULL	0	0	0	

srdl

RD Table: SCHOOL

Relation	Domain	Count	Position	Rank
DOM	dom_name	UNKNOWN	0	
DOM	length	UNKNOWN	20	
DOM	type	UNKNOWN	31	
REL	rel_name	UNKNOWN	0	
REL	tuple_size	UNKNOWN	20	
REL	ntuples	UNKNOWN	31	
RD	rel_name	UNKNOWN	0	
RD	dom_name	UNKNOWN	20	
RD	count	UNKNOWN	40	
RD	dom_pos	UNKNOWN	51	
RD	sort_rank	UNKNOWN	62	

When displaying the contents of rel_table we will not show the entries corresponding to the system relations any more. To get on-line information about the system commands, type:

```
h!

          ar!       -->   append some tuples to an existing relation
          batch!    -->   switch mode to batch
          cd!       -->   create a new domain
          cr!       -->   create a new relation
          dd!       -->   delete an existing domain
          dr!       -->   delete an existing relation
          h!        -->   display the current table
          input!    -->   redirect standard input to a UNIX file
          man!      -->   display the manual on screen
          po!       -->   display the code generated
          pr!       -->   display a relation on the screen
          q!        -->   return to UNIX
          sa!       -->   save existing relations
          sd!       -->   display the contents of dom table
          sh!       -->   get a set of shell commands and execute
          sr!       -->   display the contents of rel table
          srd!      -->   display the contents of rd table
```

To get the on-line copy of the current manual through ´more´, a UNIX facility, type:

```
man!
```

To add some domains use the command cd! (create domain).

```
cd!

enter domain name( or ´e!´ to exit):

NAME

enter the number corresponding to the desired type
  1.- boolean
  2.- integer
  3.- float
  4.- string
number:

4

enter length of string between (1 and 40):

26
```

you continue this way to enter STUID, SEC, A1, A2, MID and FIN. When you are finished with entering new domains, you exit using e! (exit).

```
enter domain name( or ´e!´ to exit):
```

e!

Suppose you entered erroneously a2 and then A2 as desired. You can delete a2 using dd! (delete domain). You can remove unused domains only.

dd!

```
enter ´p!´ if you want to be prompted with the name of
domains that can be deleted ( ´e!´ to exit):
```

The option ´p!´ is easier to use since you need not remember the spelling of the domains to be deleted. However, if there are only few domains to remove you are better off just hitting ´return´. In this case, you get:

```
enter domain name( or ´e!´ to exit): a2
enter domain name( or ´e!´ to exit): e!
```

sd!

```
              Domain Table: SCHOOL
Index   Name          Length   Actual   Type
  0     dom_name        20       T      STRG
  1     rel_name        20       T      STRG
  2     length          11       T      INTG
  3     type            11       T      INTG
  4     tuple_size      11       T      INTG
  5     ntuples         11       T      INTG
  6     count           11       T      INTG
  7     dom_pos         11       T      INTG
  8     sort_rank       11       T      INTG
  9     NAME            26       T      STRG
 10     STUID            7       T      STRG
 11     SEC              2       T      STRG
 13     A1              11       T      INTG
 14     A2              11       T      INTG
 15     MID             11       T      INTG
 16     FIN             11       T      INTG
```

Observe that the twelfth entry is missing; it has been occupied by a2. That is, no garbage collection is performed on dom_table. Let us create a new relation and enter some tuples. This is done through cr! (create relation).

```
cr!

enter relation name( or 'e!' to exit):

MARKS_420

enter domain name( or 'e!' to end):

NAME
```

and then STUID, SEC, A1, A2, MID, FIN to finish with

```
enter domain name( or 'e!' to end):

e!

relation MARKS_420 is defined on 7 domains
tuple size= 79
```

Observe that 79 is the sum of the width of the attributes on which MARKS_420 is define. It is now possible to enter a few ~~tuples~~. Relix prompts you thus:

```
enter 'a!' if you want to add a few tuples
   or 'fl' if a corresponding file already exists
   or 'e!' to exit:

a!

enter the maximum number of tuples to append:

15
```

we may enter up to fifteen tuples

```
enter 'e!' to end ( anything else to continue):
```

enter value for > NAME <:
(dc, dk or any string of length <= 26)

rivet, maurice

and then 8214512, 3, 16, 21, 9 and 41 for STUID, SEC, A1, A2, MID and FIN respectively. And so on until you have entered fifteen different tuples or replied with e! to

enter 'e!' to end (anything else to continue):

Let us enter another tuple: paulette berard, 8314201, 3, 23, 21, 11, 40 and then

e!

relation MARKS_420 contains 2 tuples

enter relation name(or 'e!' to exit): e!

Notice: we are still in create rel. To exit, enter:

e!

We can look back at the tuples just input using pr! (print relation).

pr!

enter relation name(or 'e!' to exit):

MARKS_420

```
NAME               STUID     SEC   A1   A2   MID   FIN
berard, paulette   8314201    C    23   21   11    40
rivet, maurice     8214512    C    16   21    9    41
```

You can resume adding tuples to MARKS_420 using ar! (append some tuples to an existing relation).

ar!

```
enter relation name( or ´el´ to exit): MARKS_420
enter the maximum number of tuples to append: 20
```

and entering appropriate values to each of NAME, STUID, SEC, A1, A2, MID and FIN. Assume that the following have been input:

NAME	STUID.	SEC	A1	A2	MID	FIN
arrau, antonina	8192214	A	18	20	9	42
berard, paulette	8314201	C	23	21	11	40
brady, vivian	8230267	A	11	17	8	44
christos, marilou	8215291	B	13	19	11	38
giroux, aline	8314626	A	20	16	12	46
hart, terry	8317112	A	12	11	8	25
jones, raymond	8215174	B	13	17	7	30
king, tam	8328521	C	17	22	12	36
lamontagne, paul	7913295	B	20	20	11	43
rivet, maurice	8214512	C	16	21	9	41

Alternatively, we can turn a UNIX file into a relation using cr! and f!. Let us define a relation CLASS on NAME, STUID, SEC and FEES. Assume FEES has been defined as an integer domain and the file CLASS created thus:

```
brady, vivian       8230267   A00000000117
giroux, aline       8314626   A00000000200
lamontagne, paul    7913295   B00000000171
christos, marilou   8215291   B00000000398
arrau, antonina     8192214   A00000000200
jones, raymond      8215174   B-0000000050
king, tam           8328521   C00000000034
berard, paulette    8314201   C00000000452
```

Observe: -the order in which domain values are entered in the file must be the same as the order specified when using cr!;

-domains of type string must be right-padded with blanks, those of type integer left-padded with zeroes, so that all the tuples have the same length;

-an optional minus sign appears in the leftmost position of the eleven-byte integer field;

-had we a boolean domain, we would have entered ´0´ for FALSE and ´1´ for TRUE.

The dialogue goes thus:

```
enter 'a!' if you want to add a few tuples
  or 'f!' if a corresponding file already exists
  or 'e!' to exit:
```

f!

```
enter (positive) number of tuples:
```

20

```
*** WARNING *** ladt.c: icrel fill: no more data to read
```

A warning is issued because the number of tuples is smaller than expected: not a serious offense (see error handling in chapter VIII). Using pr!, you get:

```
NAME                 STUID      SEC   FEES
arrau, antonina      8192214    A      200
berard, paulette     8314201    C      452
brady, vivian        8230267    A      117
christos, marilou    8215291    B      398
giroux, aline        8314626    A      200
jones, raymond       8215174    B      -50
king, tam            8328521    C       34
lamontagne, paul     7913295    B      171
```

Let us inspect rel_table with sr! assuming that some other relations have been created with cr! or the relational algebra (again, see section c).

```
                        Relation Table: SCHOOL
Index   Name          Tsize   Ntuples   Arity   Domains
  3     _NULL           0        0        0
  4     MARKS_420      79       10        7     NAME STUID SEC A1
                                                A2 MID FIN
  5     CLASS          46        8        4     NAME STUID SEC FEES
  6     RESULT         29       10        3     STUID A1 A2
  7     RES_SEC        13        3        2     SEC TOT_SEC
```

You may delete some relations. This can be done with dr! (delete relation).

dr!

```
WARNING: if you delete any relation used in any view
      it is safer to quit after execution of this command

enter relation name( or 'e!' to exit):
```

entering RES_SEC and RESULT will cause their removal from rel_table. The same result can be achieved with sa! (prompt the user to find which relations to save).

sa!

```
for each relation enter y/n depending whether
you want to save it or not ( e to exit)
MARKS_420 (y/n/e):
```

y

entering y, n and n for CLASS, RESULT and RES_SEC respectively would leave rel_table in the same state as dr! above, namely:

```
                        Relation Table: SCHOOL
Index  Name        Tsize  Ntuples  Arity  Domains
  3    _NULL         0       0       0
  4    MARKS_420    79      10       7    NAME STUID SEC A1
                                          A2 MID FIN
  5    CLASS        46       8       4    NAME STUID SEC FEES
```

In section c) we explain how to enter relational expressions. In chapter IV (System overview) we describe how these expressions are translated to some intermediate code for a stack machine. It may be useful for a programmer to display that code. This is done with po! (display the code generated: mnemonics and operands).

For example, to

RESULT <-[STUID, A1, A2]in MARKS_420;

corresponds the following piece of code displayed after invoking:

po!

```
 0:  PUSH_REL   RESULT
 2:  PUSH_REL   MARKS_420
 4:  PUSH_DOM   STUID
 6:  PUSH_DOM   A1
 8:  PUSH_DOM   A2
10:  PUSH       3
12:  PROJECT
13:  ASSIGN
14:  HALT
```

Observe that the code is a collection of integers. Some of them are indices in the domain or relation table. However, we display the corresponding domain or relation names.

In chapter VIII, we describe how errors are handled. In a nutshell: we attempt to keep on processing as much as possible unless a catastrophe occurs. You may impose that execution is to be stopped after errors have been detected and a predefined threshold of IO operations has been reached. This is done with batch! (switch mode to batch).

```
batch!
```

While running relix, you may still execute UNIX commands. Suppose you want to make changes to a copy of MARKS_420, named TEST, you can use either sh! (get a set of shell commands and execute) or the single line shell facility. The first one is invoked thus:

```
sh!

line consisting of 'el' terminates shell description

cp MARKS_420 TEST;

vi TEST

el
```

You return to relix when exiting from vi, after the copy has been made. For such a short sequence of commands, you may as well use the second possibility which is introduced by '%' and terminated by a carriage return, thus:

```
% cp MARKS_420 TEST; vi TEST
```

We will soon explain how to define domains or relations. Although you may always enter these definitions interactively, it may be convenient to collect them in a file, say GOOD_DEF, and run it from relix using input! (redirect standard input to a UNIX file).

```
input!

enter tty name (if unknown, exit and type: %tty)( 'e!' to exit):

tty10

enter file name ( or 'e!' to exit):

GOOD_DEF
```

assuming that tty10 is the outcome of "%tty".

Four system commands can be invoked with a single parameter. We present the relevant relix grammar rules (the complete grammar is in appendix A). Enclosed between angle brackets, '<' and '>', are syntactic categories.

```
<command-with-parameter>    ::= <command-name> '!!' <identifier>

<command-name>              ::= dr | pr | sd | sr

<identifier>                ::= <letter> ( <letter> | <digit> | '_')*

<digit>                     ::= 0 | 1 | ... | 9

<letter>                    ::= a | b | ... | z | A | B | ... | Z
```

where dr (delete a relation), pr (print a relation), sd and sr (display an entry in dom_table, respectively rel_table) have been discussed previously.

You can create new domains without using cd!. Similarly, you can declare a relation or turn a UNIX file into a relation without using cr!. The syntax is:

```
<domain-declaration>     ::=   domain <identifier> <type>

<type>                   ::=   boolean | bool | integer | intg

                         |     real | float

                         |     ( string | strg) <digit>+

<relation-declaration>   ::=   relation <identifier> <domain-list>

                               ( '<-' ( <identifier>

                               | <non_dc_dk_string>) | ϵ)

<domain-list>            ::=   <domain-list> ',' <domain-expression>

                         |     <domain-expression>

<non_dc_dk_string>       ::=   '"' [ ^ ( '"' | '' | 'O ) ] '"'
```

As in many grammars, ϵ denotes the empty string. The rule defining a string constant is read: starting and ending with ´"´, comprising no intermediate ´"´, ´\´ or carriage return. We could have obtained the same result as above with:

```
domain   NAME strg 26;

relation MARKS_420 ( NAME, STUID, SEC, A1, A2, MID, FIN)
                        <- "../MARKS";

relation VIEW_OF_420 ( NAME, STUID, SEC);
```

These relation declarations specify the attributes on which the relations are to be defined. As well, the first declaration indicates which UNIX file is to be associated with MARKS_420. In this example, the file MARKS, found in a sibling directory of the database, will be copied under the name MARKS_420. Hence, no modification to the latter can affect the former. The second declaration will not create a file associated with VIEW_OF_420. This option mat be used when defining views (see section c). Before moving on to domain definition, it is worth mentioning that starting up can follow a different course. The complete syntax is:

```
<start-up>            ::=   relix <options>
<options>             ::=   UNIX-path <other-options> | ε
<other-options>       ::=   <number-of-pages> <page-size-option> | ε
<number-of-pages>     ::=   <integer>
<page-size-option>    ::=   <integer> | ε
```

For example, suppose that you estimate that 100 pages of 2000 bytes each would be more convenient than the currently implemented default of 40 pages of 4096 bytes. Suppose also that you want to work in a database, say BANK, in a sibling directory of the current one. You may then enter:

relix ../BANK 100 2000

If you definitely do not like the default values but are also reluctant to type in relix ... 2000 repeatedly, we suggest you look up the UNIX aliasing facility.

b) DOMAIN ALGEBRA

Domain Table: SCHOOL

Index	Name	Length	Actual	Type
0	dom_name	20	T	STRG
1	rel_name	20	T	STRG
2	length	11	T	INTG
3	type	11	T	INTG
4	tuple_size	11	T	INTG
5	ntuples	11	T	INTG
6	count	11	T	INTG
7	dom_pos	11	T	INTG
8	sort_rank	11	T	INTG
9	NAME	26	T	STRG
10	STUID	7	T	STRG
11	SEC	2	T	STRG
13	A1	11	T	INTG
14	A2	11	T	INTG
15	MID	11	T	INTG
16	FIN	11	T	INTG
17	YEAR	4	T	STRG
18	CRED	11	T	INTG

Relation Table: SCHOOL

Index	Name	Tsize	Ntuples	Arity	Domains
3	_NULL	0	0	0	
4	MARKS_420	79	10	7	NAME STUID SEC A1 A2 MID FIN
5	CLASS	46	8	4	NAME STUID SEC FEES
6	DEPT	41	8	3	NAME YEAR CRED

New domains can be defined as the application of an operator or a function to previously defined domains, through the **let** statement (for the rest of this chapter, bold type is reserved for keywords, including **relix**):

let <identifier> **be** <domain-expression>

Four types of domain are available: **boolean** or **bool** (BOOL), **integer** or **intg** (INTG), **real** or **float** (REAL) and **string** or **strg** (STRG) (chain of characters). At present, no operations on real domains have been implemented further than the parsing phase.

We distinguish between horizontal and vertical domain expressions. The value of a horizontal domain expression for a given tuple depends only on values of domains within the same tuple. On the other hand, a vertical domain expression is a function of values from possibly more than one tuple.

We present the operators in order of increasing arity, starting with domains using no operators. The basic tokens are given by the following rules:

```
<boolean>    ::=   true | false | dc bool | dk bool
<integer>    ::=   <digit>+ | dc intg | dk intg
<real>       ::=   <digit>* '.' <digit>*
<string>     ::=   <non_dc_dk_string> | dc strg | dk strg
```

The maximum value of an integer constant is machine dependent. On a 32-bit machine typical values are: 2147483647 on the Masscomp and 2147418111 on the Cadmus. The symbols **dc**, for "don't care", and **dk**, for "don't know", represent null values. The first describes irrelevant information; the second, missing data. The length of any string must be between one and forty.

NO OPERATOR

A domain can be defined without using any operator, thus:

```
<domain-expression>   ::=   '(' <domain-expression> ')'
                        |   <constant>
                        |   <identifier>
<constant>            ::=   <boolean> | <integer> | <real>
```

Examples:

```
let ONE            be 1;
let TRUE           be true;
let DC_INT         be ( dc intg ),
let STUDENT_ID     be STUID;
```

The type of the expression following **be** determines the type of the new domain.

For example, ONE and DC_INT have type integer, TRUE type boolean and STUDENT_ID type string. The third example shows that superfluous parentheses can be used.

UNARY OPERATOR

Domains can also be defined in terms of unary operators:

```
<domain-expression>   ::=   <unary-op> <domain-expression>
                        |   <vertical-expression>

<vertical-expression>::=

            | red <ass-com-op>      of <domain-expression>
            | equiv <ass-com-op>    of <domain-expression>
                                    by <domain-list>
            | fun <fcn-par-op>      of <domain-expression>
                                    order <domain-list>
            | par <fcn-par-op>      of <domain-expression>
                                    order <domain-list>
                                    by <domain-list>

<unary-op>        ::=   ´-´ | ´+´ | ´¯´
<fcn-par-op>      ::=   <ass-com-op> | <other-bin-op>
                  |     pred | succ
<ass-com-op>      ::=   ´+´ | ´*´ | ´&´ | ´|´| max | min
<other-bin-op>    ::=   ´-´ | ´/´ | mod | ´**´ | ´||´
<domain-list>     ::=   <domain-list> ´,´ <domain-expression>
                  |     <domain-expression>
```

The less obvious operators have the following meanings:

<table>
<tr><td>'|'</td><td>or</td><td>'&'</td><td>and</td></tr>
<tr><td>'¬'</td><td>boolean negation</td><td>'||'</td><td>string concatenation</td></tr>
<tr><td>mod</td><td>integer remainder</td><td>'**'</td><td>exponentiation</td></tr>
<tr><td>pred</td><td>predecessor</td><td>succ</td><td>successor</td></tr>
</table>

Vertical operators are obtained by following one of **red**(uction), **equiv**(alence), **fun**(ction) and **par**(tial function) by a binary operator.

The outcome must be independent of the ordering of the tuples. Hence, the binary operators allowed with **red**, **equiv** must be associative and commutative.

On the other hand, the operators following **fun** or **par** need not be associative or commutative since an ordering of the tuples is specified by the **order** clause. The **by** and **order** lists may not be empty and, in the case of **par**, the **order** list must precede the **by** list.

Examples:

```
let CLASS_FEES    be red     + of FEES;
let SEC_FEES      be equiv   + of FEES by SEC;
let CUM_CRED      be fun     + of CRED order YEAR;
let CUM_CRED_N    be par     + of CRED order YEAR by NAME;
```

CLASS_FEES performs the addition of FEES over every tuple to produce a single result, whereas SEC_FEES partitions the tuples among the different equivalence classes, here the sections, and then computes a total for each section. If there is only one student, CUM_CRED computes, for each year, the number of credits accumulated as of the first year during which the student has completed some credits. The result would differ from one tuple to another as long as they differ in the YEAR attribute. Were there more than one student, CUM_CRED_N would achieve the same goal since it would partition the tuples by NAME before computing the cumulative sums.

BINARY OPERATOR

```
<domain-expression>::=

                <domain-expression> <ass-com-op> <domain-expression> |

                <domain-expression> <other-bin-op> <domain-expression> |

                <domain-expression> <comp-op> <domain-expression>

<comp-op>::= ´<´ | ´>´ | ´<=´ | ´>=´ | ´=´ | ´¯ =´
```

Precedence is given by the following table along with the rules:

-operators of lower precedence first

-operators on a given line have same precedence

-associativity is specified

left	associative	´|´ ´&´
non	"	´<´ ´>´ ´<=´ ´>=´ ´=´ ´¯ =´
left	"	´+´ ´-´
left	"	´*´ ´/´ mod
right	"	´**´
non	"	´¯ ´

Examples:

```
let   TOT   be ( A1 + A2 ) * 7 / 10 + MID + FIN;
let   A     be TOT >= 85;
let   B     be TOT >= 70 & TOT <= 84;
let   C     be TOT >= 55 & TOT <= 69;
```

The marking scheme is: 35 for the assignments, 15 for the midterm and 50 for the final. The result of TOT will be truncated to the nearest lower integer. Although truncation is avoidable, because it is not relevant in this case, this issue has not been dealt with at this time.

A, B and C have been defined so that we can assign grades according to the following table:

A	85 - 100
B	70 - 84
C	55 - 69
F	0 - 54

TERNARY OPERATOR

The single ternary operator provided is:

```
<domain-expression>  ::=  if <domain-expression>
                          then <domain-expression>
                          else <domain-expression>
```

The **else** clause may not be left out. The domains in the then, **else** parts must have the same type. The domain expression in the **if** part must have type boolean.

```
let GRADE be   if A then "A"
               else if B then "B"
               else if C then "C"
               else "F";
```

GRADE would assign to a student a grade according to the table given above.

FUNCTION APPLIED TO DOMAIN

New domains can be obtained by applying a function to existing ones:

```
<domain-expression>::= <function-name> '(' <domain-expression> ')'
          <function-name>    ::=    abs | cos | isknown | log10 | log2
                                |   ln | sin | tan
```

Most of them yield a domain of type real and, hence, are not fully implemented. The first one in the list above, **abs**, gives the familiar absolute value. The third one, **isknown**, has type boolean. It allows to check whether a given domain takes on value **dk**. Let us enter

let KNOWN_GRADE **be isknown**(GRADE);

this domain, be it actualized, will take on value TRUE wherever GRADE is different from dk, FALSE everywhere else.

It is worth pointing out that any domain has been defined in terms of actual domains, that is, domains already present in the database like A1 and MID, or constant domains, like 85 and "A", or previously defined domains.

The current implementation does not fully allow cyclic domains: that is, a domain redefined in terms of itself. That is, entering:

let A1 **be** A1 + A1;

would not cause any syntax or semantic error messages. However, it could not be actualized in any relation. The next section will explain why as well as describe how and when virtual domains are actualized. A virtual domain may be the result of a sequence of virtual domain definitions. At the end of the sequence, a virtual domain must be expressed in terms of actual or constant domains only. With the restriction on cyclic domains just mentioned, it is easy to see that any domain is the root of an expression tree where the leaves are the constant or actual domains or the operators.

c) RELATIONAL ALGEBRA

Just as new domains can be expressed in terms of already defined ones, new relations can be described as the result of applying zero or one operator, either unary or binary, to already defined relations. The occurrence of such a definition is called a statement. The evaluation mode allows us to distinguish between executable statement: immediate evaluation, and view statement: deferred evaluation.

```
<statement>::= <executable-statement> | <view-statement>

<executable-statement>::=
            <identifier> '<-' <relational-expression>                  (1)
          | <identifier> '<+' <relational-expression>                  (2)
          | <identifier> '[' <domain-list> '<-' <domain-list> ']'
            <relational-expression>                                    (3)
          | <identifier> '[' <domain-list> '<+' <domain-list> ']'
            <relational-expression>                                    (4)
```

Executable statements are characterized by an assignment sign:

(1) direct
(2) incremental
(3) renaming direct
(4) renaming incremental

The rules to build relational expressions are given hereafter, starting with the no operator case.

1) NO OPERATORS

```
<relational-expression>  ::=  '(' <relational-expression> ')'
                          |   <identifier>
```

Examples:

```
NEW_CLASS <- CLASS;
```

The name, here NEW_CLASS, need not be new. Had it been in use then the corresponding NEW_CLASS file would be overwritten. In no case is the relation CLASS or the corresponding CLASS file modified. Rather, they are copied under new names.

This an example of direct assignment. We will see many others hereafter. Three other types of assignment are available: incremental, renaming direct and renaming incremental. Examples are:

```
CLASS           <+ OTHER_CLASS;                       ( 5)
GRADE_305_R     [ STUID, GRADE_305
                <- STUID, GRADE] GRADE_305_R;         ( 6)
BIG_CLASS       [ STUDENT_ID, SECTION, TUITION
                <+ STUID, SEC, FEES ] CLASS;          ( 7)
```

The first one results in adding to CLASS the tuples of OTHER_CLASS. Had CLASS been a new name, then this would have been equivalent to a direct assignment. In any case, the domains of R in R <+ S must form a subset of the domains of S. S is projected over the domains of R, the result appended to R and the duplicate tuples eliminated. For example, suppose

OTHER_CLASS

NAME	STUID	SEC	FEES	COURSE
bonnallie, andre	8234187	2	152	PL/1
lucien, nicolas	8423861	3	321	pascal
brady, vivian	8230267	1	145	cobol

The result of (5) is:

CLASS

NAME	STUID	SEC	FEES
arrau, antonina	8192214	1	155
berard, paulette	8314201	3	233
bonnallie, andre	8234187	2	152
brady, vivian	8230267	1	145
christos, marilou	8215291	2	322
giroux, aline	8314626	1	112
hart, terry	8317112	1	378
jones, raymond	8215174	2	163
king, tam	8328521	3	244
lamontagne, paul	7913295	2	288
lucien, nicolas	8423861	3	321
rivet, maurice	8214512	3	364

Observe: the attribute COURSE has been eliminated as well as the duplicate tuple:

brady, vivian 8230267 1 145

In the renaming direct assignment two domain lists are specified. As for the join, not yet described in this section, both lists must have the same length and the domains must be compatible. It is noteworthy that (6) could have been used to rename GRADE instead of using the domain algebra.

The renaming incremental assignment combines the two previous. The relation CLASS is first projected on STUID, SEC and FEES, these being renamed as indicated in (7). Depending on whether BIG_CLASS is new or not, then a direct or incremental assignment is performed.

ii) UNARY OPERATORS

<relational-expression>::=

<project-clause> <where-clause> in <relational-expression>

```
<project-clause>    ::= '[' <domain-option> ']' | ε          ( 8)
<where-clause>      ::= where <domain-expression> | ε        ( 9)
<domain-option>     ::= <domain-list> | ε
```

Let R be the relational expression. We call (8) the project operator and (9) the select operator. Between the square brackets, '[' and ']', is a possibly empty list of domain names (as opposed to domain expressions) on which R is defined or which are actualizable in R. Clearly, a domain, say D, is actualizable in R if the leaves of the expression tree rooted at D are either constant domains or domains on which R is defined. Project is tantamount to stripping off the non mentioned domains and eliminating the duplicate tuples that this could have generated. A null empty relation is obtained when the list is empty. The domain expression, say D, after **where** must be boolean and actualizable in R. Only the tuples evaluating to true for D participate in the result. Remember that MARKS_420 contains the following tuples:

	NAME					
rivet, maurice	8214512	3	16	21	9	41
berard, paulette	8314201	3	23	21	11	40
jones, raymond	8215174	2	13	17	7	30
brady, vivian	8230267	1	11	17	8	44
giroux, aline	8314626	1	20	16	12	46
hart, terry	8317112	1	12	11	8	25
arrau, antonina	8192214	1	18	20	9	42
king, tam	8328521	3	17	22	12	36
christos, marilou	8215291	2	13	19	11	38
lamontagne, paul	7913295	2	20	20	11	43

and that we defined the following domains:

```
let TOT      be ( A1 + A2 ) * 7 / 10 + MID + FIN;
let A        be TOT >= 85;
let B        be TOT >= 70 & TOT <= 84;
let C        be TOT >= 55 & TOT <= 69;
let GRADE    be if A then "A"
             else if B then "B"
             else if C then "C"
             else "F";
```

We define a new relation, GRADE_420_R, thus:

```
GRADE_420_R <- [ NAME, TOT, GRADE] in MARKS_420;
```

We can invoke pr! to display the result

```
            GRADE_420_R

NAME                 TOT    GRADE
arrau, antonina       77      B
berard, paulette      81      A
brady, vivian         71      B
christos, marilou     71      B
giroux, aline         83      A
hart, terry           49      F
jones, raymond        58      C
king, tam             75      B
lamontagne, paul      82      A
rivet, maurice        75      B
```

Observe that the domains A, B and C, although actualized in order to evaluate GRADE, do not appear in the result since they were not mentioned in the domain list which defined GRADE_420_R. Indeed, using virtual domains, for example in the domain list of a project, is a way to cause their actualization. However, a domain already present in a relation is not reevaluated, even if redefined through a let statement. This is why we mentioned in the previous section that cyclicity is not supported.

Similarly, consider:

CLASS

NAME	STUID	SEC	FEES
arrau, antonina	8192214	1	155
berard, paulette	8314201	3	233
brady, vivian	8230267	1	145
christos, marilou	8215291	2	322
giroux, aline	8314626	1	112
hart, terry	8317112	1	378
jones, raymond	8215174	2	163
king, tam	8328521	3	244
lamontagne, paul	7913295	2	288
rivet, maurice	8214512	3	364

```
let CLASS_FEES  be red   + of FEES;

let SEC_FEES   be equiv + of FEES by   SEC;

CLASS_FEES_R <- [ CLASS_FEES ] in CLASS;

SEC_FEES_R   <- [ SEC, SEC_FEES ] in CLASS;
```

CLASS_FEES_R

CLASS_FEES
2404

and

SEC_FEES_R

SEC	SEC_FEES
1	790
2	773
3	841

DEPT

NAME	YEAR	CRED
brady, vivian	1984	13
brady, vivian	1985	15
jones, raymond	1983	16
jones, raymond	1984	15
jones, raymond	1985	16
rivet, michel	1982	15
rivet, michel	1983	12

```
let CUM_CRED_N  be par   + of CRED order YEAR by NAME;

DEPT_1 <- [ NAME, YEAR, CRED, CUM_CRED_N] in DEPT;
```

DEPT_1

NAME	YEAR	CRED	CUM_CRED_N
brady, vivian	1984	13	13
brady, vivian	1985	15	28
jones, raymond	1983	16	16
jones, raymond	1984	15	31
jones, raymond	1985	16	47
rivet, michel	1982	15	15
rivet, michel	1983	12	27
rivet, michel	1984	14	41

The following illustrates the select operator.

```
GOOD <- where TOT >= 75 in GRADE_420_R;
```

GOOD

NAME	TOT	GRADE
arrau, antonina	77	B
berard, paulette	81	A
giroux, aline	83	A
king, tam	75	B
lamontagne, paul	82	A
rivet, maurice	75	B

iii) BINARY OPERATORS

The unary operators depicted above generally trim down a relation horizontally (select) or vertically (project) or both. On the other hand, relations can be built up using join; that is, the result may be defined on more domains or contain more tuples or both.

```
<relational-expression> ::=
                              <relational-expression>
          '[' <domain-option> <join-op>  <domain-option> ']'
                              <relational-expression>

      <join-op>                ::=    <mu-join-op> | <sigma-join-op>
      <mu-join-op>             ::=    ijoin | natjoin | ujoin | sjoin
                               |      ljoin | rjoin | drjoin | djoin
                               |      dljoin
      <sigma-join-op>          ::=    <basic-sigma-join-op>
                               |      <negation> <basic-sigma-join-op>
                               |      icomp | natcomp
      <basic-sigma-join-op>    ::=    eqjoin | ltjoin | lejoin | sub
                               |      gtjoin | gejoin | sup | div
                               |      sep | iejoin
```

The following rules apply:

-the domain lists must have the same number of elements and, hence, they may be both empty;

-domains at the same position in both lists must be join-compatible, that is, have the same type (boolean, integer or string); moreover, if they are of type string, they must have the same length;

-if the lists are empty, the join is performed on all the domains common to both relational expressions;

-if there is no common domain special cases are considered (for example, an **ijoin** would result in a cartesian product).

The σ-join is not implemented further than the parsing phase. In our examples, we will use **ijoin**, the intersection or natural join, and **dljoin**, the left difference join.

Consider the relation MARKS_305 to be defined on the same domains as MARKS_420 and that actualizing GRADE in the former yields GRADE_305_R.

GRADE_305_R

NAME	STUID	GRADE
arrau, antonina	8192214	B
berard, paulette	8314201	C
brady, vivian	8230267	B
giroux, aline	8314626	C
king, tam	8328521	F
lamontagne, paul	7913295	A
rivet, maurice	8214512	B

We want to join on STUID to obtain the marks of the students who completed both courses. Before doing so, we must rename GRADE since a domain name can appear only once in a relation. Hence,

```
let GRADE_305 be GRADE;

let GRADE_420 be GRADE;

          GRADE_305_420 <-   ( [ NAME, STUID, GRADE_420] in GRADE_420_R)

                             [ STUID ijoin STUID ]

                             ( [ STUID, GRADE_305] in GRADE_305_R),
```

would produce

GRADE_305_420

NAME	STUID	GRADE_420	GRADE_305
arrau, antonina	8192214	B	B
berard, paulette	8314201	A	C
brady, vivian	8230267	B	B
giroux, aline	8314626	A	C
king, tam	8328521	B	F
lamontagne, paul	7913295	A	A
rivet, maurice	8214512	B	B

whereas

```
ONLY_420 <-   [ NAME, STUID] in
              ( GRADE_420_R [ STUID dljoin STUID] GRADE_305_R);
```

produces

ONLY_420

NAME	STUID
christos, marilou	8215291
hart, terry	8317112
jones, raymond	8215174

Notice that using virtual domains in the domain lists of a join is another way to induce their actualization.

VIEW STATEMENT

The assignment statements presented above are said to be executable because, when entered, they trigger an immediate evaluation of the relational expression. Nevertheless, it is possible to enter such expressions with deferred evaluation. This is called view definition and has the form

```
<view-statement>::=   <identifier>
                      ( initial <relational-expression> | ϵ)
                      is <relational-expression>
```

For a given relational expression, relix generates almost the same intermediate code whether the statement in which it occurs is executable or not. When it is a view, the interpreter refrains from executing the code (see chapters VI and VII). The code for views is kept in the code array for further use. Notice that view definitions are lost between relix sessions and that during a given session the code is evaluated each time the evaluation process is triggered.

The deferred evaluation mechanism is particularly handy when the user wants to define relations in terms of each other. Suppose that R and S are existing relations and that T and U must be defined as

```
T is R [ ujoin ] U;
U is T [ ujoin ] S;
```

had we entered '<-' instead of **is**, a run time error would have occurred when the evaluation of the first statement was attempted, since U was then still undefined.

As mentioned in chapter II, computing the transitive closure of a graph is another problem which can be solved using views. Recall that PARENT is a relation defined on SENIOR and JUNIOR which are domains of type string and length 18. Let "edward IV elizabeth of york " be a tuple of PARENT to indicate that edward IV is a parent of elizabeth of york.

PARENT

SENIOR	JUNIOR
edward IV	elizabeth of york
elizabeth of york	henry VIII
elizabeth of york	margaret
henry VII	henry VIII
henry VII	margaret
henry VIII	edward VI
henry VIII	elizabeth I
henry VIII	mary I
james IV stewart	james V stewart
james V stewart	mary stewart
margaret	james V stewart

In order to find for any two persons whether one is a descendant of the other, we compute the transitive closure of PARENT and call the result ANCESTOR. We have the following definitions:

```
relation ANCESTOR ( SENIOR, JUNIOR);
let SR1 be SENIOR;
let JR1 be JUNIOR;
```

Here are two ways, among many, in which we can use **relix** to compute ANCESTOR:

```
ANCESTOR    is PARENT [ ujoin ] [ SENIOR, JUNIOR] in (
            ([ SENIOR, JR1] in ANCESTOR)
            [JR1 ijoin SR1] ([ SR1, JUNIOR] in ANCESTOR));

ANCESTOR    initial PARENT
            is ANCESTOR [ ujoin ] [ SENIOR, JUNIOR] in (
            ([ SENIOR, JR1] in ANCESTOR)
            [JR1 ijoin SR1] ([ SR1, JUNIOR] in ANCESTOR));
```

Notice the declaration of ANCESTOR as being defined on SENIOR and JUNIOR. This is mandatory for recursively defined relations because the attributes of such a view can not be determined by the parser. That is, it would be assumed that the view is attributeless.

An alternative is to use the initial option as illustrated by the second example. This option involves only base relations. The corresponding relational expression would be evaluated and produce a, possibly empty, list of attributes when the interpreter needs to know the attributes on which the view is defined. We stress that it is not an error to use attributeless relations, as long as all the other rules are followed.

The code corresponding to the relational expression introduced by the keyword **initial** is executed only once by the interpreter whenever the view has to be evaluated.

To trigger the evaluation of a view like ANCESTOR, one has only to use it in a relational expression within an executable statement or enter

```
pril ANCESTOR
```

We show again the contents of ANCESTOR after its evaluation.

ANCESTOR

SENIOR	JUNIOR
edward IV	edward VI
edward IV	elizabeth I
edward IV	elizabeth of york
edward IV	henry VIII
edward IV	james V stewart
edward IV	margaret
edward IV	mary I
edward IV	mary stewart
elizabeth of york	edward VI
elizabeth of york	elizabeth I
elizabeth of york	henry VIII
elizabeth of york	james V stewart
elizabeth of york	margaret
elizabeth of york	mary I
elizabeth of york	mary stewart
henry VII	edward VI
henry VII	elizabeth I
henry VII	henry VIII
henry VII	james V stewart
henry VII	margaret
henry VII	mary I
henry VII	mary stewart
henry VIII	edward VI
henry VIII	elizabeth I
henry VIII	mary I
james IV stewart	james V stewart
james IV stewart	mary stewart
james V stewart	mary stewart
margaret	james V stewart
margaret	mary stewart

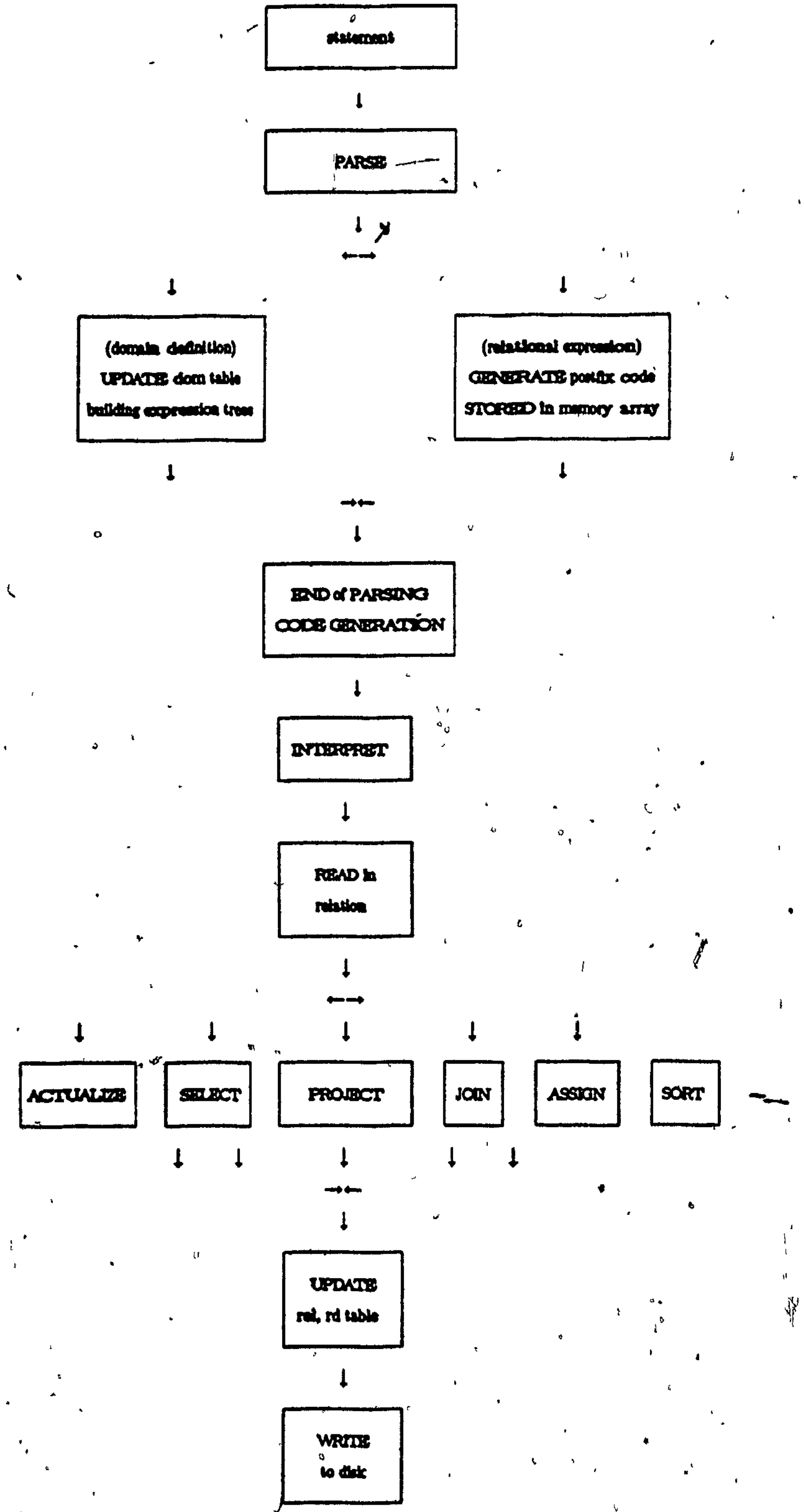
chapter IV

IMPLEMENTATION OF relix

Relix is interactive in that it accepts and executes one statement at a time. It comprises two main modules: a parser, generated by a UNIX program called yacc [JOHN75], performs type checking on the statement and generates some intermediate code; an interpreter executes this code. Relix internals can be divided between the data dictionary and the internal storage for relations. An overview of the system is presented on the next two pages. Each module is explained in the forthcoming pages. This chapter comprises the following sections:

a) System Overview
b) Data Dictionary
c) Relation Storage
d) Sorting
e) Parser / Code Generator
f) Interpreter
g) Programmer's manual

a) SYSTEM OVERVIEW



SORT is used by ACTUALIZE, PROJECT and JOIN.

(page left intentionally blank)

SYMBOLIC CONSTANTS

The following symbolic constants are used:

name	*value*	
NULL	0	
FALSE	0	
TRUE	1	
NEGATIVE	-1	
MAX_REL	40	maximum number of relations in SCHOOL
MAX_DOM	70	maximum number of domains in SCHOOL
MAX_ID	20	maximum length of an identifier
BOOL_LEN	1	width of a boolean domain in bytes
INTG_LEN	11	same for an integer domain
REAL_LEN	12	same for a real domain
STRG_LEN	40	maximum width of a string domain
BUFFER_SIZE	512	size of input buffer in bytes
MAXINT	2147483647	biggest integer available on Masscomp
	2147418111	same for the Cadmus
DT_BOOLEAN	257	boolean domain
DT_INTEGER	258	integer domain
DT_REAL	259	real domain
DT_STRING	260	string domain

The last four are mnemonics the value of which is set by yacc. We use string as an abbreviation for sequence of ASCII characters.

FILE SYSTEM

Let us present some considerations about names which are used to identify domains, relations, databases etc.

A name is a character string beginning with a letter and comprising only letters, digits or underscores; unless otherwise specified, letters may be lower case or upper case.

Due to UNIX limitations, database names must be no longer than fourteen bytes; relation names must be unique in the first fourteen bytes. Otherwise, names may comprise as many as twenty characters. A database is a directory containing two types of file.

The first type is the file associated with any relation. For example, to a relation named abcde1234567890 corresponds the file abcde123456789. Again, the name has been truncated to its fourteen first bytes; however, relix records the full name and would not understand any abbreviation. It is noteworthy that even the system relations: DOM, REL and RD are treated the same way as any other relation. They are described hereafter.

The second type has a single occurrence per database. It is called TRACE and contains user entered statements as well as messages issued by relix, mostly about errors and their severity. This file can be used for many purposes: it allows us to print a relation with some editing; it produces a trace of a work session and supplies enough documentation to report on any unexpected flaws in *relix*.

So, to the database SCHOOL corresponds a UNIX directory with the same name. This directory contains the files: DOM, REL, RD, TRACE plus one file for each relation in the database. Hence, relations in different databases can share a given name without interfering with each other.

b) DATA DICTIONARY

name	*type*	*size*
dom_table	array of records	MAX_DOM
rel_table	"	MAX_REL
rd_table	"	MAX_REL * MAX_DOM

DOM_TABLE

Information about domains is kept in the array dom_table. Each element of this array is a record comprising the following fields:

name	*type*	
name	string	user defined identifier, say D
length	integer	width of D in bytes
actual	"	flag indicating whether D is virtual or not
type	"	one of DT_BOOLEAN, DT_INTEGER, DT_REAL or DT_STRING
opnd1, opnd2	"	indexes of domains in terms of which is defined a virtual domain, say VD
operator	"	code of operator to be applied to operands when evaluating VD
by_list	array of integer	determine the equivalence classes to evaluate VD
order_list	"	determine the ordering of tuples to evaluate VD.

The relation DOM is defined on the attributes name, length and type. Only these, from dom_table, are stored in DOM. The tuples of DOM correspond to the domains of SCHOOL that have been used by at least one relation at one point. It is left to the user to eliminate those which are not used any more (see system command dd!).

The arrays by_list and order_list have variable size. The first entry contains the number of other entries.

The main operations defined on dom_table are:

```
read_dom_table()
  action:   fill dom_table from the file DOM
  return:   TRUE iff no problems to read in DOM

write_dom_table()
  action:   write dom_table to the file DOM
  return:   TRUE iff no problems to write out DOM

show_dom_table()
  action:   display dom_table on the screen

search_dom_table( NAME)
  input:    NAME
  type:     string
  return:   index of domain NAME, NEGATIVE if not found

insert_dom_table( NAME)
  input:    NAME
  type:     string
  action:   insert the domain NAME in the first
            empty location of dom_table
            -initialize all other entries to default values
  return:   index of entry where NAME has been inserted
            NEGATIVE if dom_table is full

dom_table_delete( D)
  input:    D
  type:     integer
  action:   flag domain D as deleted
            (name starting with null byte)
```

Information from dom_table is obtained through function calls. These functions take as argument an index, say D, in dom_table and return an integer unless otherwise specified.

```
is_deleted_dom( D)
  return:   TRUE iff D has been deleted

is_intermediate_dom( D)
  return:   TRUE iff D name starts with a digit
```

is_legal_dom(D)
return: TRUE iff D is a valid index in dom_table
(it is valid if positive and smaller than
the number of domains in dom_table)

is_constant_dom(D)
return: TRUE iff D is constant

dom_name(D)
return: name of D
type: string

dom_length(D)
return: length of D

dom_actual(D)
return: TRUE iff D is not virtual

dom_type(D)
return: type of D

dom_opnd1(D)
return: index of first operand of D

dom_opnd2(D)
return: index of second operand of D

dom_operator(D)
return: numeric value of operator of D

dom_by_list(D)
return: list of domains in the by list of D
type: array of integers

dom_order_list(D)
return: list of domains in the order list of D
type: array of integers

Entries in dom_table can be modified through procedure calls. One of their arguments is an index, say D, in dom_table. Unless otherwise specified their second argument, NEW, is an integer. They return no values.

change_length(D, NEW)
change_actual(D, NEW)
change_type(D, NEW)
change_opnd1(D, NEW)
change_opnd2(D, NEW)
change_operator(D, NEW)
action: as indicated by the name, replace by NEW length of D (respectively actual, type, opnd1, opnd2 or operator)

change_domname(D, NEW)
type: string
action: replace name of D by NEW

change_by_list(D, NEW)
type: linked list of integers
action: replace by_list of D by NEW

change_order_list(D, NEW)
type: linked list of integers
action: replace order_list of D by NEW

REL_TABLE

Information about relations is kept in the array rel_table. Each element of this array is a record comprising the following fields:

name	*type*	
name	string	user defined identifier, say R
tuple_size	integer	number of bytes making up a tuple of R
ntuples	"	number of tuples currently in R
start	"	base address of code to evaluate a view
domlist	list of integers	domains on which a relation is defined
sortlist	"	domains on which a relation is sorted
defined_on	"	relations on which a view is defined
defines	"	relations defined by a view

The relation REL is defined on the attributes name, tuple_size and ntuples. Only these, from rel_table, are saved in REL. Note that domlist and sortlist are saved, in normalized form, in RD (see further). The operations defined on these linked lists (domlist, sortlist, defined_on and defines) are detailed in section e. The main operations defined on rel_table are:

read_rel_table()
action: fill rel_table from the file REL
return: TRUE iff no problems to read in REL

write_rel_table()
action: write rel_table to the file REL
return: TRUE iff no problems to write out REL

show_rel_table()
action: display rel_table on the screen

search_rel_table(NAME)
input: NAME
type: string
return: index of relation NAME, NEGATIVE if not found

prefix_search_rel_table(NAME)
input: NAME
type: string
return: index of first entry such that its name has
NAME as prefix, NEGATIVE if no such entry

insert_rel_table(NAME)
input: NAME
type: string
action: insert the relation NAME in the
first empty location of rel_table
-initialize all other entries to default values
return: index of entry where NAME has
been inserted, NEGATIVE if rel_table is full

rel_table_delete(R)
input: R
type: integer
action: flag R as deleted (name starting with null byte)
remove the corresponding file

Information from rel_table is obtained through function calls. These functions take as argument an index, say R, in rel_table and return an integer unless otherwise specified. Section d, on sorting, presents a discussion of alias relation (name beginning with ´$´).

is_deleted_rel(R)
return: TRUE iff R has been deleted

is_temp_rel(R)
return: TRUE iff R is a temporary relation
(its name begins with ´$´ or ´mt.´)

is_legal_rel(R)
return: TRUE iff R is a valid index in rel_table
(it is valid if positive and smaller than
the number of relations in rel_table)

is_a_view(R)
return: TRUE iff R start is not NEGATIVE

is_alias_rel(R)
return: TRUE iff R name begins with ´$´

rel_name(R)
return: name of R
type: string

rel_tuple_size(R)
return: size of a tuple of R

rel_ntuples(R)
return: number of tuples in R

rel_arity(R)
return: number of domains on which R is defined

rel_sorted(R)
return: number of domains on which R is sorted

rel_start(R)
return: base address of code to evaluate the view R

rel_domlist(R)
return: list of domains on which R is defined
type: linked list of integers

rel_sortlist(R)
return: list of domains on which R is sorted
type: linked list of integers

rel_defines(R)
return: list of relations defined by view R
type: linked list of integers

rel_defined_on(R)
return: list of relations on which view R is defined
type: linked list of integers

Entries in rel_table can be modified through procedure calls. One of their arguments is an index, say R, in rel_table. Unless otherwise specified the second argument, NEW, is an integer. They return no values.

change_relname(R, NEW)
type: string
action: replace name of R by NEW

change_tuple_size(R, NEW)
change_ntuples(R, NEW)
change_start(R, NEW)
action: replace by NEW tuple size of R
(respectively ntuples and start)

```
change_domlist(       R, NEW)
change_sortlist(      R, NEW)
change_defines(       R, NEW)
change_defined_on(    R, NEW)
 type:      linked list of integers
 action:    replace domlist of R by NEW
            (respectively sortlist, defines or defined_on)
```

RD_TABLE

The remaining links between domains and relations are found in rd_table. To each domain used in a relation and each relation in which that domain is used corresponds an entry with the following fields:

name	*type*	
count	integer	number of different values of a domain in a relation
dom_pos	"	index of first byte of a domain in a relation
sort_rank	"	position of domain in sortlist

The relation RD is defined on the following attributes: relation and domain name, count, dom_pos and sort_rank. We just saw that to a given relation is associated a domlist (respectively sortlist) stored in rel_table. At the beginning of a session we build domlist (respectively sortlist) from dom_pos (respectively sort_rank) in RD. Symmetrically, at the end of the session we convert the information in domlist to a set of normalized tuples of RD. The main operations defined on rd_table are:

read_rd_table()
  action: fill rd_table from the file RD
  return: TRUE iff no problems to read in RD

write_rd_table()
  action: write rd_table to the file RD
  return: TRUE iff no problems to write out RD

show_rd_table()
  action: display rd_table on the screen

Information from rd_table is obtained through function calls. These functions take as argument two indices, say R and D, in rd_table and return an integer.

rd_count(R, D)
  return: number of different values taken
          by domain D in relation R

```
rd_dom_pos( R, D)
  return:    index of first byte of domain D in relation R

rd_sort_rank( R, D)
  return:    position of domain in sortlist
```

Entries in rd_table can be modified through procedure calls. Two of their arguments are indices, R and D, in rd_table. The third argument, NEW, is an integer. They return no values.

```
change_count(        R, D, NEW)
change_dom_pos(      R, D, NEW)
change_sort_rank(    R, D, NEW)
  action:    replace count ( respectively dom_pos,
             sort_rank) of D in R by NEW
```

c) RELATION STORAGE

Relations are stored on disk as pure character strings. At start up time, memory is set aside to store relations. System calls are issued to obtain num_pages of page_size consecutive bytes. These parameters may be changed by the user (see user's manual). Suppose they have been changed to 10 and 1024 respectively. To each group of page_size consecutive bytes corresponds a record, called page and defined below. Relix uses, aside from the global variables num_pages and page_size, the following data structures in order to manage the internal storage for relations (indenting is used to indicate the fields of a record):

name	*type*	*size*
frozen	array of booleans	MAX_REL

Each entry indicates whether the corresponding relation is an operand of the relational operation currently executed. Typically, operands and results must be frozen so as to avoid the preemption of the pages they occupy in memory.

page	array of records	num_pages
base	pointer	
rel_index	integer	
next	"	

To each physical memory page corresponds a page record. Base is pointing to the first byte of the (physical) page. Rel_index is the index in rel_table of the relation stored in this page, if any. Next is the index of the next page used to store other tuples of the same relation, if need be.

icrel	array of records	num_pages
page_index	array of integers	num_pages
rel_index	integer	
offset	"	
tuples_per_page	"	

A relation, say R, may occupy many pages in memory. A record named icrel (in core relation) is used to group those pages. Each non-negative entry of page_index contains the index of one of the pages used to store R. The index of R in rel_table is contained in rel_index. Offset indicates how many bytes separate the beginning of two consecutive tuples in a same page. The number of tuples of R that can fit in a single page is given by tuples_per_page.

icrel_for_rel   array of integers   MAX_REL

Each entry indicates whether the corresponding relation is presently in memory. If so, it also indicates in which icrel.

free_queue	record
front	integer
last	"

The free pages are kept in a linked list. The next field of the pages constitutes the link. Front contains the index of the first page in the queue. Last contains the index of the last one.

first_used_queue	record	
front	integer	index of front cell
last	integer	index of cell after rear
queue	array of integers	

A queue, called first_used_queue, is also used to keep track of pages that currently contain relations or have not yet been returned to free_queue. However, their next field is already used to complete the related icrel. Hence, the queue is implemented as a circular one. In other words, we merely store the indexes of the pages in use on a least recently claimed basis. Note that last is pointing one cell after the actual rear one so as to easily distinguish between an empty queue and a full one. Hence, the queue must have room for an extra entry and addition is done modulo num_pages, typical features of circular queues. On frozen, three operations are defined (R is the index of a relation):

freeze(R)
 action: set frozen[R] to TRUE

unfreeze(R)
 action: set frozen[R] to FALSE

is_frozen(R)
 return: TRUE iff frozen[R] is set to TRUE

For each relation R, icrel_for_rel[R] may be in either of two states:

NEGATIVE iff R is not presently in memory

positive, that is, the index of the icrel containing R.

On an icrel the operations are (R is an index in rel_table, I an icrel number; other argument when needed is also an integer).

icrel_line(I, i)
return: pointer to tuple number i of relation in icrel I

icrel_get(R, size)
action: get the pages needed for as many tuples as in R,
each comprising size bytes;
index of first page, say ICREL, is the icrel number

-set icrel[ICREL]. offset to size

-set icrel[ICREL]. tuples_per_page
to page_size / offset

-set icrel[ICREL]. page_index entries
to page indexes

-link the pages together

-enqueue the pages on first_used_queue

return: ICREL

icrel_free(R, I)
action: set all entries of icrel I
and those of corresponding pages to NEGATIVE

icrel_fill(R, I)
action: read the relation R from disk
into the icrel numbered I

icrel_flush(R, I, ntuples)
action: write up to ntuples from icrel numbered I to the
file associated with R, skipping those starting
with a null byte

icrel_show(R, I, ntuples)
action: display on the screen relation R,
contained in icrel I

When a relation is brought into memory, the new line character is replaced by a null byte. Hence, for each tuple we need tuple_size + 1 bytes. A page can accommodate a relation only if ntuples * (tuple_size + 1) <= 1024.

After having been created or freed, pages are kept in free_queue defined above.

```
empty_free_queue()
  return:     TRUE iff free_queue is empty

get_front_free_queue()
  return:     dequeued front page in free_queue

add_free_queue( BLOCK)
  type:       integer
  action:     enqueue page BLOCK
```

The operations on first_used_queue are as follows (their argument, where needed, is an integer):

```
enqueue_first_used( BLOCK)
  action:     add page BLOCK to the end of first_used_queue

dequeue_first_used()
  return:     front page of first_used_queue

move_end_first_used_queue( R)
  action:     move from front to the end of first_used_queue
              the chain of pages pertaining to the icrel
              containing relation R

front_first_used_queue()
  return:     front page of first_used_queue
              (does not remove it from queue)

free_front_first_used_queue()
  action:     transfer from first_used_queue to free_queue
              all the pages forming the front icrel

              -set icrel_for_rel[] of corresponding
              relation to NEGATIVE

              -free corresponding icrel

empty_first_used_queue()
  return:     TRUE iff first_used_queue is empty
```

We defined the following routine to handle the transfer of pages between free_queue and first_used_queue. It seems worthwhile to give the algorithm underlying that function.

```
get_one_page()

 if free_queue is not empty
   -return the front page of free_queue

 else
   garbage collection:
     -check whether in first_used_queue there are pages
      that are not pointed to by an entry of icrel_for_rel[]

     -if so, free the related icrel and chain of pages

   if garbage collection has been successful
     -return the front page of free_queue

   else
     -check whether all of the pages in first_used_queue
      belongs to frozen relations

     -if so
        -abort execution (no memory available)
      else
        -free the first not frozen icrel and chain of pages

        -return the front page of free_queue
```

Example:

Let R be a relation, of index 7 in rel_table, with 43 tuples, each tuple comprising 51 characters (the tuple size is fixed for any given relation). On disk, each group of 51 bytes is followed by a new line character. To get enough pages it suffices to issue the following function call:

I= icrel_get(7, 43).

This function determines that 1024 / 52 = 19 tuples can fit into a page and, so, that ceiling(43 / 19), that is, 3 pages are necessary. It then gets three free such pages, links them together and returns the number of the corresponding icrel which is always the index of the first page.

Assume that relation S, of index 11, is frozen and that it is stored in pages 4, 9 and 5. Assume also that relations U, index 3, and V, index 8, are stored in pages 6, 0, 2 and 1, 8 respectively. However, they are not frozen. Hence, we have:

relation	*index*	*icrel_for_rel*[]	*page*(s)
R	7	-1	
S	11	4	4, 9, 5
U	3	6	6, 0, 2
V	8	1	1, 8

	0	..	10	11	12	..	MAX_REL - 1
frozen:	F	..	F	T	F	..	F

page	*rel_index*	*next*	*base*	(address of storage base)
0	3	2	20000	
1	8	8	21024	
2	3	-1	22048	
3	-1	7	23072	note that consecutive
4	11	9	24096	
5	11	-1	25120	bases are 1024
6	3	0	26144	
7	-1	-1	27168	bytes apart
8	8	-1	28192	
9	11	5	29216	

free_queue . front= 3

. last = 7

first_used_queue . front= 5

. last = 2

. queue

0	1	2	3	4	5	6	7	8	9	10
1	8	?	?	?	4	9	5	6	0	2

In order that icrel_get() work properly, relations must be frozen beforehand. A typical sequence of instructions is:

```
if (I= icrel_for_rel[ R]) is NEGATIVE
  freeze( R);
  I= icrel_get( R, tuple size of R);.
  icrel_fill( R, I);

perform some operations on R in I;
icrel_flush( R, I, some number of tuples);
unfreeze( R);
```

Notice that pages 3, 7 and 6 are returned. Other structures are in the following state (naturally, bases are unchanged and, hence, not repeated):

relation	*index*	*icrel_for_rel[]*	*page*(s)
R	7	3	3, 7, 6
S	11	4	4, 9, 5
U	3	-1	
V	8	1	1, 8

	0	..	10	11	12	..	MAX_REL - 1
frozen:	F	..	F	T	F	..	F

page	*rel_index*	*next*
0	-1	2
1	8	8
2	-1	-1
3	7	7
4	11	9
5	11	-1
6	7	-1
7	7	6
8	8	-1
9	11	5

```
icrel[ 3]   . offset= 52

            . tuples_per_page = 19

            . rel_index= 7

            . page_index[ 0] = 3

            . page_index[ 1] = 7

            . page_index[ 2] = 6

            . page_index[ 3] = -1
                  .
                  .
                  .
            . page_index[ 9] = -1

      free_queue    . front= 0

                    . last = 2

  first_used_queue    . front= 0

                      . last = 7

                      . queue
```

0	1	2	3	4	5	6	7	8	9	10
1	8	3	7	4	9	5	6	?	?	?

d) SORTING

Many operations of the domain and relational algebra have been implemented in a way that requires that relations be sorted. We call the alias of a relation, say R, another relation that is identical to R up to the sort order.

Our sort procedure sets up the environment and then calls a standard quicksort routine to do the actual sorting. The procedure is invoked thus:

sort(ptr_R, ptr_I, sortlist)

name	*type*	*description*
ptr_R	pointer to integer	pointer to index in rel_table of relation R
ptr_I	"	pointer to index of the icrel containing R
sortlist	linked list of integers	ordered list of domains on which R must be sorted

The calling routine needs to know two things: the index in rel_table of the alias, say S, of the original relation sorted on sortlist and generated by sort; the number of the icrel, say J, where sort stored S. Since a function can return only one value, we use pointers, thus emulating Pascal call by variable, to pass back the information.

The tuples are preprocessed so as to extract the keys which are kept in a new icrel. For each tuple of R a pair of pointers is set and placed in an array called ptrs, each element comprising:

lptr	points to	a tuple of R
kptr	"	the key extracted from that tuple

Sorting requires only that we swap pairs of pointers when tuples are out of order, as opposed to swapping character strings. The lptr pointers are then used to put the original data in sorted order.

The algorithm goes as follows:

```
sort( ptr_R, ptr_I, sortlist)

 if  rel_sortlist( R) is a prefix of sortlist
   return

 make an alias, say S, corresponding to R and sortlist
 if S is presently in RAM, say in icrel J
   in the calling routine replace R by S, I by J;
   (this is why parameters are passed through pointers)
   return

 else
   ( 1) allocate as many ptrs as R has tuples

   ( 2) using icrel_get(), obtain a new icrel, say J,
          to store the result

   ( 3) for each integer domain in sortlist
          -replace any negative value by its 0's complement
           (so that integers are sorted properly)

   ( 4) for each tuple, i, of R
          -set ptrs[ i]. lptr to icrel_line( I, i)

          -set ptrs[ i]. kptr to sort key extracted from
            relation R in I and stored in icrel J

   ( 5) undo ( 3)

   ( 6) apply quicksort() on ptrs

   ( 7) for each tuple i of R
          copy ptrs[ i]. lptr to icrel_line( J, i)

   ( 8) free the space occupied by ptrs

   ( 9) R <- S

   (10) I <- J
```

It is worth mentioning that while we extract the keys, step (4), we also check whether the tuples are already in order and set a flag. At step (6), we inspect first this flag and, hence, avoid sorting an already sorted set of tuples.

Let us illustrate some of these steps through an example. Consider a relation named ACCOUNT, of index 13, defined on the following attributes

name	*index*	*type*	*length*	*comments*
NAME	33	STRG	10	customer's name
BALANCE	9	INTG	11	current balance
LOAN	14	INTG	11	amount of loan

Assume also: ACCOUNT is sorted on NAME and LOAN, contains 6 tuples and is stored in icrel 7. Let sortlist be BALANCE and NAME. Let the tuples of ACCOUNT be as follows:

NAME	LOAN	BALANCE
andrew	8800	90
charles	470	330
jeff	2350	500
jerome	0	-50
mark	100	800
steve	1500	-700

Let us consider the case where the alias, of index 19, is not currently residing in RAM but will be stored in icrel 5.

Relation Table

Index	Name	Tsize	Ntuples	Arity	Domains
3	_NULL	0	0	0	
13	ACCOUNT	32	6	3	NAME LOAN BALANCE
19	$13*9*33	32	6	3	NAME LOAN BALANCE

The name $13*9*33 contains the following information: 13, the index of the relation of which it is an alias; 9 and 33, index of the domains on which the alias is sorted, that is BALANCE (9) and then NAME (33) within BALANCE.

ACCOUNT, of index 13, is found in icrel 7:

tuple #	address		
0	27168	andrew	0000000880000000000090
1	27201	charles	0000000047000000000330
2	27234	jeff	0000000235000000000500
3	27267	jerome	00000000000-0000000050
4	27300	mark	0000000010000000000800
5	27333	steve	00000001500-0000000700

After step (4), the extracted keys are in icrel 5. The pointers lptr and kptr have been set.

tuple #	ptrs.lptr	ptrs.kptr
0	27168	25120
1	27201	25142
2	27234	25164
3	27267	25186
4	27300	25208
5	27333	25230

The contents of icrel 5 is:

tuple #	address	
0	25120	00000000090andre
1	25142	00000000330charles
2	25164	00000000500jeff
3	25186	-9999999949jerome
4	25208	00000000800mark
5	25230	-9999999299steve

After step (6), we would have:

tuple #	ptrs.lptr	ptrs.kptr
0	27333	25230
1	27267	25186
2	27168	25120
3	27201	25142
4	27234	25164
5	27300	25208

This permutation of the lines of icrel 5 yields the keys, and hence the tuples, in order.

old tuple #	address	
5	25230	-9999999299steve
3	25186	-9999999949jerome
0	25120	00000000090andre
1	25142	00000000330charles
2	25164	00000000500jeff
4	25208	00000000800mark

After step (7) the relation is sorted as required:

tuple #	address		
0	25120	steve	00000001500-0000000700
1	25142	jerome	00000000000-0000000050
2	25164	andrew	0000008800000000000090
3	25186	charles	0000000047000000000330
4	25208	jeff	0000000235000000000500
5	25230	mark	0000000010000000000800

e) PARSER / CODE GENERATOR

The statements entered by the user fall into two broad categories: domain definition and executable statement (strictly speaking, view statements constitute a third one, but they differ only slightly from the latter). For each statement of the first category, relix builds an expression tree in dom_table (see example below). For a statement of the second, it generates code. When compiling, it is generally easier to produce intermediate code, as opposed to machine or assembly code. Postfix code is such an intermediate code. It is characterized by the operator appearing after all of its operands in an expression. For example,

A + B * C (infix) is read A B C * + (postfix)

This code is useful because a stack may be used to evaluate it. Hence, the relix interpreter is a stack-based machine. In order to generate such code, the relix parser needs to store the operands temporarily.

OVERVIEW

```
file with description of tokens --> lex --> tokenizer (lex.yy.c)

                                        \
     file with grammar rules            |
                                        |
     file with actions                  | --> yacc --> y.tab.c (parsing tables)
                                        |
     file lex.yy.c                      |
                                        /

y.tab.c --> cc --> a.out (parser)
```

In our case the actions are grouped in a single C-function, named the semantic_analyser. This function comprises many small blocks of code. Whenever it is invoked, exactly one block is executed. Hence this function is a mere big switch statement. Lex is a UNIX program to generate a lexical analyser [LESK75].

HOW TO USE YACC

With an editor, say vi, create or modify a file (say source.y) that contains (see example included):

-description of symbolic tokens

-specification of associativity rules for non unary operators

-specification of precedence among operators

-complete set of grammar rules, complete in the sense that
all non-terminals are defined

-calls to user's routines which perform semantic analysis,
type checking and code generation

In general, we would enter the following UNIX commands in order to produce a parser:

```
yacc    source.y    creates a file named    y.tab.c
cc      y.tab.c              "              a.out that is executable
```

However, it is more convenient to use the UNIX make facility and simply enter:

```
make  (which executes the list of commands in Makefile)
```

In order to perform its various tasks, the parser use the following data structures:

name	*type*	
memory	array of integers	storage for generated code
WORD	integer	points into memory one location after the last one filled
operator_stack	"	(see below)
rd_stack	"	temporary storage for domain or relation indices

The operator_stack is used to store the flags indicating that a project has been seen or the indexes of the domains on which select operations are to be performed. Note that operator_stack and rd_stack are both of type integer_stack. No operations are defined on memory or WORD, its associated cursor; access is performed directly, by array indexing

name	*type*
domain_list_stack	array of index_list

An occurrence of the type index_list is a linked list of records. Each record, called an index_element, has two fields:

index	integer	domain or relation indexes
next	pointer	link to next index_element

The header node contains the number of other elements in the list. Construction or modification of index_list instances is done through function calls (unless otherwise specified, parameters are linked lists):

build_list(I)
type: array of integers
action: from I build an index_list
return: pointer to the head of the new index_list

cat_list(I_1, I_2)
type: array of integers
action: LIST_1== build_list(I_1); LIST_2== build_list(I_2)
removing header node , append LIST_2 to LIST_1;
update header node of LIST_1
return: LIST_1

copy_index_list(LIST)
action: traverse and copy the index_list pointed to by LIST
return: pointer to the head of the new index_list

index_list_allocate()
return: a new index_element record
(its index and next fields set to NULL)

index_list_append(LIST, INDEX)
action: add an index_element, its index field set to INDEX,
to the end of LIST (duplicates not allowed) .

index_list_change(LIST, INDEX_1, INDEX_2)
type: integer
action: in LIST replace INDEX_1 by INDEX_2

index_list_equal(LIST_1, LIST_2)
return: TRUE iff these two lists are identical: any given
index field value appearing in one list must appear
in the other list; moreover, index field values
must appear in the same order in both lists

index_list_free(LIST),
action: collect storage pointed to by LIST for latter use

index_list_member(LIST, INDEX)
return: TRUE iff INDEX is a member of LIST

index_list_prefix(LIST_1, LIST_2)
return: TRUE iff LIST_1 is a prefix of LIST_2

index_list_prepend(LIST, INDEX)
action: add an index_element, its index field set to INDEX,
to the front of LIST

nocheck_index_list_append(LIST, INDEX)
action: add an index_element, its index field set to INDEX,
to the end of LIST (do not eliminate duplicates)

pop_index_list()
action: -pop the run time stack into NDOM which then
contains the number of domains making up
a domain list;
-pop the run time stack NDOM times and build an
index_list using index_list_prepend (so that the
order, reversed by stacking process, is restored)
return: pointer to the head of the thus built index_list

remove_from_index_list(LIST, INDEX_1)
action: find and remove the node with
index field value equal to INDEX_1

trim_first_list_if_longer(LIST_A, LIST_B)
action: if LIST_A is longer than LIST_B remove enough
trailing nodes from the former so that they have
the same number of elements

The stacks comprise the fields given below. Standard stack routines (push, pop, top and is_empty) are supplied.

integer_stack		
. element	integer	flag or domain or relation index
. TOP	"	points to the last filled entry of integer_stack
domain_list_stack		
. element	pointer	to linked list of integers
. TOP	integer	points to the last filled entry of domain_list_stack

SAMPLE GRAMMAR (as a yacc source)

```
%{
#include    <stdio.h>     /* system library */
#include    <math.h>      /* system library */
#include    "lc.h"        /* All the defines */
#include    "ls.h"        /* The structures. */
#include    "le.h"        /* The externals.
#include    "ll.h"        /* The Lex externs */
%}

%token      ASSIGN IDENTIFIER LET BE RED OF IN
%left       '+' '-'
%left       '*' '/' '%'
%nonassoc   '¯'
%start      program
%%

program:

( 1)        statement

              { semantic_analyser( PROG_1); return( TRUE);}
    ;

statement:

( 2)        definition_statement ';'
    |
( 3)        executable_statement ';'
    ;

definition_statement:

( 4a)        LET IDENTIFIER

               {semantic_analyser( LET);}

( 4b)        BE domain_expression

               {semantic_analyser( BE);}
     ;
```

```
domain_expression:

( 5)        domain_expression '/' domain_expression

              {semantic_analyser( DIVIDE_HORIZONTAL);}
         |
( 6)        RED '+' OF domain_expression

              {semantic_analyser( RED_PLUS);}
         |
( 7)        '(' domain_expression ')'
         |
( 8)        domain_id
         ;

domain_id:

( 9)        IDENTIFIER

              {semantic_analyser( DOMAIN_ID);}
         ;

domain_list:

( 10)        domain_list ',' IDENTIFIER

               {semantic_analyser( DOMAIN_LIST);}
          |
( 11)        IDENTIFIER

               {semantic_analyser( DOMAIN_LIST);}
          ;

executable_statement:

( 12)        relation_id_left ASSIGN relational_expression

               {semantic_analyser( ASSIGN);}
          ;

relation_id_left:

( 13)        IDENTIFIER

               {semantic_analyser( RELATION_ID_LEFT);}
          ;
```

```
relational_expression:

( 14a)        project_list where_clause IN

                {semantic_analyser( IN);}

( 14b)        relational_expression

                {semantic_analyser( PROJECT);}
           |
( 15)         IDENTIFIER

                {semantic_analyser( RELATION_ID_RIGHT);}
           ;

project_list:

( 16)         /* null project */
           |
( 17a)        '['

                {semantic_analyser( PROJECT_ON);}

( 17b)        domain_option ']'

                {semantic_analyser( PROJECT_OFF);}
           ;

where_clause:

( 18)         /* null select */
           |
( 19a)        WHERE

                {semantic_analyser( WHERE);}

( 19b)        domain_expression
           ;

domain_option:

( 20)         /* empty */
           |
( 21)         domain_list
           ;
```

Assume that the database TEST is in the following state (from now on, the attributes Length and Type need not be always mentioned):

Domain Table: TEST

Index	Name	Length	Actual	Type
0	dom_name	20	T	STRG
1	rel_name	20	T	STRG
2	length	11	T	INTG
3	type	11	T	INTG
4	tuple_size	11	T	INTG
5	ntuples	11	T	INTG
6	count	11	T	INTG
7	dom_pos	11	T	INTG
8	sort_rank	11	T	INTG
9	z	1	T	BOOL
10	x	11	T	INTG
11	c	12	T	STRG
12	v	11	T	STRG
13	a	11	T	INTG
14	b	11	T	INTG

Relation Table: TEST

Index	Name	Tsize	Ntuples	Arity	Domains
3	_NULL	0	0	0	
4	a	35	10	4	c v x z
5	b	45	17	5	a b v x z

and that we enter the following domain definition:

let d be (red + of a) / b ;

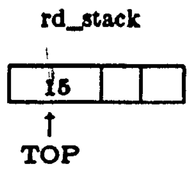
the actions triggered by the parser and performed by the semantic analyser are indicated below. We use '↑' to indicate the current token looked at by the parser.

1) let ↑ d rule (4a)

case LET:

-get index of d in dom_table (insert it if new): 15

-push index on rd_stack

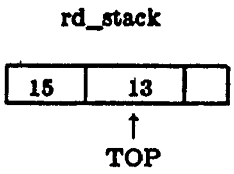


2) let d be (red + of ↑ a rule (9)

case DOMAIN_ID:

-get index of a: 13

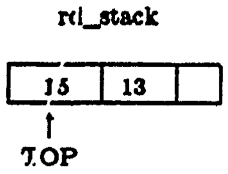
-push index on rd_stack



3) let d be (red + of a ↑) rule (6)

case RED_PLUS:

-pop rd_stack into D



-check that dom_type(D == 13) == DT_INTEGER

-make a descriptive name thus:
first operand, '[', operator code, ']', second operand

result_dom== update_dom_table(left_dom, code, NEGATIVE);

-insert it in dom_table and initialize appropriate entries

```
initialize_dom_table_entry( result_dom, length, type,
                            left_dom, NEGATIVE, code);
```

where result_dom== 16

length== INTG_LEN

type== DT_INTEGER

left_dom== 13

code== RED_PLUS

Domain Table: TEST

Index	Name	Actual	Opnd1	Opnd2	Operator
13	a	T			
14	b	T			
15	d	F			
16	13 [600]	F	13		600

observe:

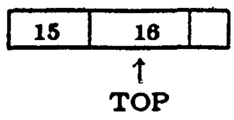
-600 is the value of the constant RED_PLUS

-second operand has been left blank

-(temporary) default values have been assumed for length, actual and type of d

-push on rd_stack the index of inserted domain: 16

rd_stack



4) let d be (red + of a ↑)   rule (7)

-nothing to do

5) let d be (red + of a) / ↑ b   rule (9)

case DOMAIN_ID:

-get index of b: 14

-push index on rd_stack

rd_stack

15	16	14
		↑ TOP

6) let d be (red + of a) / b ↑ ;   rule (5)

case DIVIDE_HORIZONTAL:

-pop rd_stack into right_dom

-pop rd_stack into left_dom

rd_stack

15	16	14
↑ TOP		

-check that dom_type(right_dom = 14)
and dom_type(left_dom = 16) are both numeric

-make a descriptive name, insert it in dom_table and initialize appropriate entries in dom_table to get

Domain Table: TEST

Index	Name	Actual	Opnd1	Opnd2	Operator
13	a	T			
14	b	T			
15	d	F			
16	13 [600]	F	13		600
17	16 [510] 14	F	16	14	510

observe that 510 is the value of
the constant DIVIDE_HORIZONTAL

-push on rd_stack the index of inserted domain

```
        rd_stack

   | 15 |  17  | 14 |
            ↑
           TOP
```

7) let d be (red.+ of a) / b ↑ ;   rule (4b)

case BE:

-pop rd_stack into I

-pop rd_stack into J

```
             rd_stack

         | 15 | 17 | 14 |

    ↑
   TOP
```

-if domain I = 15 is redeclared

-verify that its type is not changed

-if its type is STRG

-verify that its length is not changed

-set type of I to type of J = 17

-set length of I to length of J

-set operator of I to RENAME (2030)

Domain Table: TEST

Index	Name	Actual	Opnd1	Opnd2	Operator
13	a	T			
14	b	T			
15	d	F	17		2030
16	13 [600]	F	13		600
17	16 [510] 14	F	16	14	510

observe:

-length and type of d have been set properly

8) let d be (red + of a) / b ↑ ;   rule (2)

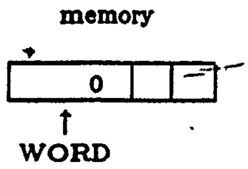
-nothing to do

9) let d be (red + of a) / b ↑ ;   rule (1)

case PROG_1:

-generate HALT (0)

-return TRUE



Our second example is a relational assignment

z <- [a, b, d] in b ;

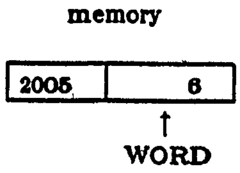
we consider all flags initially FALSE and all stacks empty.

1) z ↑ <-   rule (13)

case RELATION_ID_LEFT:

-find index of z in rel_table (insert it if new): 6

-generate PUSH_REL (2005) 6

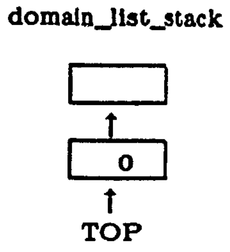


2) z <- ↑ [rule (17a)

case PROJECT_ON:

-set project_flag to TRUE

-push on domain_list_stack a header node



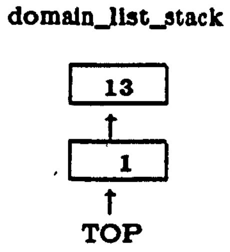
3) z <- [↑ a   rule (11)

case DOMAIN_LIST:

-get index of a in dom_table: 13

-if virtual, subtract from index 2 * MAX_DOM (70)

-append it to the list on TOP of domain_list_stack

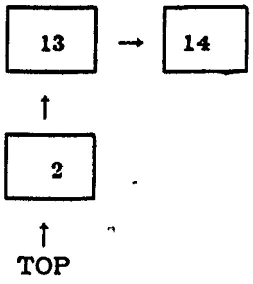


4) z <- [a, ↑ b rule (10)

case DOMAIN_LIST:

-same as previous

domain_list_stack



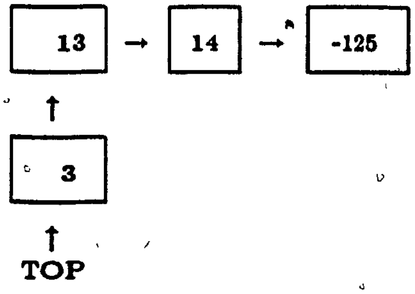
observe that the counter in the header node is incremented each time a new node is appended

5) z <- [a, b, ↑ d rule (10)

case DOMAIN_LIST:

-same again; note that the index is negative since d is virtual

domain_list_stack



6) z <- [a, b, d ↑] rule (21)

-nothing to do

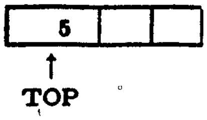
7) z <- [a, b, d ↑] rule (17b)

case PROJECT_OFF:

-push TRUE on operator_stack

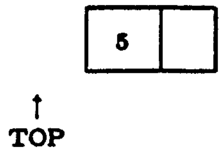
```
8)  z <- [ a, b, d] ↑ in        rule ( 18)

    -nothing to do

9)  z <- [ a, b, d] ↑ in        rule ( 14a)

    case IN:

    -if select_flag = TRUE

      -pop rd_stack into select_dom

     else

      -set select_dom to NEGATIVE

    -push select_dom = NEGATIVE on operator_stack

    -set project_flag to FALSE

10) z <- [ a, b, d] in ↑ b        rule ( 15)

    case RELATION_ID_RIGHT:

    -get index of b in rel_table: 5

    -push index on rd_stack
```

rd_stack



```
11) z <- [ a, b, d] in b ↑ ;        rule (14b)

    case PROJECT:

    -pop rd_stack into rel_id

    -if rel_id = 5 is not NEGATIVE
      (it would have been, had we had a relational expression
       instead of b, a relation name)

      -generate PUSH_REL rel_id
```

rd_stack



```
-pop operator_stack into select_dom

-pop operator_stack into project_flag

 (now select_dom == NEGATIVE and project_flag == TRUE)

-if select_dom == NEGATIVE

  -nothing to do (otherwise, we would generate
                  code for SELECT)

-if project_flag == TRUE
 (generate code for PROJECT; otherwise, nothing to do)

 -for each dom_id in the list at domain_list_stack. TOP

   -if dom_id is negative

     -add to it 2 * MAX_DOM

     -set a flag, say virtual_on

   -generate PUSH_DOM dom_id. PUSH_DOM (2000)

 -generate PUSH (2010) value of header node
  at domain_list_stack. TOP

 -if virtual_on is set

   -generate ACTUALIZE (2025)

   -reset virtual_on

   -for each dom_id in the list at domain_list_stack. TOP

     -generate PUSH_DOM dom_id

 -free the list at domain_list_stack. TOP; pop the stack

 -generate PROJECT (2020)

 -push NEGATIVE on rd_stack
```

memory

2005	6	2005	5	2000	13
2000	14	2000	17	2025	2010
3	2000	13	2000	14	2000
17	2010	3	2020		

↑
WORD

rd_stack

-1		

↑
TOP

```
12)  z <- [ a, b, d] in b ↑ ;      rule ( 12)

     case ASSIGN:

     -pop rd_stack into rel_id

     -rel_id is NEGATIVE

        -nothing to do

     -generate ASSIGN (287)

13)  z <- [ a, b, d] in b ↑ ;      rule (  3)

     -nothing to do

14)  z <- [ a, b, d] in b ↑ ;      rule ( 1)

     case PROG_1:

     -generate HALT

     -return TRUE
```

memory

2005	6	2005	5	2000	13
2000	14	2000	17	2025	2010
3	2000	13	2000	14	2000
17	2010	3	2020	0	

↑
WORD

rd_stack

-1	

↑
TOP

The code seems more meaningful when symbolic constants and names, as opposed to indices in dom_table or rel_table, are used

LOCATION	OPERATION	OPERAND
0:	PUSH_REL	d
2:	PUSH_REL	b
4:	PUSH_DOM	a
6:	PUSH_DOM	b
8:	PUSH_DOM	d
10:	PUSH	3
12:	ACTUALIZE	
13:	PUSH_DOM	a
15:	PUSH_DOM	b
17:	PUSH_DOM	d
19:	PUSH	3
21:	PROJECT	
22:	ASSIGN	
23:	HALT	

f) INTERPRETER

The interpreter reads the code generated by the parser and executes it, emulating a standard computer. It operates on the memory array, an instruction counter: relix_IC (an index in the memory array), the relation registers (icrel), the data dictionary and a run time stack (STACK). Only the last one is new. It is defined as a record comprising the following fields:

name	*type*
TOP, max	integer
el	array of integers

The classical stack operations are defined thus (S points to an occurrence of STACK, the other argument, when needed, is an integer):

stk_init(S, num_els)
action: allocate an array of num_els integers;
set max to num_els and TOP to NEGATIVE;

pop(S)
return: the top element unless TOP is NEGATIVE;
decrement TOP

push(S, element)
action: check if the stack is full (TOP = max);
if not, increment TOP and insert element in
that new position

The interpreter instruction set, internally a collection of integers, comprises the primitives given by the table below. The operators in the last five rows are referred to as the JOIN´s.

ACTUALIZE	ASSIGN	ASSIGN_OPTION	DEL_R
HALT	INCREMENT	INIT_VIEW	IS
PRINT_R	PROJECT	PUSH	PUSH_DOM
PUSH_REL	RENAME	RENAME_INCREMENT	SELECT
SHOW_D	SHOW_R		
DL_JOIN	DR_JOIN	EQ_JOIN	GE_JOIN
GT_JOIN	IE_JOIN	I_COMP	I_JOIN
LE_JOIN	LT_JOIN	L_JOIN	NEQ_JOIN
NGE_JOIN	NGT_JOIN	NLE_JOIN	NLT_JOIN
R_JOIN	S_JOIN	U_JOIN	

The interpreter

-fetches the instruction pointed to by relix_IC, which is incremented

-decodes the instruction

-if necessary
 -fetches operands from memory or
 -pops them from the STACK

-invokes subroutines to perform operations on relations

-pushes the result on the STACK when appropriate

Instructions are of variable length and thus the decoding / fetching phases overlap. We group the instructions by the number of operands that they pop from or push on the stack. In order to illustrate how each primitive works we consider various statements of the relational algebra. We show the contents of the memory array as filled in by the parser / code generator. We show also the contents of STACK before and after the execution of the operation. TOT and GRADE are the virtual domains defined in chapter III.

Domain Table: SCHOOL

Index	Name	Length	Actual	Type
9	NAME	26	T	STRG
10	STUID	7	T	STRG
12	A1	11	T	INTG
13	A2	11	T	INTG
16	TOT	11	F	INTG
17	GRADE	2	F	STRG
36	TOT_305	11	T	INTG
37	GRADE_305	2	T	STRG

Relation Table: SCHOOL

Index	Name	Tsize	Ntuples	Arity	Domains
4	MARKS_420	79	10	7	NAME STUID SEC A1 A2 MID FIN
13	GRADE_305_R	39	7	3	NAME TOT_305 GRADE_305

```
GRADE_420_R <- [ NAME, TOT, GRADE] in MARKS_420;

        0:   PUSH_REL    GRADE_420_R
        2:   PUSH_REL    MARKS_420
        4:   PUSH_DOM    NAME
        6:   PUSH_DOM    TOT
        8:   PUSH_DOM    GRADE
       10:   PUSH        3
       12:   ACTUALIZE
       13:   PUSH_DOM    NAME
       15:   PUSH_DOM    TOT
       17:   PUSH_DOM    GRADE
       19:   PUSH        3
       21:   PROJECT
       22:   ASSIGN
       23:   HALT
```

PUSH, PUSH_DOM and PUSH_REL

pop none, push 1 (fetching from memory)

```
relix_IC →   0:   PUSH_REL    GRADE_420_R
             2:   PUSH_REL    MARKS_420
```

□ (left corresponds to bottom of STACK)

↑
TOP

-push a constant on the STACK
 (respectively a domain, relation index)

-operand is in the next location of the memory array

The parser determines from the grammar rule recognized whether the current operand is a scalar or an index in the domain or relation table. Similarly, the subroutines of the interpreter take as arguments a combination of scalars and indices. They are built to interpret correctly the integers with which they are fed. In particular, we do not need three different PUSH primitives. However, the code is more readable this way. Assume that 6 is the index of GRADE_420_R in rel_table.

```
             0:  PUSH_REL   GRADE_420_R
            ___
relix_IC →   2:  PUSH_REL   MARKS_420
                  ┌───┐
                  │ 6 │
                  └───┘
                    ↑
                   TOP
```

PUSH and PUSH_DOM work similarly.

ACTUALIZE and PROJECT

pop variable number of operands, push 1

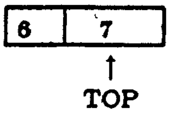
```
relix_IC →  12:  ACTUALIZE
            ___
            13:  PUSH_DOM   NAME

  ┌───┬───┬───┬────┬────┬────┐
  │ 6 │ 4 │ 9 │ 16 │ 17 │ 3  │
  └───┴───┴───┴────┴────┴────┘
                          ↑
                         TOP
```

the STACK contains a domain list and a relation index:
3 is the number of domains
9, 16 and 17 are the domain indices
4 is the relation index

-pop domain list into Dlist
-pop relation index into R
-S <- list_actualize(Dlist, R)
-push S on STACK

The functions, like list_actualize, that are called by the interpreter to implement different operations of the relational algebra will be explained in chapter VI. Assume that 7 is the index of the relation resulting from the actualization of 9, 16 and 17 in 4.

```
              12:   ACTUALIZE
              ___
relix_IC →    13:   PUSH_DOM    NAME
```

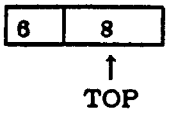


PROJECT works similarly with the difference that project(Dlist, R) is called. Assume that 8 is the index of the relation resulting when we project relation 7 on domains 9, 16 and 17.

ASSIGN, ASSIGN_OPTION, INIT_VIEW, INCREMENT and IS

pop 2 , push none

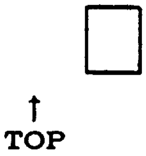
```
relix_IC →    22:   ASSIGN
              ___
              23:   HALT
```



the STACK contains two relation indexes: 8 and 6

-pop relation indexes into S and R
-assign(R, S), that is S to R

```
              22:   ASSIGN
              ___
relix_IC →    23:   HALT
```



The others work similarly. ASSIGN_OPTION, INIT_VIEW and IS invoke also assign(R, S). INCREMENT invokes either assign(R, S) or increment(R, S).

HALT, DEL_R, PRINT_R, SHOW_D and SHOW_R

pop none, push none

dr!! MARKS_420

```
relix_IC →  0:  DEL_R   MARKS_420

            3:  HALT
```

-invoke
  rel_table_delete(4)

-the others invoke respectively
  -print_one_rel(R)
  -show_dom_table(D)
  -show_rel_table(R)

```
            0:  DEL_R   MARKS_420

relix_IC →  3:  HALT
```

-no action is performed;
 control is returned to the main driving routine

-except in the case of HALT, the operand is in the next location of the memory array

-the STACK is not modified

SELECT

pop 2 , push 1

GOOD <- where TOT >= 75 in GRADE_420_R;

```
0:  PUSH_REL    GOOD

2:  PUSH_REL    GRADE_420_R

4:  PUSH_DOM    16[ 332 ]20

6:  SELECT

7:  ASSIGN

8:  HALT
```

```
relix_IC →   6:   SELECT
            ────
             7:   ASSIGN

          | 9 | 6 | 48 |
                     ↑
                    TOP
```

Domain Table: SCHOOL

Index	Name	Actual	Opnd1	Opnd2	Operator
16	TOT	T			
20	$75	F			258
48	16[332]20	F	16	20	332

the STACK contains a domain and a relation index:
48 is the domain and 6 the relation index

-pop domain index into D
-pop relation index into R
-S <- select(D, R)
-push S on STACK

Assume that 10 is the index of the relation resulting from the select operation.

```
             6:   SELECT
            ────
relix_IC →   7:   ASSIGN

          | 9 | 10 |
                ↑
               TOP
```

RENAME, RENAME_INCREMENT

pop variable number of operands , push none

TO_BE_POSTED [STUID, ASSIGNMENT_1, ASSIGNMENT_2

<- STUID, A1,A2] MARKS_420;

Relation Table: SCHOOL

Index	Name	Tsize	Ntuples	Arity	Domains
9	GOOD	39	6	3	NAME TOT GRADE
11	TO_BE_POSTED	29	10	3	STUID ASSIGNMENT_1 ASSIGNMENT_2
13	GRADE_305_R	39	7	3	NAME TOT_305 GRADE_305

Domain Table: SCHOOL

Index	Name	Length	Actual	Type
9	NAME	26	T	STRG
10	STUID	7	T	STRG
12	A1	11	T	INTG
13	A2	11	T	INTG
16	TOT	11	T	INTG
17	GRADE	2	T	STRG
49	ASSIGNMENT_1	11	T	INTG
50	ASSIGNMENT_2	11	T	INTG

```
 0:   PUSH_REL    TO_BE_POSTED
 2:   PUSH_DOM    STUID
 4:   PUSH_DOM    ASSIGNMENT_1
 6:   PUSH_DOM    ASSIGNMENT_2
 8:   PUSH        3
10:   PUSH_REL    MARKS_420
12:   PUSH_DOM    STUID
14:   PUSH_DOM    A1
16:   PUSH_DOM    A2
18:   PUSH        3
20:   RENAME
21:   HALT
```

```
relix_IC →  20:   RENAME
            21:   HALT
```

11	10	49	50	3	4	10	12	13	3

↑
TOP

the STACK contains two domain lists and two relation indexes:
- 10, 12 and 13 constitute the first domain list
- 10, 49 " 50 the second one
- 4 and 11 are the relation indexes

-pop domain list into DSlist
-pop relation index into S
-pop domain list into DRlist
-pop relation index into R
-rename_increment(R, DRlist, S, DSlist, opcode)
(where opcode is either RENAME or RENAME_INCREMENT)

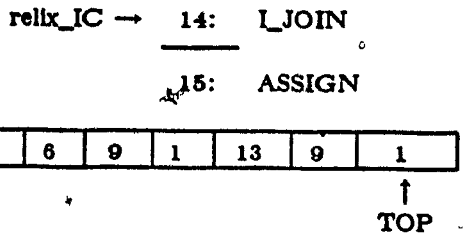
```
                20:  RENAME
                ___
relix_IC →      21:  HALT

                     □
                ↑
               TOP
```

JOIN

pop variable number of operands , push 1

GRADE_420_305 <- GRADE_420_R [NAME ijoin NAME] GRADE_305_R;

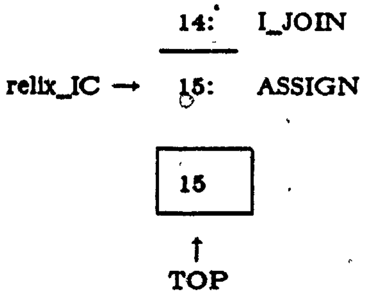
```
 0:  PUSH_REL   GRADE_420_305
___
 2:  PUSH_REL   GRADE_420_R
___
 4:  PUSH_DOM   NAME
___
 6:  PUSH       1
___
 8:  PUSH_REL   GRADE_305_R
___
10:  PUSH_DOM   NAME
___
12:  PUSH       1
___
14:  I_JOIN
___
15:  ASSIGN
___
16:  HALT
___
```



the STACK contains two domain lists and two relation indexes:
  -both domain lists comprise a single domain: 9
  -13 and 6 are the relation indexes

```
-pop domain list into DSlist
-pop relation index into S
-pop domain list into DRlist
-pop relation index into R
-T <- mu_join( R, DRlist, DSlist, S, mode)
-push T on STACK
```

where mode is any member of the family of JOIN's. The μ-join have been implemented and documented by Ann T. Chong; the σ-join remains to be implemented. Assume that 15 is the index of the result of the ljoin.



g) PROGRAMMER'S MANUAL

How one would go about adding new features to relix? We give here some guidelines in the form of two examples. The first details the steps to follow in order to develop redit, a relational editor. The second one, from the domain algebra, illustrates how we implemented the isknown function.

IMPLEMENTATION OF REDIT

In order to recognize redit, a new token, we add the following rule to ll.l, the lex source:

```
redit    {listing( PROGRAM_BUFFER, yytext); return( REDIT);}
```

and to ly.y, the yacc source, the declaration:

```
%token   REDIT
```

Suppose we want redit to obey the following syntax and semantic specifications.

syntax:

relational_expression::==

{ [domain-list] } redit { relational-expression } .

semantic:

-both [domain-list], arg-1, and relational-expression, arg-2,
are optional, but they may not be both omitted

-arg-1 specifies the sort order to be used by the editor;
if arg-2 is empty, it also supplies the domains on which
a new relation is to be defined

-arg-2 is evaluated to yield a relation, say of index R,
before invoking the editor; if arg-1 is left out
the domain list of R provides the sort order

-the elements of arg-1 must be names, not expressions

We add the following grammar rules to ly.y:

```
relational_expression:

    project_list REDIT

      {semantic_analyser( REDIT_1);}

    relational_expression_option

       {semantic_analyser( REDIT);}

relational_expression_option:

    /* empty */

      {semantic_analyser( REDIT_NULL);}

    | relational_expression

       {semantic_analyser( REDIT_NO_NULL);}
```

the rule for project_list has been given in section e. The semantic_analyser must be modified thus:

```
 case REDIT_1:

  -if arg-1 is present
   -create a new index_list and push it on the stack;
     push_domain_list_st ck( index_list_allocate());

 case REDIT_NULL:

  -push FALSE on  operator_stack:
    S_PUSH( operator_stack, FALSE);

 case REDIT_NO_NULL:

  -push TRUE on  operator_stack
```

```
case REDIT:

 -pop operator_stack into redit_flag

 -if redit_flag is FALSE
   -generate: PUSH_REL NEGATIVE

 -pop domain_list_stack into right_ptr:
   right_ptr= pop_domain_list_stack();

 -for each domain in the list right_ptr

   -check that it is a valid domain name

   -generate: PUSH_DOM domain index:
     generate_code( PUSH_DOM);
     generate_code( right_ptr-> index);

 -generate: REDIT
```

It remains to add the following code to the procedure interpreter() in iscc.c:

```
case REDIT:
  pop relation id and number of domains from run-time stack:
    domlist_R= pop_index_list();
    R= pop_and_eval_if_view( &run_time_stk);

  invoke the relational editor:
    S= redit( domlist_R, R);
    push( &run_time_stk, S);
    index_list_free( &domlist_R);

  break;
```

The function pop_and_eval_if_view() is explained in chapter VII. The programmer must now supply the body of the function redit() in order to perform the tasks of the editor.

function name: redit

arguments:

 domlist_R: (possibly empty) linked list of domain indices

 R: index (possibly NEGATIVE) of the relation to edit

return:

 S: index (possibly R) of the resulting relation

Let us present, together with the intermediate code generated, a few examples of relational expressions using redit.

```
[ FIN, STUID] redit MARKS_420

22:  PUSH_REL     MARKS_420
24:  PUSH_DOM     FIN
26:  PUSH_DOM     STUID
28:  PUSH         2
30:  REDIT
```

```
redit MARKS_420

14:  PUSH_REL     MARKS_420
16:  PUSH         0
18:  REDIT
```

```
[ FIN, STUID] redit

22:  PUSH_REL     UNKNOWN (NEGATIVE)
24:  PUSH_DOM     FIN
26:  PUSH_DOM     STUID
28:  PUSH         2
30:  REDIT
```

The next one would cause the following error message to be printed: domain list and relational expression can not be both empty.

```
                a <- redit;
        32:   PUSH_REL    a
        34:   PUSH_REL    UNKNOWN
        36:   PUSH        0
        38:   REDIT
        39:   ASSIGN
```

IMPLEMENTATION OF ISKNOWN

Likewise, in order to implement isknown we modify ll.l, ly.y and lz.c thus:

ll.l:

```
 isknown       {listing( PROGRAM_BUFFER, yytext);
              return( ISKNOWN);}
```

ly.y:

```
 %token    ISKNOWN

 domain_expression: ISKNOWN '(' domain_expression ')'

                    {semantic_analyser( ISKNOWN);}
```

lz.c:

```
 case ISKNOWN:

   left_dom== S_POP( rd_stack);
   type     == DT_BOOLEAN;
   length   == BOOL_LEN;
   result_dom== update_dom_table( left_dom, code, NEGATIVE);
   initialize_dom_table_entry( result_dom, length,
                        type, left_dom, NEGATIVE, code);
   S_PUSH( rd_stack, result_dom);
   break;
```

let KNOWN_ID be isknown(STUID);

would modify dom_table thus:

Domain Table: SCHOOL

Index	Name	Length	Actual	Type	Opnd1	Operator
10	STUID	7	T	STRG		
20	KNOWN_ID	1	F	BOOL	21	2030
21	10 [271]	1	F	BOOL	10	271

In is1_dom_op(o) (irdut.c) we replace

```
return(   is_vertical_op( o) || o == RENAME ||

          o == ABS || o == NOT || o == UNARY_PLUS ||

          o == UNARY_MINUS);
```

by

```
return(   is_vertical_op( o) || o == RENAME || o == ISKNOWN ||

          o == ABS || o == NOT || o == UNARY_PLUS ||

          o == UNARY_MINUS);
```

we modify tuple_actualize() (itact.c)
thus (* indicates additions):

```
 if( is1_dom_op( op)){
    if( is_vertical_op( op ))
      evaluate_vertical_dom( pos1, l1, op, ptr_R, D, ptr_I);
    else
      switch( op ){
      case NOT:
           .
           .
           .
*     case ISKNOWN:
*       test for DK and append T or F:
*         for( i = 0; i < rel_ntuples( R); i++){
*           strncpy( tuple, &icrel_line( I, i)[ pos1], l1);
*           tuple[ l1] = ' ';
*           strcat( icrel_line( I, i),
*                isknown( dom_type( dom_opnd1( D)), tuple));
*         }
           .
           .
           .
      } end of switch */
 } end of if( is1_dom_op( op)) */
```

In iarith.c we add the function isknown which returns a pointer to a string:

```
isknown( type, s)
 input:    type is an integer, s a string
 return:   result of comparing s with DK
           (of appropriate type)
```

s	x	DC	DK
	F	F	T

```
 switch( type){
 case DT_BOOLEAN:
  if( *s == *DK_BOOL_s)
    return( FALSE_s);
  else
    return( TRUE_s);

 similar case statements for the other types.
 }
```

CHAPTER V

DOMAIN ALGEBRA IMPLEMENTATION

In this chapter, we describe the algorithms to actualize a domain D for each tuple of a relation R contained in icrel I. The chapter comprises two sections:

a) Extraction of Domain Values

b) Operations on Domain Values

a) EXTRACTION OF DOMAIN VALUES

The domain D is defined as a function of some other domains. The routines described in this section locate these other domains in relation R, extract the corresponding values, invoke the functions described in the next section in order to compute the value of D from these operands and append the so obtained value for D to the appropriate tuples of R in icrel I.

TUPLE_ACTUALIZE

The procedure actualize, described in the following chapter, calls tuple_actualize. This one takes three parameters: D, the index of the domain to be actualized; ptr_R, a pointer to the index, R, of the relation in which D is ~~to be~~ actualized; ptr_I, a pointer to the icrel, I, containing R. As mentioned before, pointers are used since actualizing vertical domains defined through equiv, fun or par requires us to invoke our sort procedure described in the previous chapter.

Tuple_actualize determines whether D is a horizontal or a vertical domain. In the first case, it performs the work described below. In the second case, it invokes evaluate_vertical_dom.

```
tuple_actualize( D, ptr_R, ptr_I)
 input:  D        index of virtual domain
         ptr_R    pointer to relation index
         ptr_I    pointer to icrel
```

```
1- if D is a horizontal domain
 a- constant:
   -extract value of constant from dom_name( D)
   -append to each tuple

 b- unary:
   -find position and length of operand domain
   -extract value and store it in opnd1

   -select on dom_operator( D)

     case NOT:
       -append field swapping boolean value:
          strcat( LINE, negate_boolean( opnd1));

     case UNARY_PLUS or RENAME:
       -just copy attribute value:
          strcat( LINE, opnd1);

     case UNARY_MINUS:
       -append negation of operand field

     case ABS:
       -put '0' in first byte of opnd1
       -append to tuple

 b- binary:
   -find position and length of both operand domains
   -extract values and store them in opnd1 and
      opnd2 respectively

   -select on dom_operator( D)
      case EQ, NE, LT, LE, GT or GE:
       -compute and append boolean value using the
        following functions depending on the type of
        the operand domains:
          compare_integer() or compare_string()

     case CAT_HORIZONTAL:
       -append to tuple the concatenation of opnd1 and opnd2

     case INTEGER operator:
       -compute result using appropriate function,
        after conversion to integer:
            opnd1 to val1, opnd2 to val2:

       case PLUS_HORIZONTAL:
         result== add( val1, val2)

       case MINUS_HORIZONTAL:
         result== subtract( val1, val2)
      case TIMES_HORIZONTAL:
        result== multiply( val1, val2)

      case DIVIDE_HORIZONTAL:
        result== divide( val1, val2)

      case MODULO_HORIZONTAL:
        result== modulo( val1, val2)

      case MAX_HORIZONTAL:
        result== max( val1, val2)

      case MIN_HORIZONTAL:
        result== min( val1, val2)

      case EXP:
        result== power( val1, val2)

      -convert result to string and append to tuple

    case BOOLEAN operator:
      -compute and append boolean value:

      case AND_HORIZONTAL:
        strcat( LINE, and( opnd1, opnd2)

      case OR_HORIZONTAL:
        strcat( LINE, or( opnd1, opnd2)

c- ternary:
  -find position and length of the three operand domains

  -extract values and store them in opnd1,
     opnd2 and opnd3 respectively

  -select opnd1

    case TRUE:
      -append opnd2 to tuple

    case FALSE:
      -append opnd3 to tuple

    case DC:
      -append DC to tuple

    case DK:
      -append DK to tuple

  -if opnd2 and opnd3 have different length
     pad with blanks
```

```
2- if D is a vertical domain
     evaluate_vertical_dom( pos1, l1, operator,
                                             ptr_R, D, ptr_I);
```

EVALUATE_VERTICAL_DOM

The following, called by tuple_actualize to evaluate a vertical domain D, determines through which of red, equiv, fun and par is D defined. It makes sure that the relation is sorted correctly, as required by equiv, fun or par. It then invokes either reduction, for red and equiv, or function, for fun or par, to complete the actualization process.

```
evaluate_vertical_dom( pos1, l1, operator, ptr_R, D, ptr_I)
 input:   pos1 and l1    integer      position and length
                                      of operand
          operator          "         code of operator
                                      defining D
          D                 "         index in dom_table
          ptr_R          pointer to   relation index
          ptr_I             "         icrel

 -operator falls in one of the four groups

 RED_PLUS     EQUIV_PLUS     FUN_PLUS       PAR_PLUS
 RED_TIMES    EQUIV_TIMES    FUN_TIMES      PAR_TIMES
 RED_MAX      EQUIV_MAX      FUN_MAX        PAR_MAX
 RED_MIN      EQUIV_MIN      FUN_MIN        PAR_MIN
 RED_AND      EQUIV_AND      FUN_AND        PAR_AND
 RED_OR       EQUIV_OR       FUN_OR         PAR_OR
                             FUN_MINUS      PAR_MINUS
                             FUN_DIVIDE     PAR_DIVIDE
                             FUN_MODULO     PAR_MODULO
                             FUN_EXP        PAR_EXP
                             FUN_CAT        PAR_CAT
                             FUN_SUCC       PAR_SUCC
                             FUN_PRED       PAR_PRED

 -if operator belongs to the first group
    reduction( operator, pos1, l1, 0,
                      rel_ntuples( *ptr_R), *ptr_I);

 -if operator belongs to the second group
   -sort on by_list:
     sort( ptr_R, ptr_I, build_list( dom_by_list( D));

   -determine the strata or equivalence classes

   -for each stratum
     reduction( operator, pos1, l1, tfirst, tlast, *ptr_I);
     (tfirst== first tuple of stratum,
      tlast== last tuple of stratum)
```

```
-if operator belongs to the third group
 -sort according to the ordering attribute (on order_list)

 -apply
     function( operator, pos1, l1, 0, rel_ntuples( *ptr_R),
                          *ptr_I, domlist, *ptr_R);

-if operator belongs to the fourth group
 -sort to get both the strata and the proper ordering
     append order_list to by_list so as to get newlist

     sort( ptr_R, ptr_I,  newlist);

 -determine the strata or equivalence classes

 -for each equivalence class
     function( operator, pos1, l1, tfirst, tlast,
                          *ptr_I, domlist2, *ptr_R);
    (tfirst== first tuple of stratum,
     tlast== last tuple of stratum)
```

REDUCTION

reduction(operator, pos1, l1, t1, t2, I)

input:	pos1 and l1	integer	position and length of operand
	operator	"	code of operator defining D
	t1 and t2	"	first and last tuple of stratum
	I	"	index of icrel

```
-initialize accumulator, according to operation performed,
   with appropriate null value or identity element

case RED_PLUS:     accum= 0;

case RED_TIMES:    accum= 1;

case RED_MAX:      accum= - MAXINT;

case RED_MIN:      accum= MAXINT;

case RED_AND:      accum= TRUE;

case RED_OR:       accum= FALSE;


-for each tuple
  -extract opnd

  -compute accum= accum operator opnd

-append accum to each tuple
```

FUNCTION

function(operator, pos1, l1, t1, t2, I, domlist, R)

input:	pos1 and l1	integer	position and length of operand
	operator	"	code of operator defining D
	t1 and t2	"	first and last tuple of stratum
	R and I	"	index of relation and icrel
	D	"	domain index
	domlist	list	linked list of domain indices

```
-any operator except PRED and SUCC

 -initialize accumulator, according to operation performed,
    with appropriate null value or identity element
```

```
  -for each stratum
      (all tuples with same value for the by attributes)

    -from first tuple to last (backward if operator is EXP)

      -if current tuple differ from previous
          in ordering attribute

        -extract opnd

        -compute accum== opnd operator accum

      -append accum to current tuple

-operator PRED or SUCC

  -for each stratum

    -from first tuple to last

      -if current tuple differ from previous
          in ordering attribute

        -SUCC
          -append value to previous from current
              (first stratum follows the last one)

        -PRED
          -append value from previous to current
              (last stratum precedes the first one)
```

b) OPERATIONS ON DOMAIN VALUES

This section contains the functions which perform the operations on the values extracted from the relation R in which the domain D is actualized. They return the value of D in R. These functions help isolating the implementation of the domain algebra from the rest of relix.

We handle here the operations on the null values DC and DK. We follow the approach described in [MERR84a]. Remember that DC represents irrelevant information and means "don't care". DK describes missing data and means "don't know".

In this implementation each domain type is a set containing DC and DK represented thus:

	DC	DK
boolean	#	!
string	#	!
integer	-2147418111	-2147418112

These values have been chosen so that they are ordered the same way for all domain types: DK smaller than DC which is smaller than all other values in the domain type. The DC null value is taken to behave as a special value with properties similar to those of non-null values. With respect to operators the behavior of DC is best explained by the following tables. In a nutshell: it acts like an identity element for operations like +, x, max and min. In other words, it is ignored. The comparison $x = DC$ (respectively $x < DC$) has value true (respectively false) if x is DC and false (respectively DC) if x is non-null.

The DK null value is more troublesome. Conceptually, it is a variable ranging over all the non-null values of a domain type. That is, if an expression involves DK then all the non-null values of the same domain type are substituted for DK. If the result is always the same, this is the value of the expression. Otherwise, the expression has value DK. However, having chosen a special value from the domain type to represent it, we approximate the rule just described with three-valued logic. DK is seen as a third logical value, the other two being true and false, that a logical expression may take on. Any comparison between a non-null and DK has value DK. The tables below indicate how DK behave as an operand of the logical operators: and, or, negation. The result of any arithmetic operation on DK is DK.

Notice that this approach contains some inconsistencies: the tautology ((a < DK) or (a >= DK)) evaluates to DK; similar problem occurs if DK is replaced by DC in the preceding example. Our simplistic approach in handling DC and DK has the advantage of supplying the users with a way to experiment with null values. Thus, feedback from those may indicate better avenues to explore. Due to the high modularity of this implementation, changes need to be made only to some of the functions below in order to probe a different approach.

While executing the following functions, various errors may occur: arithmetic overflow (absolute value of result is not smaller than MAXINT), division by zero, invalid operand (for example, a boolean argument is none of true, false, DC or DK) and so on. In such a case, we signal an error of class SEVERE (see chapter VIII) and return DK of the appropriate domain type. In each table below, x is anything but DC or DK.

add(a, b)
input: a, b integers
return: sum of a and b

a\ b	x	DC	DK
x	a+b	a	DK
DC	b	DC	DK
DK	DK	DK	DK

multiply(a, b)
input: a, b integers
return: product of a and b

a\ b	x	DC	DK
x	a*b	a	DK
DC	b	DC	DK
DK	DK	DK	DK

subtract(a, b)
input: a, b integers
return: difference of a and b

a\ b	x	DC	DK
x	a-b	a	DK
DC	-b	DC	DK
DK	DK	DK	DK

divide(a, b)
input: a, b integers
return: quotient of a by b

a\ b	x	DC	DK
x	a/b	a	DK
DC	1/b	DC	DK
DK	DK	DK	DK

modulo(a, b)
input: a, b integers
return: remainder of the division of a by b

a\ b	x	DC	DK
x	a mod b	a	DK
DC	1 mod b	DC	DK
DK	DK	DK	DK

negate_integer(a)
input: a integer
return: product of a by -1

a	x	DC	DK
	-a	DC	DK

max(a, b)
  input: a, b integers
  return: biggest of a and b

a\ b	x	DC	DK
x	a max b	a	DK
DC	b	DC	DK
DK	DK	DK	DK

min(a, b)
  input: a, b integers
  return: smallest of a and b

a\ b	x	DC	DK
x	a min b	a	DK
DC	b	DC	DK
DK	DK	DK	DK

absolute(a)
  input: a integer
  return: absolute value of a

a	x	DC	DK
	|x|	DC	DK

negate_boolean(a)
  input: a boolean
  return: negation of a

a	x	DC	DK
	¯ x	DC	DK

power(a, b)
input: a, b integers
return: a raised to the power b

a\ b	x	DC	DK
x	a**b	a	DK
DC	DC	DC	DK
DK	DK	DK	DK

isknown(type, s)
input: type is an integer, s a string
return: result of comparing s with DK
(of appropriate type)

s	x	DC	DK
	F	F	T

and(a, b)
input: a, b booleans
return: logical and of a and b

a\ b	F	T	DC	DK
F	F	F	F	F
T	F	T	T	DK
DC	F	T	DC	DK
DK	F	DK	DK	DK

or(a, b)
input: a, b booleans
return: logical or of a and b

a\ b	F	T	DC	DK
F	F	T	F	DK
T	T	T	T	T
DC	F	T	DC	DK
DK	DK	T	DK	DK

compare(op, a, b)
input: a, b both integers or strings
op: one of EQ, NE, LT, GT, LE or GE
return: a op b according to the tables below

case EQ

s1\s2	x	DC	DK
x	s1 EQ s2	F	DK
DC	F	T	F
DK	DK	F	DK

case NE

s1\s2	x	DC	DK
x	s1 NE s2	T	DK
DC	T	F	T
DK	DK	T	DK

case LT or GT

s1\s2	x	DC	DK
x	s1 op s2	DC	DK
DC	DC	F	DC
DK	DK	DC	DK

case LE or GE

s1\s2	x	DC	DK
x	s1 op s2	F	DK
DC	F	T	F
DK	DK	F	DK

CHAPTER VI

RELATIONAL ALGEBRA IMPLEMENTATION

In this chapter we describe the implementation of the various operations of the relational algebra in terms of the basic components introduced in chapter IV: System Overview. These operations have been implemented as a collection of functions or procedures written in the programming language C. We already saw how these functions or procedures were invoked by the Interpreter. We give the main lines of the algorithms underlying these functions or procedures intermixed with statements in C so that it can be used as part of a programmer's manual.

PROJECT

Project creates a new relation by projecting a relation, of index R, on domlist, a linked list of domain indexes. The parser has already checked that there is no expression among these domains, but names only. It remains to check that R is defined on these domains. Sort is used to eliminate duplicates which may be produced when attributes are removed. So, the resulting relation is sorted on domlist.

Project is invoked not only by the Interpreter but also by the following routines (described below): list_actualize, to remove the domains which have been created by the actualization process but were not specified by the user in the project list; increment, to remove from the right operand any domain not appearing in the left one; rename_increment, to trim down either operand according to the domain lists specified by the user.

```
project( domlist, R)
 input:   domlist    linked list of domain indexes
          R          relation index

 return:
        result_R, index of relation obtained
        when projecting R on domlist

 method:
   if domlist is empty
     return index of NULL relation (no domains, no tuples)
```

```
else
 ( 1) create result_R, the resulting relation

 ( 2) set tuple_size of result_R to sum of length of
        domains in domlist

 ( 3) set ntuples of result_R to number of tuples of R

 ( 4) set domlist and sortlist of result_R to domlist

 ( 5) length== sum of the length of the domains in domlist

 ( 6) make an alias out of R and domlist;
        rel_index is the index of that alias;
        copy info in rd_ and rel_table from R to rel_index;

 (7A) if neither relation rel_index nor R are in core
   ( a) get storage for R and read it in, thus:

          freeze( R);
          I== icrel_get( R, rel_tuple_size( R));
          icrel_fill( R, I);

   ( b) get memory for the pairs of iptr, kptr pointers;
          one pair for each tuple of R

   ( c) get memory to store the result:
            J== icrel_get( rel_index, length);

   ( d) at this point we are done with claiming storage;
          we unlock the pages for R and rel_index:
             unfreeze( R); unfreeze( rel_index);

   ( e) if domlist is a prefix of rel_sortlist( R)
         (no sort is needed)
         -set each kptr to the key extracted
             from the corresponding tuple:
               ptrs[ i]. kptr== get_key( icrel_line( J, i),
                              R, I, i, domlist, length);

       else
         -set each kptr to the key extracted from the
            corresponding tuple as above; however convert
            any negative integer to its 9-complement
            before extracting the key and after sorting:

            _9_s_complement( R, I, domlist);

            ptrs[ i]. kptr== get_key( icrel_line( J, i), R,
                              I, i, domlist, length);
            quick_sort( 0, rel_ntuples( R) - 1);

            _9_s_complement( res, J, domlist);
        -get the projected tuples in order:
            strcpy( icrel_line( I, i), ptrs[ i]. kptr);

        -free the pairs of kptr, lptr
        -set icrel_for_rel[ R] to NEGATIVE

(7B) else
        (R or rel_index is in icrel I)
        get the projected tuple using get_key:
            strcpy( icrel_line( I, i),
                get_key( tuple, R, I, i, domlist, length));

( 8) flag projected tuples which are duplicate:
        *icrel_line( I, i)= '\0';

( 9) let j be the number of non-duplicates:
        change_ntuples( res, j);

(10) write result to disk:
        icrel_flush( res, I, rel_ntuples( R));

(11) set icrel_for_rel[ rel_index] and
        icrel_for_rel[ res] to NEGATIVE
```

SELECT

Select creates a new relation from a relation, of index R in rel_table, and a boolean domain of index D in dom_table. Only the tuples evaluating to true for D in R participate in the result. D is a domain either on which R is defined or actualizable in R. The resulting relation is defined on the same domains as R. When D must be actualized in R all domains, including D, created through the actualization process are removed once the select operation has been done.

```
select( D, R)
 input:   D     index of    boolean domain on which to select
          R        "        operand relation

 return:
      tempR index of result relation

 method:
  ( 1) create tempR, the resulting relation

  ( 2) copy info in rd_ and rel_table from R to tempR

  ( 3) tsize== rel_tuple_size( R)

  ( 4) find the size of a tuple once all the domains
        needed to evaluate D have been actualized:
       -mark all domains as unvisited
       -for each domain reachable from D
          add exactly once its length to
          tuple_size of tempR

  (5A) if D is a virtual domain in R
       -read in R:
         freeze( tempR);
         I== icrel_get( tempR, rel_tuple_size( tempR));
         icrel_fill( R, I);

       -actualize D in tempR:
         actualize( D, &tempR, &I);

       -D is the last domain actualized:
         testpos== rel_tuple_size( tempR) - 1

       -tempR is now defined on all the domains
        initially in R plus those created to evaluate D;
        eliminate these from domlist of tempR:
           change_domlist(  tempR,
                  copy_index_list( rel_domlist( R)));
          -sort order of tempR is undefined:
              change_sortlist( tempR, index_list_allocate());

     (5B) else
          (D is actual in R)
          -if not already in core, read in R:
              freeze( R);
              if(( I== icrel_for_rel[ R]) === NEGATIVE){
                I== icrel_get( R, rel_tuple_size( R));
                icrel_fill( R, I);
              unfreeze( R);
              set icrel_for_rel[ R] to NEGATIVE

          -get position of D in R:
              testpos== rd_dom_pos( R, D);

     ( 6) for each tuple in icrel I
          if D was virtual in R
            prune all the domains not originally in R:
              icrel_line( I, i)[ tsize]== '\0';

          if D is not TRUE
            flag tuple as deleted:
              *icrel_line( I, i)== '\0';

     ( 7) let j be the number of tuples satisfying the
          selection criterion:
            change_ntuples( tempR, j);

     ( 8) write relation tempR to disk:
            icrel_flush( tempR, I, rel_ntuples( R));

     ( 9) set icrel_for_rel[ tempR] to NEGATIVE
```

LIST_ACTUALIZE

List_actualize creates a new relation from a relation, of index R in rel_table, and domlist, a linked list of domain indexes. The parser has detected that at least one domain in domlist may be virtual in R. We check first that, indeed, at least one domain is virtual in R. The result relation, S, consists of all the tuples of R, each tuple augmented with the values for the virtual domains. Each domain must be actualizable in R. It may be seen as an inverse project operation.

As already mentioned, any domain may be seen as the root of an expression tree. If a domain is constant or actual, the tree consists of a single node: the domain itself. Otherwise, the tree is non-trivial and may overlap some other expression trees. Therefore, domlist may be seen as a forest of possibly overlapping expression trees. As the forest is visited, we mark the domains. Hence, any domain is actualized at most once. For each domain in domlist, list_actualize calls the recursive procedure actualize.

Extra domains, i.e. domains not appearing in domlist, may be created by the actualization process. Hence, we return the result of the projection of S on domlist in order to eliminate these extra domains.

```
list_actualize( domlist, R)
  input:    domlist    linked list of domain indexes
            R          relation index

  return:
    S, index of relation obtained when actualizing domlist in R;

  method:
    ( 1) compute size: the width of a tuple of R augmented
         with all the values obtained when actualizing
         exactly once all the domains in the trees
         rooted in domlist:
          -mark unvisited all domains
          -for each domain reachable from members of domlist
             add exactly once its length to size

    ( 2) create S, the resulting relation

    ( 3) copy info in rd_ and rel_table from R to S

    ( 4) set tuple_size of S to size
( 5) bring R into memory:
       freeze( S);
       I= icrel_get( S, size);
       icrel_fill( R, I);

( 6) for each virtual domain, DD, in domlist,
       actualize DD in S in I:
         actualize( DD, &S, &I);

( 7) eliminate intermediate domains obtained through the
       actualization process and which appear neither in
       rel_domlist( R) nor in domlist:
         -append to domlist all the domains on
          which R is defined
         -S= project( domlist, S);

( 8) write result to disk:
         icrel_flush( S, I, rel_ntuples( R));

( 9) unfreeze( S);
```

ACTUALIZE

Given the virtual domain of index D, the relation of index R, pointed to by ptr_R and contained in the icrel pointed to by ptr_I, actualize computes for each tuple the value of D in R. If R is defined on D then we return to the calling routine. Otherwise, we actualize the left operand of D through a recursive call. If D is defined in terms of a binary or ternary operator or if it is a vertical domain associated with a by_list or an order_list, then recursive calls are also used to actualize the other operands. The calls are issued so that the tree rooted at D is visited in a preorder fashion.

Pointers are used to pass the relation and icrel indexes since actualizing may require the sorting of R which may create an alias and store the alias in a new icrel.

```
actualize( D, ptr_R, ptr_I)
 input:   D        index of virtual domain
          ptr_R    pointer to relation index
          ptr_I    pointer to icrel

 (1A) if D is not virtual in *ptr_R
        return

 (1B) else
        if D is not a constant domain
          actualize the first operand domain:
            actualize( dom_opnd1( D), ptr_R, ptr_I);

    (2A) if D is a 2-operand domain
            actualize the second domain:
              actualize( dom_opnd2( D), ptr_R, ptr_I);

    (2B) else
     (3A) if D is a vertical domain
            if dom_operator( D) is equiv or par
              for each d in dom_by_list( D)
                actualize( d, ptr_R, ptr_I);

          if dom_operator( D) is fun or par
            for each d in dom_order_list( D)
              actualize( d, ptr_R, ptr_I);
```

```
(3B) else
         if D is a 3-operand domain
           actualize the second domain:
             actualize( dom_opnd2( D), ptr_R, ptr_I);
           actualize the third domain (stored in header
                                node of by_list):
             actualize( *dom_by_list( D), ptr_R, ptr_I);


-if D is defined through a RENAME in terms of
   a constant domain
     modify accordingly domlist in ptr_R:
       index_list_change( rel_domlist( *ptr_R),
                                  dom_opnd1( D), D);
 else
    compute and append value for D:
       tuple_actualize( D, ptr_R, ptr_I);

-append D to domlist of *ptr_R
```

RENAME_INCREMENT

After execution of the following routine, the domlist of R will have been replaced by the domlist supplied by the user. The routine also completes a renaming or renaming incremental assignment. In the first case, the procedure rename is invoked. In the second case, if R is new a mere assignment is performed. Otherwise, the routine increment completes the work.

```
rename_increment( R, domlist_R, S, domlist_S, opcode)
 input:   R and S           relation indexes
          domlist_R( or S)  linked list of domain indexes
          opcode            either RENAME or RENAME_INCREMENT
 method:
  project S on domlist_S:
   S= project( domlist_S, S)

  if opcode == RENAME
   change_domlist( R, domlist_R)
   rename( R, S)

  else
   if R is a new relation
    assign( R, S)
    change_domlist( R, domlist_R)
    change_sortlist( R, index_list_allocate())

   else
    project R on domlist_R
    increment( R, S)
```

ASSIGN

The following routine completes a simple assignment. The relation of index R will be identical to the relation of index S. S is deleted if it is a temporary relation.

```
assign( R, S)
 input:    R and S    relation indexes

 method:
  -copy all info in rd_ and rel_table from S to R
  -make a copy of file S under the name R
  -reset icrel_for_rel[ R] to NEGATIVE
```

INCREMENT

The following completes an incremental assignment. The file for R is replaced by the concatenation of the current file for R and the file for S. R is projected on its domlist to eliminate any duplicates created by the concatenation operation.

```
increment( R, S)
 input:    R and S    relation indexes

 method:
  -append file for S to the one for R
  -project R on its domains to eliminate duplicates:
     T== project( rel_domlist( R), R);
  -assign( R, T);
```

RENAME

The following, called by rename_increment, performs the book-keeping necessary in order to complete a renaming operation.

```
rename( R, S)
 input:    R and S    relation indexes

 method:
   -set ntuples of R based on ntuples of S
   -set tuple_size of R based on tuple_sizeof S

   -reset rd_ and rel_table entries for R:
      reset_count( R);
      reset_dom_pos( R);

   -change sortlist of R to its domlist
   -make a copy of file S under the name R
   -reset icrel_for_rel[ R] to NEGATIVE
```

chapter VII

VIEW EVALUATION

In this chapter we give the algorithms used to evaluate recursively defined relations. Our solution is general in that it works for all the views that a user can define in Aldat whether they are recursive or not. A base relation is a relation currently existing in the database. The evaluation of a view produces a base relation. We repeat here the syntax to define a view.

```
<view-statement>::=
    <identifier> ( initial <relational-expression> | ε)
    is <relational-expression>
```

The rules for <identifier> and <relational-expression> have been illustrated many times in chapter III. As well, the same chapter III explains the usefulness of the initial option.

Recall that the relation PARENT is defined on SENIOR and JUNIOR which are domains of type string and length 18. If "edward IV   elizabeth of york " is a tuple of PARENT, it indicates that edward IV is a parent of elizabeth of york. We already mentioned that in order to find for any two persons whether one is a descendant of the other it suffices to compute ANCESTOR, the transitive closure of PARENT. We have the following:

```
relation ANCESTOR ( SENIOR, JUNIOR);
ANCESTOR is PARENT [ ujoin ]
                ( ANCESTOR [ JUNIOR icomp SENIOR] ANCESTOR)
```

where icomp is the natural composition.

Recall the deferred evaluation mode characterizing a view statement. That is, when the user enters such a statement intermediate code is generated. The interpreter will evaluate the code only when the user triggers the evaluation process. That is, whenever ANCESTOR is used in an executable statement or the user enters pr!!ANCESTOR. Notice the declaration of ANCESTOR as being defined on SENIOR and JUNIOR. Recall that this is mandatory for recursively defined relations because the attributes of such a view can not be determined by the parser.

An alternative is to use the initial option.

Merrett [MERR84b] has suggested that ANCESTOR defined above is easily implemented as the iterative loop:

```
ANCESTOR <- PARENT;
repeat
  TEST <- ANCESTOR;
  ANCESTOR <- PARENT [ ujoin ]
                  ( ANCESTOR [ JUNIOR icomp SENIOR] ANCESTOR);
until( TEST = ANCESTOR);
```

Let us consider a more involved example. We use a simplified syntax in order to avoid cluttering up the presentation with details irrelevant to the discussion. In particular, we do not specify on which attributes the views are defined.

```
V is rel_exp( W, X);    W is rel_exp( A, Z, X);
Z is rel_exp( A, Y);    Y is rel_exp( W, B);
X is rel_exp( Q, B);    Q is rel_exp( A, T);
T is rel_exp( C, X);    S is rel_exp( W, Q);
```

The first one indicates that V is defined in terms of W and X, typically through a μ-join or a σ-join. Assume that A, B and C are base relations.

Observe that some views are defined in terms of each other. We already saw a relation, ANCESTOR, defined in terms of itself. A closer look at the example reveals that X is defined in terms of Q, Q in terms of T and T in terms of X. That is, X is recursively defined and so are Q and T. In such a case, we say that the recursion is indirect whereas in the case of ANCESTOR we say that it is direct.

A collection of view definitions determines a directed graph, say G, where the vertices are the relations in the database and an edge, say RS, from vertex R to vertex S indicates that the view R is defined on S, or, more specifically, that S appears in the relational expression defining R. If we are to use the iterative process described above, then X, Q and T must be evaluated simultaneously.

```
repeat
  TEST_X <- X;
  TEST_Q <- Q;
  TEST_T <- T;
  X is rel_exp( Q, B);
  Q is rel_exp( A, T);
  T is rel_exp( C, X);
until( TEST_X = X  and  TEST_Q = Q  and  TEST_T = T)
```

This would work fine, had one of X, Q and T been used in an executable statement. Similarly, W, Y and Z are defined in terms of each other and such a mechanism would be adequate. However, if the evaluation of V were triggered this would not quite work because V depends on X (and indirectly on Q and T) but X does not depend on V.

Therefore, whenever a view appears in an executable statement we

1.- determine all the views on which it depends;

2.- determine which views are mutually recursive;

3.- determine in which order the views must be evaluated;

4.- repeat the iterative process given above for each view, evaluating simultaneously the views which are mutually recursive.

Step 1, 2 and 3 are achieved by finding the maximal components of G.

Definition: we call strongly connected component of a directed graph a maximal set of vertices such that there is a path between any two vertices in the set.

Algorithm FSCC: find strongly connected components

1.- Perform a depth first search of G and number the vertices in order of completion of the recursive calls.

2.- Construct a new directed graph, GR, by reversing the direction of every arc in G.

3.- Perform a depth first search of GR starting from the highest numbered vertex and according to the numbering found at step 2. If not all vertices are reached start the next depth first search from the highest numbered remaining vertex.

This algorithm is taken from [AHO 83]. Implementations of algorithms to perform depth first search and determine strongly connected components are rather mundane. Hence, we will not dwell much on ours. Recall that an entry of rel_table comprises the following fields, the first two are non-empty for views only:

start	index in memory array of intermediate code;
points to the beginning of the code corresponding
to the relational expression defining the view V.

defined_on	linked list of relation indexes appearing in
the relational expression for the view V.

defines	linked list of relation indexes defined by a
relational expression in which V appears
(in this case, V need not be a view).

The graph G is described by defined_on, GR by defines. GR, so obtained, is actually bigger than needed but it contains as a subgraph all the vertices and edges of interest. This, because we need to consider only the relations visited during FSCC step 1 whereas GR may contain some vertices which have not been visited during that step. The lists defined_on and defines are built at parse time. Let us pursue our example:

	defined_on	defines
A		Q, W, Z
B		X, Y
C	T	
Q	A, T	S, X
S	Q, W	
T	C, X	Q
V	W, X	
W	A, X, Z	S, V, Y
X	B, Q	T, V, W
Y	B, W	Z
Z	A, Y	W

Each time FSCC step 3 is performed, we build a list of the views revisited. Each list is a strongly connected component of G. We store these lists in tree which is an array of index_list, a data type described in chapter IV (section c). Suppose that the evaluation of V is triggered. Tree would contain six lists.

tree:	0	1	2	3	4	5
	V	W	X	C	B	A
		Y	T			
		Z	Q			

1.- The graph G contains only the vertices appearing in tree. That is, the vertices which can be reached by following edges coming out of V. These are the relations on which V depends. However GR contains also the view S since S is reached when we consider the elements of the defines lists of W and Q.

2.- Base relations occur by themselves since they do not depend on any other relations.

3.- The ordering is not unique. However, the algorithm guarantees that all the relations defining, directly or not, a view appear on the right of the view in tree.

4.- Since a list is a strongly connected component, if it comprises a view then it contains all the views which are mutually recursive with it and only these.

The following procedure completes the implementation of the mechanism to evaluate views. It is called each time the interpreter is about to pop a relation from the run time stack. It checks whether the top element of the run time stack is a view. If so, the strongly connected components are determined. Step 4 described above is performed for each element of tree, starting with the rightmost.

The determination of the strongly connected components requires to mark the vertices as visited, revisited and so on. We use for that purpose arrays of integers, each comprising MAX_REL entries. We mention rank_or_initial. It is used to store the number assigned to a vertex during FSCC step 1. It is later used to store the starting address of the code corresponding to the relational expression in the initial option of the views. If S is a view defined with the initial option, the address of the corresponding code is saved in rank_or_initial[S] and the start field of S changed so as to point to the code defining the view. We restore the start field of S once S has been evaluated. Thus, each time the evaluation of S is triggered, the initial code is reevaluated exactly once as claimed in chapter III. We use test_rel, an array of MAX_REL integers, to store the index in rel_table of the TEST relations, one per view, mentioned above.

POP_AND_EVAL_IF_VIEW()

```
pop_and_eval_if_view( S)
 input:    S pointer to run_time_stack
 return:   top element of run_time_stack after popping it
           and triggering its evaluation if it is a view

 method:
  -R== pop( S)
    if R is not a view or
     we are already evaluating another view
        return( R)
```

```
else
( 1) for each relation S in rel_table reset flag:
       rank_or_initial[ S]== NEGATIVE;

( 2) rightmost== find_strongly_connected_components( R)

( 3) for each relation S in rel_table reset flags:
        rank_or_initial[ S]== NEGATIVE;
        test_rel[ S]== NEGATIVE;

( 4) for each T in tree starting with the rightmost

  ( a) current_T_done <- FALSE

  ( b) for each relation S in T
          if S is not a view
            current_T_done <- TRUE
            go to ( 5)
          else
            test_rel[ S]== make_test_rel( S);

  ( c) repeat
         for each relation S in T
           TEST_S <- S:
             assign( test_rel[ S], S);

         for each relation S in T
           evaluate S:
            interpreter( rel_start( S));

         for each relation S in T
           compare S with TEST_S:
             if not relations_are_equal( test_rel[ S], S)
               current_T_done== FALSE

        until current_T_done== TRUE

( 5) already_evaluating== FALSE;

( 6) restore start of view defined with initial
      option for further use of the view:
        change_start( S, rank_or_initial[ S]);

( 7) delete test rel

( 8) return( R)
```

RELATIONS_ARE_EQUAL()

The following performs relation comparisons. Two relations, R and S, are equal if and only if they are defined on the same attributes and contain the same tuples up to a reordering of the lines and/or the columns.

```
relations_are_equal( R, S)
 input:     R and S, indexes in rel_table

 return:     TRUE iff relations R and S are equal

 method:

  if R == S

    return( TRUE)

 return:     TRUE iff relations R and S are equal


  if rel_ntuples( R) !== rel_ntuples( S)
    return( FALSE)

  if rel_arity( R) !== rel_arity( S)
    return( FALSE)

  if any domain of R is not a domain of S or vice versa
    return( FALSE)

   project both R and S on a common ordering
   of their attributes; say domlist of R:
     assign( R, project( rel_domlist( R), R));
     assign( S, project( rel_domlist( R), S));

   if their corresponding files differ
     return( FALSE)
   else
     return( TRUE)
```

MAKE_TEST_REL(R)

The following, given an index R in rel_table, makes a unique name, inserts it in rel_table and returns the corresponding index.

make_test_rel(R)

input: R, index in rel_table

return: index of a test relation corresponding to R

method: -make a name like (assume R== 17) test.17

-insert it in rel_table and return its index

chapter VIII

ERROR HANDLING

As mentioned previously, relix can operate in either one of two modes with respect to errors: interactive or batch mode. Errors fall in one of the four following classes:

WARNING: relix has detected something which is not absolutely regular, but which is very unlikely to ~~cause~~ any problem.

Example: when entering a list of domains, a given domain has been specified more than once; a file contains fewer tuples (or one of its tuples seems longer) than expected; operations are attempted on domains of type real; a user-specified identifier is too long and truncated.

ERROR: relix is experiencing some difficulties. However, it expects to be able to resume processing correctly. Some results may be lost, but operations not depending on them can still be carried out.

Example: relix can not open a file for writing; relix can not redirect the standard input to a user-specified file; the parser encounters an invalid token.

SEVERE: relix has encountered something which is clearly illegal. Continuing processing the current statement, although physically possible, is unlikely to produce any valid result.

Example: division by zero; arithmetic overflow; mismatch of domain type in a domain expression; system table overflow; syntax error (see below).

CATASTROPHE: at this level, not only is keeping on processing meaningless, but also, impossible: some relix data structures have probably been damaged, and dangerous: some files may get corrupted.

Example: no memory available for result or operand relation; standard input can not be redirected to the terminal; relix bugs (see below).

Syntax errors constitute a major source of SEVERE errors. The parser used by relix, although powerful in many respects, can just not handle any wrong statements like

let let	(the first let must be followed by name);
A be B + C	(be must be preceded by let);
R <- [[A, B] in S	([is misplaced)

When compiling a multi-statement program and a serious error occurs, compilers operating in batch mode use the following approach: no more code is generated, parsing of the remaining statements is attempted in order to produce as much useful information as possible. In our interactive setting, we adapt this as follows: ignore the current (erroneous statement), reset the parser to its initial state and get the next statement. Changes made to external (to the parser), global data structures like REL and DOM are not undone. Hence, the next statements may produce results of dubious value. Last, but not least, a relix bug will generate, usually and hopefully, a CATASTROPHE trap. The user may then report any such problems to the people in charge of maintenance.

Relix keeps a tally of the IO (input/output) operations performed. Whenever an error occurs relix issues an explanatory message and records the most serious level of error encountered. Whenever an IO request is issued, relix performs the following:

```
if mode is interactive
  -return

otherwise
  if level is ERROR or SEVERE
    -abort if the threshold of IO operations for
      that level has been reached
```

The thresholds can not be changed by the user. They decrease with the severity of the related errors. Relix tolerates many venial mistakes, more so than serious offenses.

Error messages have the form:

"ladt.c: icrel_fill: no more data to read"

where ladt.c is the name of a relix source code file, icrel_fill is the name of the function where the error has been detected and "no more data to read" is a tentative explanation of what most probably happened. In this case, the relation file contained fewer tuples than expected.

CHAPTER IX

CONCLUSION

This thesis has outlined the design and implementation of a new version of Aldat. The design is characterized by the following:

1.- Aldat is presented as a stand-alone programming language. The relation is the unique data structure available to the user. The syntax is simple.

2.- The user can define views in a natural way. Only minor additions to the syntax were needed.

The resulting system, relix, is characterized by the following:

1.- Relations must be small enough so that the operands of any operation of the relational algebra can fit into primary memory.

2.- Relations are stored as character data. Attributes are of fixed-length. Hence, all the tuples of a given relation have the same length.

3.- The following domain types are available: boolean, integer and string (array of characters). The domain algebra, including null values, is implemented.

4.- The following features of the relational algebra are implemented: project and select with actualize; a wide range of assignments. These operations have been implemented using sort techniques where appropriate. The μ-join is also implemented, but by a complementary work, not as a part of this thesis.

5.- Evaluation of recursively defined relations is supported. In particular, one can easily compute the transitive closure of a graph of which the edges are the tuples of a relation. The user need not, and can not, use loop structures. These are hidden in the implementation.

6.- The system is interactive with a short response time as illustrated in the following section. The implementation is highly portable from one UNIX system to another.

a) SOME RESULTS

We work with a database named CROSS_REF in order to illustrate the response time that one can expect when using relix. These tests have been realized on the Cadmus operating in single-user mode.

Domain Table: CROSS_REF

Index	Name	Length	Actual	Type
10	CALLER	40	T	STRG
11	CALLEE	40	T	STRG
12	FILE	15	T	STRG
13	FTYPE	10	T	STRG

FILE	name of a	file containing C-functions
CALLER	"	function in the previous
CALLEE	"	invoked by the previous
FTYPE	type of	the calling function

For each example, we use three subsets of the same relation. They differ in the number of tuples: 100, 500 and 1611. To begin, we turn the UNIX file XREF into a relation of the same name defined on the following domlist: FILE, FTYPE, CALLER and CALLEE. A tuple like

attribute name	*attribute value*
FILE	"ladt.c "
FTYPE	"int "
CALLER	"buffer_get "
CALLEE	"enqueue_first_used "

indicates that the function buffer_get returns an integer, calls the function enqueue_first_used and is found in the file ladt.c.

```
RELATION XREF ( FILE, FTYPE, CALLER, CALLEE) <- XREF;
```

We consider two cases: first, the file is ordered according to domlist; second, a single tuple is out of order. Recall that this is the worst case for our sort routine based on quicksort. Times are given in seconds.

	100	500	1611
sorted	1	13	45
one out of order	3	30	165

Let us consider the following select operations:

```
SMALLER          <- where CALLER < CALLEE in XREF;

NOT_SMALLER      <- where CALLER >= CALLEE in XREF;
```

	100	500	1611
SMALLER	(42) 2	(258) 7	(643) 30
NOT_SMALLER	(58) 2	(262) 7	(968) 30

We indicate between parentheses the number of tuples in the resulting relation. Observe that the columns add up to the number of tuples in XREF. The selection conditions being quite similar to each other explains why it takes the same time to perform either.

We consider now a project operation which requires us to sort the relation:

```
INV_XREF <- [ CALLEE, CALLER, FILE] in XREF;
```

	100	500	1611
INV_XREF	(100) 3	(500) 16	(1611) 55

We finish with a few virtual domain definitions and actualizations. The first actualization is very simple. It entails a project operation which does not modify the sort order of the relation.

```
          let FRILL_A be "/* ";

          let FRILL_B be " */";

FRILLS <- [ FRILL_A, FILE, FTYPE, CALLER, FRILL_B] in XREF;
```

	100	500	1611
FRILLS	(22) 4	(87) 15	(279) 43

The second actualization counts the number of tuples agreeing in the FILE attribute. This one does not require a different sort order either.

```
let COUNT be equiv + of 1 by FILE;
COUNT_R <- [ FILE, COUNT] in XREF;
```

	100	500	1611
COUNT_R	(1) 6	(5) 30	(15) 105

The last actualization considered numbers the tuples using a different sequence for each value of the FILE attribute. Notice that the attributes CALLER and CALLEE form together a key of XREF. That is, any tuple is uniquely identified by the values of these two attributes. We permute them at will so that relix must perform many sorts in order to produce the requested result.

```
let NUMBER be par + of 1 order CALLER, CALLEE by FILE;
NUMBER_R <- [ NUMBER, CALLEE, CALLER] in XREF;
```

	100	500	1611
NUMBER_R	(100) 11	(500) 55	(1611) 195

These results support our conclusion with respect to response time. That is, a relation can be processed with very acceptable a response time on the Cadmus, provided it does not comprise more than a few tens of kilobytes. Faster machines, like the Masscomp and the Vax-780, should support the same response time for even bigger relations.

b) LIMITATIONS AND FURTHER WORK

Some features of Aldat are not currently available. Furthermore, even if relix presents a satisfying response time, some optimizations could be considered. Hence the following points constitute areas where further work could be done.

1.- When evaluating relational expressions, intermediate results are written from memory pages to disk. This is not necessary provided a mechanism is designed in order to write out pages just before they are to be used for some other relations. Of course, before terminating execution some pages may need to be saved on disk. It seems however that the probability of losing results, say in the case of a system crash, is bigger with such an approach than with ours.

2.- A similar phenomenon occurs when evaluating recursive relations. However, the volume of these temporary relations is quite impressive. Recall the relation PARENT and the view ANCESTOR. When PARENT contains five consecutive generations, nearly thirty such relations are generated in order to compute ANCESTOR. Since this evaluation has been built on top of our implementation of project, select and actualize, one could study how these can be modified to reduce the bulk of such intermediate results and improve the overall response time.

3.- As mentioned in various places, many features can be added to the current implementation. With respect to the domain algebra, one could add procedures in order to: fully support domains of type real; allow the user to define his own operations on domains; allow new domain types like chronological, set, interval and others. Work is underway by other implementors to add a relational editor and QT-selector to the relational algebra. Some work remains to be done to complete the implementation of the σ-join.

4.- Relix depends heavily on the assumption that the operands of any relational operation fit into main memory. One could study the changes to make in order to remove that restriction. Only past the code generation phase are changes required. Naturally, the effect on the response time should be minimized.

5.- Attributes of variable length, for example those of type string, would provide a more flexible tool. In order to support these, the fixed-length tuple assumption must be dropped. It seems that the modifications required are so significant that a new implementation should be considered.

6.- It is not clear what use can be made of recursively defined domains. This is an area where more research is needed. As far as implementation is concerned, it seems that only minor modifications are required in order to support, at least to some extent, that type of recursion.

7.- Most of the facilities supplied by UNIX are accessible from within relix. The idea of presenting relix as an operating system built on top of UNIX is appealing from user and programmer points of view. On the other hand, a casual database user is likely to desire tools to manage the information in the database. The full power of UNIX is not needed and user's needs may be better served in an environment specially tailored to provide these tools.

APPENDIX A

ALDAT GRAMMAR

```
<program>     ::=   <command> | <statement>

<command>     ::=   APPEND_REL | BATCH | CREATE_DOM | CREATE_REL

                |   DEL_DOM | DEL_REL | HELP | INPUT_FROM

                |   LINE_SHELL | MANUAL | PRINT_OBJECT

                |   PRINT_REL | QUIT | SAVE | SHELL

                |   SHOW_DOM | SHOW_RD | SHOW_REL

<statement>   ::=   <short_command> ';'

                |   <domain-declaration> ';'

                |   <relation-declaration> ';'

                |   <definition-statement> ';'

                |   <executable-statement> ';'

                |   <view-statement> ','
```

```
<short_command> ::=

            ( DEL_R | PRINT_R | SHOW_D | SHOW_R ) <identifier>

<domain-declaration>::=

            domain <identifier> <domain-type>

<identifier>::=

            <letter> ( <letter> | <digit> | '_')*

<domain-type>::=

            boolean | bool | integer | intg | real | float

            | ( string | strg) <digit>+

<relation-declaration>::=

            relation <identifier> <domain-list>

            ( '<-' ( <identifier> | <non_dc_dk_string>) | ε)
```

```
<definition-statement>::=

        let <identifier> be <domain-expression>

<domain-expression>::=

         <boolean-expression>

      |  <domain-expression> <ass-com-op> <domain-expression>

      |  <domain-expression> <other-bin-op> <domain-expression>

      |  <function-name> '(' <domain-expression> ')'

      |  <unary-op> <domain-expression>

      | red     <ass-com-op> of      <domain-expression>

      | equiv   <ass-com-op> of      <domain-expression>

                                     by <domain-list>

      | fun     <fcn-par-op> of      <domain-expression>

                                     order <domain-list>

      | par     <fcn-par-op> of      <domain-expression>

                                     order <domain-list>

                                     by <domain-list>

      | if      <boolean-expression> then <domain-expression>

                                     else <domain-expression>
```

```
    |   '(' <domain-expression> ')'

    |   <constant>

    |   <identifier>

<boolean-expression>::=

    <domain-expression> <comp-op> <domain-expression>

<ass-com-op>          ::=   '+' | '*' | '&' | '|' | max | min

<other-bin-op>        ::=   '-' | '/' | mod | '**' | '||'

<function-name>       ::=   abs | cos | isknown | log10 | log2

                       |    ln | sin | tan

<unary-op>            ::=   '-' | '+' | '¬ ' | not

<fcn-par-op>          ::=   <ass-com-op> | <other-bin-op>

                       |    pred | succ

<domain-list>         ::=   <domain-list> ',' <domain-expression>

                       |    <domain-expression>

<constant>            ::=   <boolean> | <integer> | <real> | <string>

<boolean>             ::=   dc bool | dk bool | false | true

<integer>             ::=   <digit>+ | dc intg | dk intg

<real>                ::=   <digit>* '.' <digit>*
```

```
<string>              ::=   <non_dc_dk_string> | dc strg | dk strg

<non_dc_dk_string>    ::=   '"' [ ^ ( '"' | '' | '0 ) ] '"'

<digit>               ::=   0 | 1 | ... | 9

<letter>              ::=   a | b | ... | z | A | B | ... | Z

<comp-op>             ::=   '<' | '>' | '<=' | '>=' | '=' | '~ ='

<executable-statement>::=

        <identifier> '<-' <relational-expression>

    |   <identifier> '<+' <relational-expression>

    |   <identifier> '[' <domain-list> '<-' <domain-list> ']'

                              <relational-expression>

    |   <identifier> '[' <domain-list> '<+' <domain-list> ']'

                              <relational-expression>

<view-statement>::=

        <identifier> ( initial <relational-expression> | ε)

    is  <relational-expression>
```

```
<relational-expression>::=

        <project-clause> <where-clause>

            in <relational-expression>

      |  <relational-expression>

            '[' <domain-option> <join-op> <domain-option> ']'

                  <relational-expression>

      |  '(' <relational-expression> ')'

      |  <identifier>

<project-clause>     ::=    '[' <domain-option> ']' | ε

<where-clause>       ::=    where <domain-expression> | ε

<domain-option>      ::=    <domain-list> | ε

<join-op>            ::=    <mu-join-op> | <sigma-join-op>

<mu-join-op>         ::=    ijoin | natjoin | ujoin

                     |      sjoin | ljoin | rjoin

                     |      drjoin | djoin | dljoin

<sigma-join-op>      ::=    <basic-sigma-join-op>

                     |      <negation> <basic-sigma-join-op>

                     |      icomp | natcomp
```

```
<basic-sigma-join-op>

        ::=    eqjoin | ltjoin | lejoin

        |      sub | gtjoin | gejoin
```

TABLE OF PRECEDENCE

-operators of lower precedence first

-operators on a given line have same precedence

-associativity is specified

left	'|' '&'
nonassoc	'<' '>' '<=' '>=' '=' '~='
left	max min
left	'+' '-'
left	'*' '/' mod
right	'**'
nonassoc	NOT

PARAMETERLESS COMMANDS

APPEND_REL	ar!	BATCH	batch!
CREATE_DOM	cd!	CREATE_REL	cr!
DEL_DOM	dd!	DEL_REL	dr!
HELP	h!	INPUT_FROM	input!
LINE_SHELL	'%'.*\n	MANUAL	man!
PRINT_OBJECT	po!	PRINT_REL	pr!
QUIT	q!	SAVE	sa!
SHELL	sh!	SHOW_DOM	sd!
SHOW_RD	srd!	SHOW_REL	sr!

ONE-PARAMETER COMMANDS

DEL_R	dr!!	PRINT_R	pr!!
SHOW_D	sd!!	SHOW_R	sr!!

RESERVED KEYWORDS

abs	be	bool	boolean	by	cos
dc	div	djoin	dk	dljoin	domain
drjoin	else	eqjoin	equiv	false	float
fun	gejoin	gtjoin	icomp	iejoin	if
ijoin	in	initial	integer	intg	is
isknown	lejoin	let	ljoin	ln	log10
log2	ltjoin	max	min	mod	natcomp
natjoin	not	of	order	par	pred
real	red	relation	rjoin	sep	sin
sjoin	strg	string	sub	succ	sup
tan	then	true	ujoin	where	

BIBLIOGRAPHY

[AHO 79] Aho A. V. and Ullman J. D., "Universality of Data Retrieval Languages", Proc. of Sixth ACM Symposium on Principles of Programming Languages, January 1979, pp 110-120.

[AHO 83] Aho A. V., Hopcroft J. E. and Ullman J. D., "Data Structures and Algorithms", Addison-Wesley, 1983.

[BARB85] Barbic F. and Rabitti F., "The Type Concept in Office Document Retrieval", Proc. of eleventh Int. Conf. on VLDB, Aug. 1985, pp 34-48.

[BERN83a] Bernstein P. A, "Database Theory: Where Has it Been? Where is it Going?", ACM SIGMOD Proc. Annual Meeting, May 1983, pa 2.

[BERN83b] Bernstein P. A and Goodman N., "Multiversion Concurrency Control-Theory and Algorithms", ACM TODS-8-4, Dec. 1983, pp 465-483.

[CERI84] Ceri S. and Pelagatti G., "Distributed Databases: Principles and Systems", McGraw-Hill, 1984.

[CHAK82] Chakravarthy U., Minker J. and Tran D., "Interfacing Predicate Logic Languages and Relational Databases", Proc. of first International Logic Programming Conf., Marseille, 1982.

[CHAM76] Chamberlin D. D. et al., "SEQUEL 2: a Unified Approach to Data Definition, Manipulation and Control", IBM Journal of Research and Development, Vol. 20, No. 6, Nov. 1976, pp 560-575.

[CHAM80] Chamberlin D. D., "A Summary of User Experience with the SQL Data Sublanguage", Proc. Int. Conf. on Data Bases, July 1980, pp 181-203.

[CHAM81] Chamberlin D. D., Gilbert A. M. and Yost R. A., "A History of System R and SQL/Data System", Proc. of seventh Int. Conf. on VLDB, Sept. 1981, pp 456-464.

[CHAN81] Chang C., "On Evaluation of Queries Containing Derived Relations in a Relational Database". in Advances in Database Theory (Plenum Press, ed.), pp 235-260.

[CHIU82] Chiu G., "MRDSA User's Manual", SOCS-82-9, May 1982.

[CHON86] Chong A. T., "Implementation of Mu-join in Relix", McGill University, August 1986.

[CODD70] Codd E. F., "A Relational Model of Data for Large Shared Data Banks", CACM, Vol. 3, No. 6, June 1970, pp 377-387

[CODD71] Codd E. F., "Relational Completeness of Data Base Sublanguages", in Data Base Systems (R. Rustin, ed.), pp 65-98.

[CODD75] Codd E. F., "Understanding Relations", (ACM) FDT 7:3-4, 1975, pp 23-28.

[CODD79] Codd E. F., "Extending the Database Model to Capture More Meaning", ACM TODS-4-4, Dec 1979, pp 397-434.

[DATE82] Date C. J., "An Introduction to Database Systems", third ed., Addison-Wesley, 1982.

[FERR82] Ferrier A. and Stangret C., "Heterogeneity in the Distributed Database Management System SIRIUS-DELTA", Proc of eighth Int. Conf. on VLDB, Sept. 1982, pp 45-53.

[FORB85] Forbes G. and Lalliberté N., "MINM: A Domain Algebra Implementation", McGill University, April 1985.

[GALL81] Gallaire H., Minker J. and Nicolas J. M., "Advances in Database Theory", Vol. 1, Plenum Press, 1981.

[GARD84] Gardarin G. and Gelenbe E., "New Applications of Data Bases", Academic Press, 1984.

[GELE82] Gelenbe E. and Gardy D., "The Size of Projections of Relations Satisfying a Functional Dependency", Proc of eighth Int. Conf. on VLDB, Sept. 1982, pp 325-333.

[HENS83] Henschen L. H. and Naqvi S,., "Synthesizing Least Fixed-Point Queries into Iterative Programs", Proc. IJCAI, Karlsruhe, 1983.

[HENS84] Henschen L. H. and Naqvi S,., "On Compiling Queries in Recursive First-Order Databases", JACM, January 1984.

[IMIE83] Imielinski T. and Lipski W. Jr., "Incomplete Information and Dependencies in Relational Databases", ACM SIGMOD Proc. Annual Meeting, May 1983, pp 178-184.

[JOHN75] Johnson S. C., "Yacc: Yet Another Compiler Compiler", Computing Science Technical Report No. 32, 1975, Bell Laboratories.

[JONG83] de Jonge W., "Compromising Statistical Databases Responding to Queries about Means", ACM TODS-8-1, March 1983, pp 60-80.

[KAME80] Kamel R. F., "The Information Processing Language Aldat: Design and Implementation", SOCS-80-14, August 1980.

[KENT83] Kent W., "The Universal Relation Revisited", ACM TODS-8-4, Dec. 1983, pp 644-648.

[KERS84] Kerschberg L., "Proceedings of the First International Workshop on Expert Database Systems", October 1984.

[KIM 79] Kim W., "Relational Database Systems", ACM Computing Survey, Vol. 11, No. 3, Sept. 1979, pp 185-211.

[KORT86] Korth H. F. and Silberschatz A., "Database System Concepts", McGraw-Hill, 1986.

[KUNG81] Kung H. T. and Robinson J. T., "On Optimistic Methods for Concurrency Control", ACM TODS-6-2, June 1981, pp 213-226.

[LACR77] Lacroix M. and Pirotte A., "Domain-Oriented Relational Languages", Proc. of third Int. Conf. on VLDB, Oct. 1977, pp 370-378.

[LESK75] Lesk M. E., "Lex-a Lexical Analyser Generator", Computing Science Technical Report No. 39, 1975, Bell Laboratories.

[LOUI82] Louis G. and Pirotte A., "A Denotational Definition of the Semantics of DRC, a Domain Relational Calculus", ACM SIGMOD Proc. Annual Meeting, September 1982, pp 348-356.

[MACG85] MacGregor R. M., "ARIEL-a Semantic Front-End to Relational DBMSs", Proc. of eleventh Int. Conf. on VLDB, Aug. 1985, pp 305-315.

[MAIE83] Maier D. and Ullman J. D., "Fragments of Relations", ACM SIGMOD Proc. Annual Meeting, May 1983, pp 15-22.

[MAIE83a] Maier D. and Ullman J. D., "Maximal Objects and the Semantics of Universal Relation Databases" ACM TODS-8-1, March 1983, pp 1-14.

[MAIE83b] Maier D., "The Theory of Relational Databases", Computer Science Press, 1983.

[MAIE84] Maier D., Vardi M. Y. and Ullman J. D., "On the Foundations of Universal Relation Model", ACM TODS-9-2, June 1984, pp 283-308.

[MERR76] Merrett T. H., "MRDS: An Algebraic Relational Database System", Canadian Computer Conference, May 1976, pp 102-124.

[MERR77] Merrett T. H., "Relations as Programming Language Elements", Information Processing Letters, Vol. 6, No. 1, Feb. 1977, pp 29-33.

[MERR81] Merrett T. H. and Zaidi S. H. K., "MRDSP User's Manual", SOCS-81-27, August 1981.

[MERR84a] Merrett T. H., "Relational Information System", Reston, 1984.

[MERR84b] Merrett T. H., "The Relational Algebra as a Typed Language for Logic Programming", Proc. of first Int. Workshop on Expert Database Systems, October 1984, pp 735-739..

[NILS79] Nilsson J., "Computational Scheme for Recursive Relational Languages", Proc. Workshop on Formal Basis for Databases, Toulouse, 1979.

[OZKA86] Ozkarahan e., "Database Machines and Database Management", Prentice-Hall, 1986.

[SACC82] Sacco G. M. and Schkolnick M., "A Mechanism for Managing the Buffer Pool in a Relational Database System Using the Hot Set Model", Proc of eighth Int. Conf. on VLDB, Sept. 1982, pp 257-262.

[SAGI81] Sagiv Y. and Yannakakis M., "Equivalence among Relational Expressions with the Union and Difference Operators", JACM 27:4, 1981, pp 633-655.

[SCHM77] Schmidt J. W., "Some High Level Language Constructs for Data of Type Relation", ACM TODS-2-3, Sept 1977, pp 247-261.

[SHOP79] Shopiro J. E., "Theseus-a Programming Language for Relational Databases", ACM TODS-4-4, Dec 1979, pp 493-517.

[SICH83] Sicherman G., de Jonge W. and van de Riet R. P., "Answering Queries without Revealing Secrets", ACM TODS-8-1, March 1983, pp 41-59.

[STON76] Stonebraker M, Wong E. and Kreps P., "The Design and Implementation of INGRES", ACM TODS-1-3, Sept 1976, pp 189-222.

[SU 78] Su S. Y. W. and Eman A., "Casdal: CASSM Data Language", ACM TODS-3-1, March 1978, pp 57-91.

[TODD76] Todd S. J. P., "The Peterlee Relational Test Vehicle- a System Overview", IBM System Journal, Vol. 15, No. 4, 1976, pp 285-308.

[TURN85] Turner R. and Lowden B. G. T., "An Introduction to the Formal Specification of Relational Query Languages", The Computer Journal, Vol. 28, No. 2, May 1985, pp 162-169.

[ULLM82] Ullman J. D., "Principles of Database Systems", Computer Science Press, second edition, 1982.

[ULLM83] Ullman J. D., "On Kent's `Consequences of Assuming a Universal Relation`", ACM TODS-8-4, Dec. 1983, pp 637-643.

[ULLM85] Ullman J. D., "Implementation of Logical Query Languages for Databases", ACM TODS-10-3, Sept. 1985, pp 289-321.

[VANR83] Van Rossum T., "Implementation of a Domain Algebra and a Functional Syntax", SOCS-83-18, August 1983.

[WONG83] Wong E. and Katz R. H., "Distributing a Database for Parallelism", ACM SIGMOD Proc. Annual Meeting, May 1983, pp 23-29.

[ZANI85] Zaniolo C., "The Representation and Deductive Retrieval of Complex Objects", Proc. of eleventh Int. Conf. on VLDB, Aug. 1985, pp 458-469.

[ZLOO77] Zloof M. M., "Query-by-Example: a Data Base Language", IBM System Journal, Vol. 16, No. 4, 1977, pp 324-343.