Implementation of Procedures in a Database Programming Language

Rebecca Lui

School of Computer Science
McGill University, Montreal

November 1996

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

Abstract

This thesis documents the design and implementation of procedures in a database programming language. The purpose of this thesis is to integrate procedure facilities into an existing relational database system.

A relation is defined over a set of attributes. Given the values of a subset of attributes as input, a selection operation looks up the relation and outputs the values of the remaining attributes. Our proposed procedure construct supports this concept: a procedure is defined over a set of parameters, and the procedure can be invoked with different subsets of input parameters. This is accomplished by allowing procedures to have a sequence of *blocks* within the procedure body. Each block abstracts a sequence of actions which requires a subset of parameters as input. Users can select different blocks to be activated by supplying different subsets of input parameters. While a relation can be selected with any subset of input attributes, a procedure can only be invoked with defined subsets of parameters.

Our proposed procedures also support the usual properties of procedural abstraction: encapsulation and parameterization. A procedure declaration defines the procedure name, formal parameters, and the body. Procedure invocation is through the use of a stand-alone procedure call statement which specifies the procedure name and a list of actual parameters. Before a procedure is activated, the formal parameters within the procedure body are replaced by the corresponding actual parameters. Moreover, a procedure can be printed, shown, deleted, called by itself or other procedures, and passed as a parameter.

Résumé

Cette thèse documente la conception et la création de procédures dans un langage de programmation pour les bases de données. Le but de cette thèse est d'intégrer la notion de procédure dans un système de base de données relationnelle existant.

Une relation est définie sur un ensemble d'attributs. Les valeurs d'un sous-ensemble d'attributs étant données à l'entrée, l'opération de sélection vérifie la relation et retourne la valeur des attributs restants. Notre outil de construction de procédures proposé supporte le concept suivant: une procédure est définie sur un ensemble de paramètres, et elle peut être appelée avec différent sous-ensembles de paramètres d'entrée. Cela est accompli en permettant aux procédures d'avoir une séquence de blocs ("blocks") dans leur définition. Chaque bloc correspond à une séquence d'actions qui requiert un sous-ensemble des paramètres d'entrée. L'utilisateur peut sélectionner différents blocs à activer en envoyant différents sous-ensembles de paramètres d'entrée. Malgré que la relation puisse être sélectionnée avec n'importe quel sous-ensemble d'attributs à l'entrée, la procédure ne peut être appelée que sur un sous-ensemble défini de paramètres.

Les procédures proposées supportent aussi les propriétés usuelles d'abstraction procédurale: encapsulation et paramétrisation. La déclaration d'une procédure définie son nom, ses paramètres formels, et son implantation. L'appel d'une procédure se fait par l'utilisation d'une seule instruction spécifiant le nom de la procédure ainsi que la liste des paramètres. Avant que la procédure soit lancée, les paramètres utilisés dans la description d'implantation sont remplacés par les paramètres fournis lors de l'appel de procédure. De plus, une procédure peut être imprimée, visu-

alisée , détruite, appelée par elle-même ou par une autre procédure, ou passée en paramètre.

Acknowledgements

I would like to express my gratitude to my thesis supervisor, Professor T. H. Merrett, for his attentive guidance, invaluable advice, and endless patience throughout the research and preparation of this thesis.

I would like to thank my colleagues in the ALDAT lab, especially Dr. Heping Shang and Xiaoyan Zhao for their assistance on the usage of facilities in the lab and their consultation on the existing Relix system. A special thanks goes to Josée Turgeon who translated the abstract of this thesis to French. I would also like to thank all the secretaries of the School of Computer Science for their continuous encouragement, and all the system staff for their technical assistance.

I am grateful to Fonds pour la Formation de Chercheurs et l'Aide à la Recherche (FCAR) for their financial support throughout my study of the master program.

Finally, I would like to dedicate this thesis to my husband and my son, Phillip Hurst and James Hurst, whose love and inspiration made this thesis possible.

Contents

Chapter 1

Introduction

This thesis documents the implementation of procedures in a database programming language. The work was done by extending an existing relational database system with procedure features. We first have a historical overview of database programming languages and subprograms. The aim and outline of this thesis is given at the end of this section.

1.1 Database Programming Languages

Database management systems (DBMSs), such as System R [A^+76] and INGRES [SWKH76], usually impose severe restrictions on the kind of computations that can be performed. As a result programs that require complicated manipulation of the database cannot be written in these systems. The traditional solution to this problem has been to use two languages, embedding the database query languages in a computationally more powerful host programming languages. For example the query language SQL of System R may be used with COBOL [COD68], and the query language QUEL of INGRES may be embedded in a C [KR78] program. Access to data is done by issuing database commands in the host languages, and the database commands are preprocessed. As a consequence data has to be passed between these two languages. Since both languages might be semantically as well as structurally different, such transformations may lead to a loss of information. This problem is known as "impedance mismatch" [Atk78]. Its avoidance is an essential motivation for the development of database programming languages (DBPLs), which attempt to integrate features from programming languages and databases.

Early work on DBPLs has been surveyed [AB87]. Essentially two different approaches have been employed :

- the programming language approach which extends programming languages with database features; and
- the database approach which extends database systems with programming language features.

1.1.1 Programming Language Approach

This approach extends programming languages with database features. The first successful example is Pascal/R [Sch77]. Pascal/R is a system that extends Pascal [Wir71] with a relational database system. It extends the type system of Pascal to include a **relation** type built on Pascal records, and a **database** type built on relations. Other extensions to Pascal include:

- the iterative **foreach** statement and built-in procedures (**low** and **next**) to traverse relations;
- operators to update and to manipulate relations; and
- predicates to evaluate relations.

Pascal/R demonstrates that a simple extension to an existing programming language allows the manipulation of relations together with the appropriate mechanisms to support persistence and efficiency. This research has evolved into the design of DBPL [MS89], a relational programming language which extends Modula - 2 [Wir83] with **keyed set** and first-order predicates.

In late 1970s, a number of similar proposals and implementations emerged:

- PLAIN [Was79] extends Pascal to support two kinds of database type declarations: **relation** and **marking**. Markings are designed for storing intermediate results derived by evaluating expressions in the relational algebra over relations and markings. Relational update operators almost identical with those in Pascal/R are available. Unlike Pasal/R, the **foreach** statement in PLAIN is consistently defined to be applicable to all structured variables.

- RIGEL [RS79]extends Modula [Wir77] with **relation**, **view**, and **tuple** types. An expression that produces sequences of values, called a *generator*, is defined which integrates relational query expression with the **for**-statement to provide a consistent form of iteration. The language also provides a data abstraction facility to encapsulate the database interface and to support the disciplined use of views.

- Theseus [Sho79] extends EUCLID [L^+77] with the data types **relation** and **a-set** (for "association set"). Relations are defined to be sets of a-sets. Theseus ensures that the key values are unique and present in every a-set in any relation declared. The language allows programmers to define the insertion and deletion procedures on relations to support constraints on the database.

However, these languages limit the programmer to a particular data model with a limited set of operations for manipulating them. In addition, the programmer does not have explicit control over which data or data definitions are persistent and similarly which data is large scale.

A second important step at integrating features from databases to programming languages is the addition of persistence to programming languages. The persistence of a data object is the length of time that the object exists. In traditional programming languages data cannot last longer than the activation of the program without the explicit use of some storage agency such as a file system or a DBMS. In persistent programming, data can outlive the program, and the method of accessing the data is uniform whether it be long or short term data.

Atkinson, one of the early advocates of the persistence programming languages, proposes two principles for persistent data [Atk89]:

1. persistence independence whereby the persistence of a data value is independent of how the program manipulates that data value, and conversely, a fragment of program is expressed independently of the persistence of the data it manipulates; and

2. persistence data type orthogonality whereby, consistent with the general programming language design principle of data type completeness, all data values, whatever their type, should be allowed the full range of persistence.

Languages that achieve both of the principles are called persistent programming languages. Some examples are PS-algol [ACC81, ABC+83], Napier [AM88], and Galileo [ACO85].

PS-algol adds persistence to S-algol [CM82]. In PS-algol a database exists as a persistent heap external to the program, and there is a persistent object management system which makes the database appear transparently as part of the standard program heap. Data on the heap may persist beyond the lifetime of the program if they are made reachable from the top level table of a database. PS-algol supports procedures as first class data objects. That is, procedures are allowed the same rights as any other data object in the language, such as being assignable, the result of expressions or other procedures, elements of structures, etc. Combining persistence with first class procedures, one can simulate abstract data types, modules, and views in PS-algol [Atk85]. The work on PS-algol has evolved into the design of Napier which supports **rel**ational types and operations. In Napier, a column in a relation can be of any type permitted in the language (including procedure) .

Galileo is a strongly-typed, interactive DBPL based on ML [Mil84]. Galileo is designed to support semantic data model features (classification, aggregation, and specialization) as well as the abstraction mechanisms of modern programming languages (types, abstract types, and modularization). In Galileo, the database is part of a global environment inside which each transaction runs. A database program is an expression evaluated in this environment, and this evaluation may modify the updatable objects in the database.

Object-oriented database management systems (OODBMSs) start from object-oriented programming languages and add persistence and object sharing. Objects (instances of abstract data types) are encapsulated and only accessible through a predefined interface. Object manipulation is by invoking type-specific interface functions (methods). Typically object-oriented databases have a complete programming language environment but provide little means of descriptive set-operations [GM88]. This reflects their origin from the programming language realm. Varieties of research prototypes and commercial products are available. For examples, Gemstone [MSOP86], which is the first OODBMS, extends Smalltalk [GR83]. O2 [Deu90] and ObjectStore [LLOW91] evolved from C++ [Str86].

The DBPLs developed using the programming language approach provide the ability to manipulate database relations or objects but all at the tuple or object level. The interface to the storage manager ends up with a tuple- or an object-at-a-time interface thus forcing a lot of data and command traffic between the application program and the storage manager. Moreover, tuples and objects are assumed to be small enough to fit into RAM. Hence, objects that are too large for RAM cannot be programmed in these languages.

1.1.2 Database Approach

Another approach of the research in DBPLs is to extend relational database systems with programming language constructs such as complex objects, user-defined types, procedures and functions, etc. This approach can overcome the weaknesses of the programming language approach because database systems are oriented toward efficient access of large amounts of secondary storage data. Secondary storage access is embedded in the evaluation methods for the relational operators, making application independent of the access structures. We now review some examples of this approach.

POSTGRES [SR86] is an extended relational database system developed at the University of California, Berkeley in 1986 as a successor to INGRES. POSTGRES supports programming language concepts such as complex objects and user-defined data type. The query language of POSTGRES is POSTQUEL which is an extension of QUEL used in INGRES. The following built-in data types are provided in POSTGRES:

- integers;
- floating point numbers;
- fixed length character strings;
- unbounded varying length arrays of fixed types with an arbitrary number of dimensions;
- POSTQUEL, which contains a sequence of data manipulation commands used to simulate shared complex objects [SAHR84]; and
- procedure, which contains procedures written in a general purpose programming language with embedded data manipulation commands also used to represent complex objects [SAH87].

In addition to these built-in data types, user-defined data types can be defined using an interface similar to the one developed for ADT-INGRES [Sto86].

Starburst [LLPS91] is an experimental database system developed at the IBM Almaden Research Center in San Jose, California as a successor of System R. The data access and manipulation interface is an extension of SQL. Starburst includes support for complex objects and user-defined types and functions similar to those in POSTGRES.

Relix is an experimental relational database system developed at McGill University in 1986 [Lal86]. The data manipulation language of Relix is Aldat, an *al*gebraic *dat*a language proposed by Merrett [Mer77]. The purpose of developing Relix is to provide users with an interactive version of Aldat for exploring the concept of the relational database model as described in [Mer84]. Aldat has been extended to connect several programming language concepts to relations.

- Scalars, arrays, and records[ML89]. Together with associated operations and syntax, these data types are implemented as special cases of relations. For example, an one-dimensional array can be represented by a relation A(row, col, value), and the selection of an array element A[i,j] is a special case of projection and selection

$$[value]\ \textbf{where}\ row = i\ \textbf{and}\ col = j\ \textbf{in}\ A.$$

- Scoping[MS91]. It permits us to build up a structure of databases such as

  Canada

      R0

      Ontario

          R1

          R2

          ...

Quebec

Rn

Here, Canada, Ontario, and Quebec are databases and sub-databases, while R0, R1, R2, and Rn are relations. Canada contains the relation R0 and databases Ontario and Quebec. Ontario contains relations R1 and R2, and Quebec contains relation Rn. Assuming we were initially in the parent database Canada, the command

begin *Quebec*

would place ourselves in the scope of the Quebec database, and the command

end

would return again to Canada.

- Object-orientation and inheritance [Mnu92]. Classes are identified with relations. Object identity (Id) is given by ordinary attributes. Class hierarchy is represented by a relation where each superclass-subclass pair is represented by a tuple. Attribute inheritance is implemented by projecting the superclass's attributes from the join of the subclass with its superclass on the ID field:

  Superclass [*SuperId* **isa** *SubID*] *Subclass*

- Functions and parameterization [Mer93]. Functions are treated as special forms of relations called computations. As with relations, computations are defined over sets of attributes. Subset of attributes can be defined as input attributes and the remaining attributes are output. We will further discuss computations in Chapter 2.

1.2 Subprograms

The evolution of a programming language involves three steps[AS85]:

1. primitive expressions, which represent the simplest entities with which the language is concerned;

2. means of combination, by which compound expressions are built from simpler ones; and

3. means of abstraction, by which compound objects can be named and manipulated as units.

In programming we deal with two kinds of object: data and operations. Subprograms represent operational abstraction with two important properties[DJ95] :

- encapsulation which refers to the isolation of the operational details of a subprogram from the environment where it is used; and

- parameterization which refers to the ability to create a generalized abstraction that has the flexibility to perform a variety of activities based on the values of parameters.

In other words, the basic components of a procedural abstraction mechanism are [Ten81]:

- a body, a construct whose interpretation is deferred until the subprogram is invoked by being supplied with appropriate arguments; and

- a formal parameter part (possibly empty) which contains binding occurrences of the formal parameter identifiers .

The advantages of using subprograms as abstractions are as follows.

- Program units are simpler. This simplicity results in units that are easier to read, write, and modify. By hiding lower levels of detail, a unit can concentrate on a single task. This property is important in the implementation of top-down design.

- Program units are independent. Abstraction permits the actions of a subprogram to be independent from its use. Therefore, the program using the abstraction is not affected by the details of the abstraction's implementation.

- Program units are reusable. A subprogram, once defined, can be used with many different programming environments. This eliminates redundant programming effort and reduces errors.

1.2.1 Using Subprograms

The use of subprograms for realizing program modularity was developed as early as 1951 [WWG51]. Today two forms of subprogram are commonly used in programming languages, procedures (or subroutines) and functions. A procedure is a sequence of operations that is invoked as though it were a single statement. A function is a sequence of operations that returns a single value and is invoked from within an expression. Usually control returns to the point of invocation after execution of the subprogram thus forming a one-in, one-out control structure.

Languages such as FORTRAN [Ame66], Pascal [Wir71], and Ada [U.S80] make clear distinctions between procedures and functions. ALGOL 68 [W^+75] and C [KR78] regard a procedure as a special function in which the returned value is void. Modula - 2 [Wir83] treats a function as nothing more than a procedure which returns a value. In this thesis, we will maintain the distinction between procedures and functions. We will use subprograms to refer to both procedures and functions.

1.2.2 Parameters

There must be some means of passing data between the subprogram and the invoking unit. The usual methods of passing data are through global variables and through parameters. Moreover, parameters are an important part of almost every facility for subprograms, for it is through parameters that we generalize the action of a subprogram.

Parameters can be classified into three groups or modes [ML87]:

1. **INPUT** parameters, which are passed from the invoking unit to the subprogram at the time of invocation;

2. **OUTPUT** parameters, which are passed from the subprogram to the invoking unit at the time of return; and

3. **UPDATE** parameters, which pass information both ways.

A parameter needs to be specified at two points: in the invoking unit and in the subprogram definition itself. The parameters specified in the invoking unit are referred to as actual parameters or arguments. The parameters in a subprogram definition are referred to as formal parameters. The identifiers used for the formal parameters of a subprogram are purely local to the subprogram and have no connection with any identifiers used outside the subprogram.

When a subprogram is invoked (or called), each actual parameter is associated with its corresponding formal parameter. There are four mechanisms for pairing actual and formal parameters [DJ95].

- Positional association, which is commonly used by most imperative languages, simply associates the actual and formal parameters according to their relative positions in their parameter lists.

- Named association requires that the name of the formal parameter be appended to the actual parameter in the invocation statement.

- Mixed association wherein positional can be used for all parameters up to the first named association, after which all remaining associations must be named.

- Default association permits the specification of default values for formal parameters in the procedure header. When an invocation is made with no actual parameter associated with the formal parameter, the default value is used. This is appropriate only for input parameters.

Six mechanisms for passing information between the invoking unit and the subprograms are usually used.

- Pass by value is used for INPUT parameters only. The formal parameter acts as a local variable which is initialized with the value of the actual parameters. The actual parameter can be a variable or an expression.

- Pass by result is used for OUTPUT parameters only. The formal parameter again acts as a local variable but its value must be initialized locally with the subprogram body. After the statements of the body have been executed, the value of the formal parameter is assigned to the corresponding actual parameter. In this case the actual parameter must be a variable.

- Pass by value-result is used for UPDATE parameters only. The formal parameter is considered as a variable local to the subprogram. Its initial value is given by the value of the corresponding actual parameter, and the final value of the formal parameter is assigned to the actual parameter on completion of execution of the subprogram. This is another case where the corresponding actual parameter must be a variable.

- Pass by location (or reference). The address of the actual parameter is passed to the formal parameter. The formal parameter is considered as a local variable of the subprogram, but its location is the location of the corresponding actual parameter. Thus, any reference to the value of the formal parameter is considered to be a reference to the value of the actual parameter. Here again, the actual parameter must be a variable.

- Pass by Name and Expression. The actual parameter can be a variable or an expression. This involves the equivalent of textual substitution of the actual parameter for each occurrence of the formal parameter. In pass by name and expression, the actual parameter is passed unevaluated and is recalculated each time the formal parameter is used during the execution of the subprogram.

- Pass by Name. This is the same as pass by name and expression but the actual parameter must be a variable.

Type is an error detection mechanism for matching formal parameters and actual parameters. In a strongly typed language like Pascal, the types of the formal parameters must be specified at subprogram definition, and the corresponding actual and formal parameters must have the same type when the subprogram is invoked. We propose to use type as a mechanism to check if a procedure is being called correctly. We will discuss this mechanism in Chapter 3.

1.2.3 Others Forms of Subprograms

Other than procedures and functions, some programming languages support other forms of subprograms.

- Overloaded subprograms permit two or more subprograms to have the same name if they can be distinguished by the number or type of their formal parameters, or, in the case of a function, by the type of the return value. Overloading permits subprograms that perform the same operation on different parameter types to be called by the same name.

- Generic subprograms allow the same algorithm to be applied to different types of parameters. A generic subprogram is a template from which an actual subprogram may be obtained by instantiation. The compiler stores the templates internally. Whenever some code makes use of an instantiation of a template, the complier replaces the formal **type** parameter in the subprogram with the actual type and generates the necessary code.

- Coroutines are special subprograms that, when invoked, execute from the point where they last suspended execution up to the next instruction that later suspends their execution.

- Event handlers subprograms that are invoked implicitly by the occurrence of an event. An event is a condition that requires some immediate action on the part of the program. Such a condition might be an error, a user-generated interrupt, a modification of a specified storage location, etc. When events occur, the event handler is invoked implicitly; that is, without an explicit call. Two different actions are possible on termination of the event handler: resumption of the invoking unit, as with procedures, or termination of the invoking unit.

1.3 Thesis Aim and Outline

The purpose of this thesis is to extend Aldat with procedure facilities. This thesis is divided into five chapters.

- Chapter 1 contains a literature review of database programming languages and subprograms.

- Chapter 2 provides a tutorial on the Relix system, the relational database system developed at McGill University. The chapter also presents the syntax of the statements and commands of Relix that are relevant to the work in this thesis.

- Chapter 3 is the users' manual on procedures which shows the semantic and syntax for procedure definition and invocation, commands on procedures, type of procedure, error messages, and examples.
- Chapter 4 discusses the implementation details of procedures in Relix.
- Chapter 5 concludes the thesis with a summary and proposals for future work.

Chapter 2

Relix Tutorial

Relix, standing for **Rel**ational database on Un**ix**, is a database programming language based on relational and domain algebra [Mer84]. All the design and implementation work in this thesis follows the conceptual framework of existing Relix. The purpose of this chapter is to provide readers with enough Relix background to understand the rest of the thesis. Therefore, we will present only the subset of Relix that is relevant to this thesis.

We will use a tutorial style to present Relix. Simple examples are given during the presentation to illustrate the general format and functionality of the commands and statements. Formal syntax for the commands and statements is also provided for readers to explore the full capability of the language.

We will use the following convention for the examples throughout this thesis.

- The `typewriter` font indicates program output.
- The **boldface** font is used for reserved words in user input which must be typed as it is.
- The *italic* font indicates identifiers in user input which can be substituted by other sequences of characters.

BNF (Backus-Naur Form) notation is used to illustrate the formal syntax throughout this thesis. BNF contains a set of rules. A rule has the form

<rule_name> := definition

which means "assign the definition to the rule_name". The convention of the BNF definition is briefly summarized in Table 2.1.

Form	Meaning
< symbol >	The symbol is a rule name and must be further substituted.
' symbol '	The symbol is a reserved word and must be typed as it is.
{ symbol }	The symbol is optional.
(symbol)*	The symbol may appear zero or more times.
{ symbol)+	The symbol may appear one or more times.
$symbol_1$ | $symbol_2$	Either $symbol_1$ or $symbol_2$ can be used.

Table 2.1: BNF Definition Convention

We begin with Section 2.1 describing how to start Relix. Section 2.2 explains the declarations for domains and relations in Relix. Section 2.3 introduces some basic Relix commands on domains and relations. Relational algebra and domain algebra are discussed in Section 2.4 and Section 2.5 respectively. Section 2.6 introduces the update statements, and Section 2.7 introduces the if-then-else control statement. Finally we will end the chapter by presenting a special form of relation called computation.

2.1 Getting Start

Relix is an interactive command driven parser, interpreter for relational database programming language running on an UNIX operating system. Relix accepts and executes one command or statement at a time. When the user enters a command or statement, Relix checks the syntax of the input using the grammar predefined in the system. If the input is syntactically correct, Relix executes it.

Relix runs on the UNIX operating system. To start Relix with a new or existing database one should use the syntax

'relix' <database_name>

at the UNIX command prompt. For example, the command

relix *sample*

starts a database named "sample". Relix will setup the database and display a welcome message as shown below.

```
setup-database: create a new database "sample"
    relix(g) parser/interpreter (V.4) October 1994
-------------------------------------------------------

>
```

The symbol '> ' is a Relix prompt indicating that Relix is ready to receive input from the user. To display an on-line manual, enter **man!** at the Relix prompt. To quit Relix, enter **q!**.

2.2 Domain and Relation Declarations

Relix deals with two kinds of data objects: domains and relations. A relation is defined on one or more attributes. The domain of a given attribute determines its data type.

Figure 2.1 shows a *Cheque_Book* relation which is defined on four attributes: *Chq#*, *To*, *Amount*, and *Back*. The domain of the *Chq#* attribute is an integer. The domain of the *To* attribute is a string. The domain of the *Amount* attribute is a real number. The domain of the *Back* attribute is a boolean value. In this section, we will use the *Cheque_Book* relation to describe how to declare domains and relations.

```
Cheque_Book:
------------------------------
Chq#    To      Amount     Back
------------------------------
1       Joe     225.00     true
2       Tom     250.00     true
3       Mary    225.00     false
------------------------------
```

Figure 2.1: Cheque_Book relation

2.2.1 Domain Declaration

In order to create a relation, the domains of its attributes must be declared. The declarations

> **domain** *Chq#* **integer ;**

> **domain** *To* **string ;**

> **domain** *Amount* **real ;**

> **domain** *Back* **boolean ;**

declare the attributes of the *Cheque_Book* relation. The first statement declares the domain of an attribute named *Chq#* to be an integer. The second one declares the domain of an attribute named *To* to be a string. Similarly the third and the fourth statement declare the domains of *Amount* and *Back* to be a real number and a boolean value respectively.

Formal Syntax

The formal syntax for domain declaration is

<domain_declaration>	:=	'domain' <attribute_name> <data_type> ';'
<attribute_name>	:=	<identifier>

where:

- <identifier> is a case sensitive combination of characters, digits, underscore (`_`), number sign (`#`), and single quote (`'`). with a maximum length of 80 characters;
- if the *attribute name* already exists in the database and has another *data type*, the system may have two possible actions whereby if the attribute is free[1] the old declaration will be overwritten by the new declaration, otherwise the new declaration is denied; and
- <data_type> is one of the six basic data types summarized in Table 2.2 with the short forms which can be used instead of the usual data type name.

2.2.2 Relation Declaration

The declaration

> \> **relation** *Cheque_Book(Chq#, To, Amount, Back)*;

[1] An attribute is free if it is not used by any relation or virtual attribute. Virtual attribute will be discussed in Section 2.5.

Data Type	Short Form	Domain Values	Examples
Integer	intg	signed integer	32, -445
long	long	signed long integer	2347468, -4
short	short	signed short integer	32711, -234
real	real	signed floating point	3.1416, 0.61E-2
string	strg	sequence of characters	"day", "don't"
boolean	bool	TRUE and FALSE	true, false

Table 2.2: Basic Data Types in Relix

declares the previous *Cheque_Book* relation. We can also declare a relation and initialize it with data at the same time. For example, the declaration statement

> > **relation** *Cheque_Book(Chq#, To, Amount, Back)* **<** **−**
> {(1, "Joe", 225.00, true),
> (2, "Tom", 250.00, true),
> (3, "Mary", 225.00, false)}**;**

declares the *Cheque_Book* relation and initialize with the data shown in Figure 2.1.

The symbol '< −' in the above statement is a Relix assignment operator. New relations and attributes can be created using the assignment operator in a relational algebra statement being discussed in Section 2.4. The statement

$> \quad Chq \;\; <- \; Cheque\_Book;$

creates a new relation named *Chq* and copies the attributes and data from the *Cheque_Book* relation to the *Chq* relation. The statement,

$> \quad Cheque \; [\; Num, \; Name \;\; <- \; Chq\#, \; To \;] \; Cheque\_Book;$

creates two new attributes: *Num* having the same domain as *Chq#*, and *Name* having the same domain as *To*. The new relation *Cheque* is shown below.

```
Cheque:
-----------
Num     Name
-----------
1       Joe
2       Tom
3       Mary
-----------
```

Formal Syntax

The formal syntax for relation declaration is

<relation_declaration>	:=	'relation' <relation_name> '(' <attribute_list> ')'
		{<optional_init_data>} ';'
<relation_name>	:=	<identifier>
<attribute_list>	:=	<attribute_name> (',' <attribute_name>)*

<optional_init_data>	:=	'< −' <relation_value>
<relation_value>	:=	'empty'
		| <relation_name>
		| '{' <constant_tuple_list> '}'
<constant_tuple_list>	:=	<constant_tuple> (',' <constant_tuple>)*
<constant_tuple>	:=	'(' <constant_list> ')'
<constant_list>	:=	<constant_token> (',' <constant_token>)*
<constant_token>	:=	| <constant>

where:

- if the *relation_name* already exists in the database, the old declaration will be overwritten by the new declaration;
- the domains of the attributes in the <attribute_list> must be previously declared in the database;
- <relation_value> can be
  - an empty relation which is a relation with no tuples,
  - an existing relation, or
  - tuples of constant lists enclosed by a pair of curly brackets ({ ...});
- a <constant> can be
  - a signed integer,
  - a signed floating point number,
  - a sequence of characters enclosed in a pair of double quotes (""),
  - a boolean value true or false,
  - a null value 'dc' (don't care) or 'dk' (don't know) which represents irrelevant information and missing data respectively.

2.3 Basic Commands On Domains and Relations

Relix provides users with basic commands to print, show, and delete domains and relations declared in the database.

2.3.1 Print Relation Command

The "print relation" command displays the data associated with a particular relation. The formal syntax is

<print_relation> := 'pr!!'<relation_name>

and the command

> **pr!!** *Cheque_Book*

displays the data of the *Cheque_Book* relation. The display is shown below.

```
------------------------------------------------------------
Chq#     To          Amount      Back
------------------------------------------------------------
1        Joe         225         true
2        Tom         250         true
3        Mary        225         false
------------------------------------------------------------
relation: "Cheque_Book" has "3" tuple(s)
```

2.3.2 Show Commands

The family of "show" commands displays the declaration and system information associated with the relations and domains declared in the database. There are three types of "show" commands: "show relation", "show domain", and "show relation with domain".

Show Relation

The "show relation" command displays the name(s), the number of attributes, the number of attributes sorted, the number of tuples, and the sort status of the relation(s) in the database. The formal syntax is

<show_relation> := 'sr!!'{<relation_name>}

where a *relation name* is optional. Without a *relation name*, the command will display the information for all the relations declared in the database. The command

> **sr!!***Cheque_Book*

displays the information associated with only the *Cheque_Book* relation.

Show Domain

The "show domain" command displays the name(s), the data type(s), and other system information associated with the attribute(s) in the database. The formal syntax is

<show_domain> := 'sd!!'{<attribute_name>}

and the command

> **sd!!***Chq#*

displays the information associated with the *Chq#* attribute.

Show Relation With Domain

The "show relation with domain" command displays the relation name(s), the associated attributes, the attribute positions, and other system information of the relation(s) in the database. The formal syntax is

<show_relation_domain> := 'srd!!'{<relation_name>}

and the command

> **srd!!** *Cheque_Book*

displays the attributes information associated with the *Cheque_Book* relation.

2.3.3 Delete Commands

The family of "delete" commands removes the declaration of a particular relation or attribute from the database. There are two kinds of "delete" commands: "delete relation" and "delete domain".

Delete Relation

The "delete relation" command removes the declaration of a specified relation from the database. The formal syntax is

<delete_relation> := 'dr!!'<relation_name>

and the command

> **dr!!** *Cheque_Book*

deletes the *Cheque_Book* relation from the database.

Delete Domain

The "delete domain" command removes the declaration of a specified attribute from the database. However if the specified attribute is used in any existing relation, the command would fail. Therefore one must delete all the relations associated with the specified attribute before issuing the "delete domain" command. The formal syntax is

<delete_domain> := 'dd!!'<attribute_name>

and the command

> **dd!!***Back*

deletes the attribute named *Back* from the database.

2.4 Relational Algebra Statements

Relational algebra consists of a set of algebraic operations to manipulate relations. Operands and results of any relational algebra operation are always relations. Relational algebra operations can be combined into an expression called a relational expression. The result of a relational expression can be further used in other relational expressions or assigned to a relation object.

In Relix, the relational algebra statement has the formal syntax

<relational_statement> := <assignment_by_name>
| <assignment_by_position>
| <increment_by_name>
| <increment_by_position>

<assignment_by_name> := <relation_name> ‘< −’ <relational_expression> ‘;’

<assignment_by_position> := <relation_name> ‘[’ <attribute_list> ‘< −’ <attribute_list> ‘]’ <relational_expression> ‘;’

<increment_by_name> := <relation_name> ‘< +’ <relational_expression> ‘;’

<increment_by_position> := <relation_name> ‘[’ <attribute_list> ‘< +’ <attribute_list> ‘]’ <relational_expression> ‘;’

and the semantics are given below.

- <assignment_by_name> assigns the values of resulting relation on the right hand side (RHS) of the assignment operator ($<-$) to the relation specified on the left hand side (LHS).

- <assignment_by_position> assigns, by position, the values of the subset of attributes of the resulting relation on the RHS to the set of attributes of the relation specified on the LHS. The attributes in the attribute lists of both sides must have the same domain by position.

- <increment_by_name> adds tuples from the resulting relation on the RHS of the increment operator ($<+$) to an existing relation specified on the LHS. The relations on both sides must be defined on the same set of attributes.

- <increment_by_position> adds tuples from the subset of attributes of the resulting relation on the RHS to the set of attributes of an existing relation specified on the LHS. The attributes in the attribute lists of both sides must have the same domain by position.

- <relational_expression> is a relation name or an expression of relational algebra operations summarized in Figure 2.2.

A detailed discussion on the relational algebra operations can be found in [Mer84].

In this section we will focus on the syntax of the operations and use examples to illustrate the functionalities of some of the commonly used operations.

2.4.1 Unary Relational Operations

Unary relational operations take a relation as an operand and produce a relation as a result. The unary operations are projection and selection.

Projection

Projection is a vertical operation on attributes of a relation. It extracts a subset of the attributes of an operand relation and produces a relation with those attributes. Duplicate tuples in the resulting relation are removed. The formal syntax for projection operation is given below.

<projection>	:=	'[' <project_list>']' 'in' <relational_expression>
<project_list>	:=	| <attribute_list>

For example, the projection expression

$$[\mathit{Course}]\ \textbf{in}\ \mathit{TA}$$

extracts the *Course* attribute from the operand *TA* relation. The *TA* relation and the resulting relation for the above projection expression are shown below.

TA:

```
-----------------
Student    Course
-----------------
Joe        CS102
Mary       CS314
Tom        CS102
Tom        CS243
-----------------
```

[*Course*] in *TA :*

```
------
Course
------
CS102
CS243
CS314
------
```

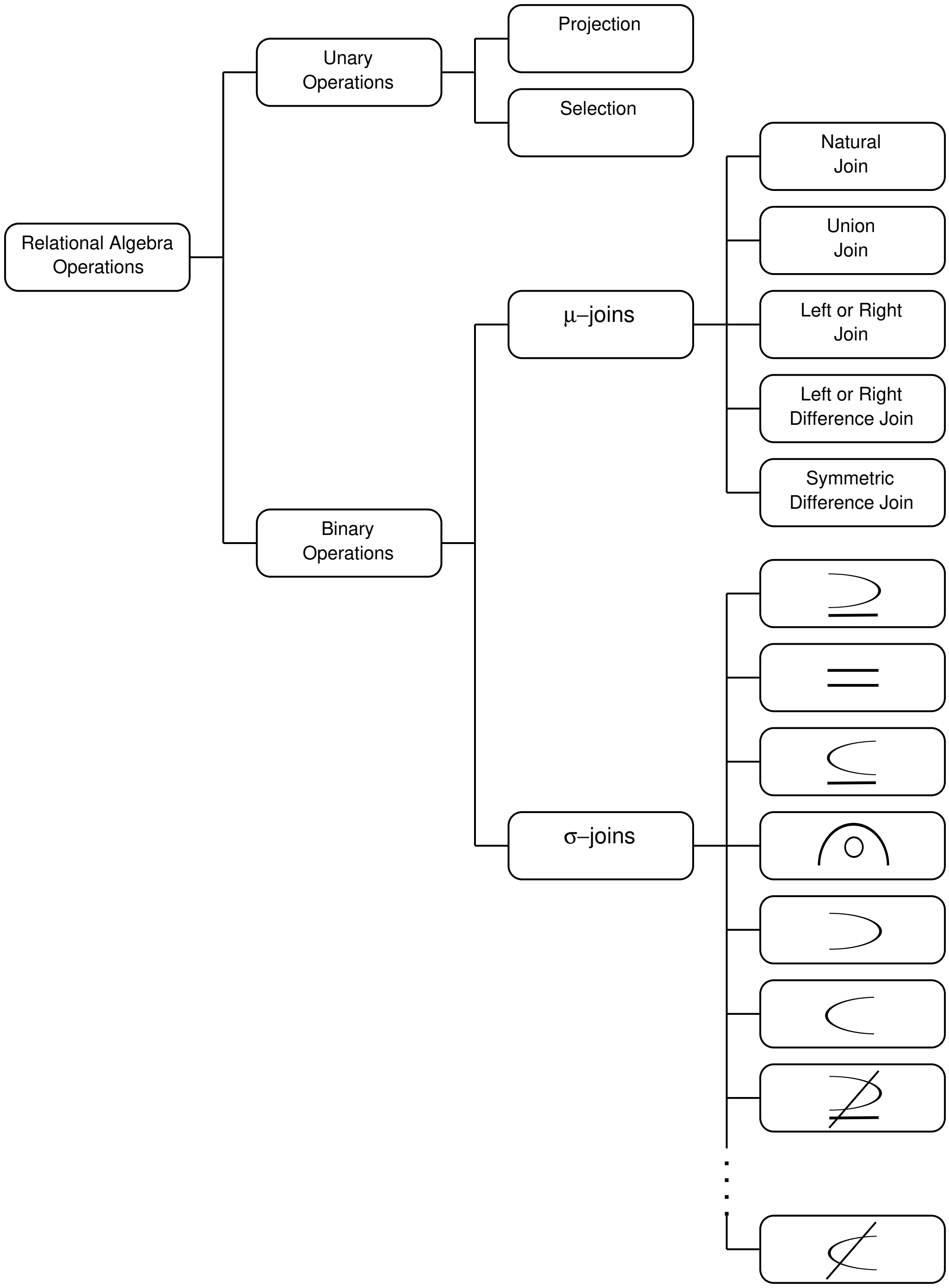


Figure 2.2: Relational Algebra Operations in Relix

If the <project_list> is empty, the resulting relation is a boolean singleton scalar relation[2]. The value in the resulting relation is *true* if the operand relation has tuples. Otherwise the value is *false*. For example, the projection expression

[ ] **in** *TA*

creates a relation below

[ ] in ***TA***:

```
-----
.bool
-----
ture
-----
```

as a result. The attribute '.bool' is a system attribute whose domain is a boolean value.

Selection

Selection is a horizontal operation on tuples of a relation. It extracts the tuples of an operand relation using the condition given in the selection clauses. The formal syntax for selection operation is

<selection> := 'where' <selection_clauses> 'in' <relational_expression>

where <selection_clauses> is a comma separated list of logical domain expressions that can be evaluated horizontally to true or false on each tuple of the operand relation. Domain expression will be discussed in Section 2.5. The selection expression

where *Course* = "*CS102*" **in** *TA*

[2] A scalar relation is a relation defined on only one attribute. A singleton scalar relation is a scalar relation which has only one tuple. A boolean singleton scalar relation is a singleton scalar relation whose value is of type boolean.

extracts the tuples for *CS102* course in the *TA* relation as shown below.

TA:

Student	Course
Joe	CS102
Mary	CS314
Tom	CS102
Tom	CS243

where *Course = "CS102"* **in** *TA :*

Student	Course
Joe	CS102
Tom	CS102

Relix also allows the projection and selection operations to be combined in a single operation called T-selection. T-selection has the formal syntax below.

<T-selection> := '[' <attribute_list> ']' 'where'
<selection_clause> 'in' <relational_expression>

The T-selection expression

$$[Student]\ \textbf{where}\ Course = "CS102"\ \textbf{in}\ TA$$

extracts the *Student* attribute on the tuples having *CS102* course in the *TA* relation as shown below.

TA:

Student	Course
Joe	CS102
Mary	CS314
Tom	CS102
Tom	CS243

[*student*] where
Course = "CS102" **in** *TA :*

Student
Joe
Tom

2.4.2 Binary Relational Operations

Binary relational operations take two relations as operands and produce a relation as a result. In Relix, expressions with binary relational operations have two formal syntaxes.

1. <relational_expression> <binary_operator> <relational_expression>
   This syntax is used for relations which have common attributes. Relix uses the set of common attributes of both operand relations as the set of join attributes in the operation specified by the binary operator.

2. <relational_expression> '[' <attribute_list>
   <binary_operator>
   <attribute_list> ']' <relational_expression>
   This syntax is used for relations which do not have common attributes. Relix pairs the listed attributes from each operand relation, and put the pairs in the set of join attributes. Each pair of attributes must have the same data type.

As shown in Figure 2.2, binary relational operations in Relix can be subdivided into two families: the family of join operations using set operators called μ-joins, and the family of join operations using set comparisons called σ-join.

μ-joins

μ-joins consist of a family of join operations using set operators such as intersection ($\cap$), union ($\cup$), and difference ($-$). The μ-joins on two operand relations are based on three components:

1. center, the combined tuples from both operand relations such that the values of the attributes in the set of join attributes are in the intersection of the values of the set of join attributes of both operand relations;

2. left, the tuples in the left operand relation such that the values of the attributes in the set of join attributes is the difference between the values of the set of join attributes of the left operand relation and the right operand relation; and

3. right, the tuples in the right operand relation such that the values of the attributes in set of join attributes is the difference between the values of the set of join attributes of the right operand relation and the left operand relation .

For example, we have a *TA* relation and an *Office* relation as shown in Figure 2.3. Since *TA* and *Office* relations have the common attribute *Student*, we use the first

TA:

Student	Course
Joe	CS102
Mary	CS314
Tom	CS102
Tom	CS243

Office:

Student	Room
Joe	101
Kim	208
Tom	105
Tom	208

Figure 2.3: TA and Office relation

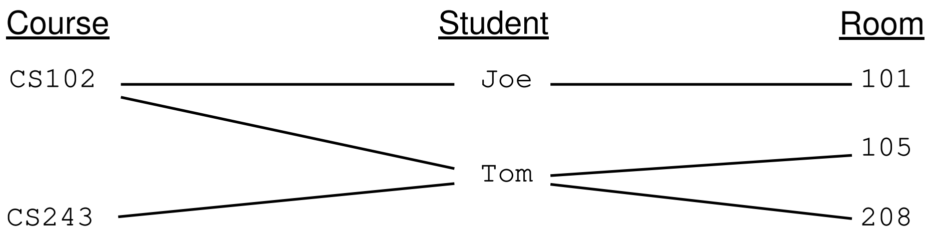
general form for binary relational operations

TA <μ-join-operator> *Office*

with *TA* being the left operand relation, and *Office* being the right operand relation. The set of join attributes is $\{Student\}$. The values of $\{Student\}$ for the left operand relation is $\{Joe,\ Mary,\ Tom\}$. The values of $\{Student\}$ for the right operand relation is $\{Joe,\ Kim,\ Tom\}$. Since

$$\{Joe,\ Mary,\ Tom\} \cap \{Joe,\ Kim,\ Tom\} = \{Joe,\ Tom\}$$

the center component consists of combined tuples of both relations having the value *Joe* or *Tom* in the *Student* attribute. The tuples are combined by connecting everything on one side of the join attribute with everything on the other side as shown in the graph below,



The relation form of the center component is shown below.

centre component of *TA* and *Office*:

Student	Course	Room
Joe	CS102	101
Tom	CS102	105
Tom	CS102	208
Tom	CS243	105
Tom	CS243	208

Since

$$\{Joe,\ Mary,\ Tom\} - \{Joe,\ Kim,\ Tom\} = \{Mary\}$$

the left component consists of the tuple from the *TA* relation having the value *Mary* in the *Student* attribute as shown below.

left component of *TA* and *Office*:

Student	Course
Mary	CS314

Likewise

$$\{Joe,\ Kim,\ Tom\} - \{Joe,\ Mary,\ Tom\} = \{Kim\}$$

the right component consists of the tuple from the *Office* relation having the value *Kim* in the *Student* attribute as shown below.

right component of *TA* and *Office*:

Student	Room
Kim	208

A particular μ-join operation determines which of the three components are combined to produce a resulting relation. Table 2.3 lists the seven μ-joins, their corresponding operators in Relix, and the components of the resulting relations.

The μ-join operator for natural join is *natjoin* or *ijoin*, and the resulting relation of the expression

<relational_expression> **ijoin** <relational_expression>

is the center component of the operand relations. For instance, the expression

TA **ijoin** *Office*

produces a relation as shown below.

TA **ijoin** *Office* :

Student	Course	Room
Joe	CS102	101
Tom	CS102	105
Tom	CS102	208
Tom	CS243	105
Tom	CS243	208

μ–joins	μ–join-operator	Resulting Relation
Natural Join	‘natjoin’ or ‘ijoin’	centre
Union Join	‘ujoin’	left U centre U right
Left Join	‘ljoin’	left U centre
Right Join	‘rjoin’	right U centre
Left Difference Join	‘djoin’ or ‘dljoin’	left
Right Difference Join	‘drjoin’	right
Symmetric Difference Join	‘sjoin’	left U right

Table 2.3: μ-join Operators in Relix and Resulting Relations

The μ-join operator for union join is *ujoin*, and the resulting relation of the expression

<relational_expression> **ujoin** <relational_expression>

is the union of the left, center, and right components of the operand relations. For instance, the expression

TA **ujoin** *Office*

produces a relation as shown below.

TA **ujoin** *Office* :

Student	Course	Room
Joe	CS102	101
Kim	DC	208
Mary	CS314	DC
Tom	CS102	105
Tom	CS102	208
Tom	CS243	105
Tom	CS243	208

Note that the value "DC" in the above relation is a "don't care" null value which means that the value of the attribute is irrelevant.

Similarly, the other five μ-joins on the *TA* and *Office* relations produce the resulting relations that follow below.

TA **ljoin** *Office* :

Student	Course	Room
Joe	CS102	101
Mary	CS314	DC
Tom	CS102	105
Tom	CS102	208
Tom	CS243	105
Tom	CS243	208

TA **rjoin** *Office* :

Student	Course	Room
Joe	CS102	101
Kim	DC	208
Tom	CS102	105
Tom	CS102	208
Tom	CS243	105
Tom	CS243	208

TA **dljoin** *Office* :

Student	Course
Mary	CS314

TA **drjoin** *Office* :

Student	Room
Kim	208

TA **sjoin** *Office* :

Student	Course	Room
Kim	DC	208
Mary	CS314	DC

If the *TA* relation in Figure 2.3 is defined on attribute *Tutor* rather than *Student*, the *TA* and *Office* relations would not have any common attributes. We must use the second general form for binary relational operations, and the join attributes on those two relations must be specified. In our example the form

TA [*Tutor* <binary_operator> *Student*] *Office* pairs the *Tutor* and *Student* attributes as the set of join attributes in the operation. The set of join attributes is now $\{(Tutor,\ Student)\}$. The μ-joins would operate on the values of $\{Tutor\}$ of the left relation and the values of $\{Student\}$ of the right relation. The resulting relations of the μ-joins on these two relations would have an extra attribute *Tutor* whose values would be the same as the *Student* attribute in the same tuple. For instance, the expression

TA [*Tutor* **sjoin** *Student*] *Office*

would produce a relation as show below:

***TA* [*Tutor* sjoin *Student*] *Office*:**

Tutor	Course	Student	Room
Kim	DC	Kim	208
Mary	CS314	Mary	DC

σ-joins

σ-joins consist of a family of join operations using set comparisons : superset ($\supseteq$), equal set ($=$), subset ($\subseteq$), etc. Each operand relation is grouped by the "non-join attributes". For each group in the left relation the set of values of the join attributes is compared against the set of values of the join attributes of every group in the right relation. The particular σ-join determines the set comparison. The values of the non-join attributes of the comparing groups are accepted if the specified set comparison on the join attributes is satisfied.

For example, the σ-joins on the *TA* and *Office* relations, defined in Figure 2.3 ,

TA <σ-join-operator> *Office*

first group the relations by the non-join attributes. Since the join attribute is *Student*, the *TA* relation is grouped by *Course* and the *Office* relation is grouped by *Room* as shown below.

```
TA:                                      Office:
------------------                       ---------------
Course    Student                        Room     Student
------------------                       ---------------
CS102     { Joe  }                       101      { Joe }  Group 1
CS102     { Tom  }  Group 1

CS243     { Tom  }  Group 2              105      { Tom }  Group 2

CS314     { Mary }  Group 3              208      { Kim }
                                         208      { Tom }  Group 3
------------------                       ---------------
```

The set of values of the join attribute in each group is surrounded by a pair of curly brackets ($\{\ldots\}$). For instance, the set of values of the join attribute for the first group in the *TA* relation is {Joe, Tom}, and the set of values of the join attribute for the first group in *Office* relation is {Joe}.

Table 2.4 lists all the σ-joins in Relix with their set comparison and operators. The Relix σ-join operator 'sup' is for superset ($\supseteq$) comparison. The σ-join expression,

TA **sup** *Office*

would first compare the set {Joe, Tom} against the sets {Joe}, {Tom}, and {Kim, Tom}. Since {Joe, Tom} $\supseteq$ {Joe} is true, the values 'CS102' and '101' are accepted for the non-join attributes *Course* and *Room* respectively. Likewise, {Joe, Tom} $\supseteq$ {Tom} is also true and the values 'CS102' and '105' are accepted. The next comparison is {Joe, Tom} and {Kim, Tom }. However, {Joe, Tom} $\supseteq$ {Kim, Tom} is not satisfied and the values 'CS102' and '208' are rejected. The operation continues in the same fashion to compare the rest of the sets in the *TA* relation against the sets in the *Office* relation. The final result of the operation is shown below.

σ-joins	Set Comparison	σ-join Operator
$\supseteq$	Superset	‘div’ or ‘sup’ or ‘gejoin’
$=$	Equal Set	‘eqjoin’
$\subseteq$	Subset	‘sub’ or ‘lejoin’
$\textcircled{0}$	Intersection Empty	‘sep’
$\supset$	Proper Superset	‘gtjoin’
$\subset$	Proper Subset	‘ltjoin’
$\not\supseteq$	Not Superset	‘~sup’
$\neq$	Not Equal Set	‘~eqjoin’
$\not\subseteq$	Not Subset	‘~sub’
$\not\textcircled{0}$	Intersection Not Empty	‘icomp’
$\not\supset$	Not Proper Superset	‘~gtjoin’
$\not\subset$	Not Proper Subset	‘~ltjoin’

Figure 2.4: σ-join Operators in Relix

TA **sup** *Office* :

```
----------------
Course      Room
----------------
CS102       101
CS102       105
CS243       105
----------------
```

Similarly, the other σ-joins on the *TA* and *Office* relations would compare the sets according to the specified set comparison. The resulting relation of the *eqjoin*, *sub*, *sep*, *gtjoin*, and *ltjoin* on those two relations are shown below.

TA **eqjoin** *Office* :

```
----------------
Course      Room
----------------
CS243       105
----------------
```

TA **sub** *Office* :

```
----------------
Course      Room
----------------
CS243       105
CS243       208
----------------
```

TA **sep** *Office* :

```
----------------
Course      Room
----------------
CS243       101
CS314       101
CS314       105
CS314       208
----------------
```

TA **gtjoin** *Office* :

```
----------------
Course      Room
----------------
CS102       101
CS102       105
----------------
```

TA **ltjoin** *Office* :

```
----------------
Course      Room
----------------
CS243       208
----------------
```

The resulting relations for the negated set comparisons *not superset*, *not equal set*, etc., would be the complements of the corresponding set comparisons. For instance, the operator for "intersection empty" is *sep*, and its negation is "intersection not empty" with operator *icomp*. The expression

TA **icomp** *Office*

produces a resulting relation as show below.

TA **icomp** *Office* :

```
----------------
Course      Room
----------------
CS102       101
CS102       105
CS102       208
CS243       105
CS243       208
----------------
```

2.5 Domain Algebra Statements

Domain algebra consists of a set of operations to manipulate attributes such as mathematical operations, grouping attributes, ordering attributes, etc. Domain algebra allows operations over a single tuple (horizontal operations) or over sets of tuples (vertical operations). Figure 2.5 summarizes the domain algebra operations in Relix. A thorough description of domain algebra can be found in [Mer84].

In Relix, domain algebra statements define virtual attributes with formal syntax

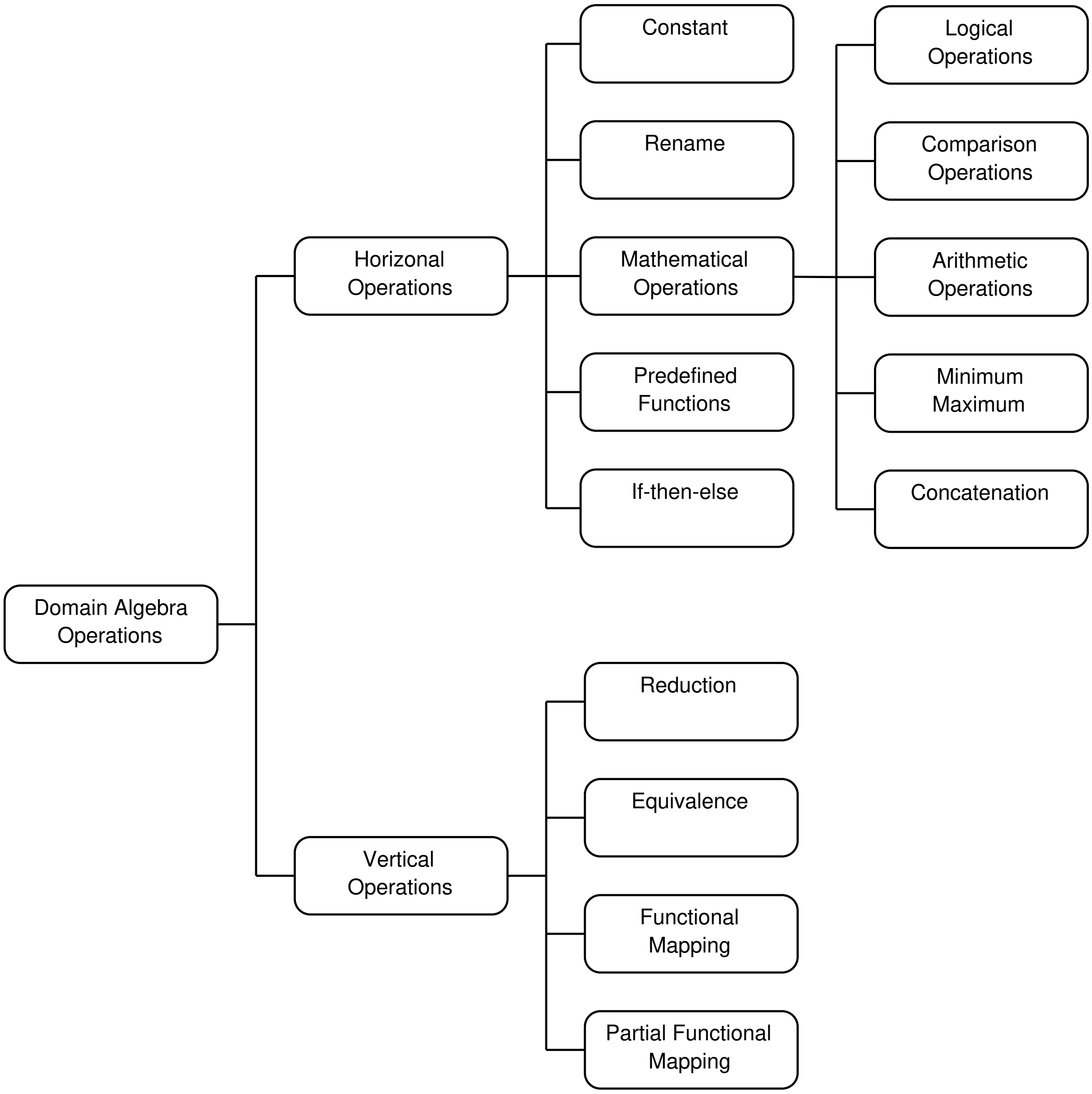


Figure 2.5: Domain Algebra Operations in Relix

<domain_statement> := ‘let’ <attribute_name> ‘be’ <domain_expression> ‘;’

<domain_expression> := <horizontal_operation> | <vertical_operation>

where <horizontal_operation> and <vertical_operation> are expression using horizontal operations or vertical operations which may involve existing actual attributes or other virtual attributes. Horizontal operations and vertical operations will be discussed later in this section.

A domain algebra statement defines a new virtual attribute specified by the <attribute_name>. The definition of the new virtual attribute remains in the database until the user exits Relix. The domain algebra statement

$>$ **let** *Final* **be** *Project + Exam* **;**

declares a virtual attribute named *Final* to be the domain expression *Project + Exam*. The virtual attribute *Final* can be used on any relation containing attributes *Project* and *Exam* as in the following example.

```
Marks:
--------------------
Student Project Exam
--------------------
Joe     30      40
Mary    20      30
Tom     35      45
--------------------
```

Virtual attributes are not associated with any relation until they are actualized by relational algebra statements. To actualize the virtual attribute *Final* we issue a relational algebra statement such as

$>$ *Final_Marks* $<-$ [*Student, Final*] **in** *Marks* **;**

which creates a new relation *Final_Marks* defined on actual attributes *Student* and

Final as shown below.

```
Final_Marks :
----------------
Student    Final
----------------
Joe        70
Mary       50
Tom        80
----------------
```

2.5.1 Horizontal Operations

Horizontal operations work on a single tuple of a relation. The value of a virtual attribute in a tuple is evaluated in terms of the values of the operand attributes in the same tuple. The virtual attribute *Final* already discussed uses a horizontal operation: *Project + Exam*. The value of *Final* in a tuple is the sum of the value of *Project* and the value of *Exam* in the same tuple. Listed below are some domain algebra statements using different types of horizontal operations.

- Constant: **let** *One* **be** *1* ;
- Rename: **let** *Ex* **be** *Exam* ;
- Mathematics: **let** *Profit* **be** *(Price - Cost) * Units* ;
- Predefined Function: **let** *log2* **be** *log(2)* ;
- If-then-else: **let** *Grade* **be if** *Final* > *65* **then** "*Pass*" **else** "*Fail*" ;

Domain expressions with horizontal operations have formal syntax as show below.

<horizontal_operation>	:=	<constant>
		| <attribute_name>
		| <math_expression>
		| '(' <domain_expression>[3]')'
		| <function> '(' <domain_expression> ')'
		| 'if' <domain_expression>
		'then' <domain_expression>
		'else' <domain_expression>
<math_expression>	:=	<unary_opr> <domain_expression>
		| <domain_expression> <binary_opr>
		<domain_expression>
<unary_opr>	:=	'+' | '−' | '!' | 'not'
<binary_opr>	:=	<compare_opr> | <arith_opr> | <logical_opr>
		| 'min' | 'max' | 'cat'
<compare_opr>	:=	'=' | '!=' | '~=' | '<=' | '<' | '>' | '>='
<arith_opr>	:=	'+' | '−' | '*' | '/' | '**' | 'mod'
<logical_opr>	:=	'and' | '&' | 'or' | '|'
<function>	:=	'abs' | 'isknown' | 'round' | 'ceil' | 'floor' | 'sqrt'
		| 'ln' | 'log' | 'log10' | 'acos' | 'asin' | 'atan'
		| 'cos' | 'sin' | 'tan'| 'cosh' | 'sinh' | 'tanh'

[3]It is a recursive definition since domain expression is defined as:

<domain_expression> := <horizontal_operation> | <vertical_operation>

2.5.2 Vertical Operations

Vertical operations process the values of the operand attributes from more than one tuple of a relation. Vertical operations are reduction, equivalence, functional mapping, and partial functional mapping.

Reduction

A simple reduction produces a single result from the values of the specified operand attribute of all tuples in a relation. For example, in

```
>   let Total be red + of Final ;
>   let Cnt be red + of 1 ;
>   Total_Marks <- [Student, Final, Total, Cnt] in Final_Marks ;
```

the first statement declares a virtual attribute *Total* to be the summation of the values of the attribute *Final* in a relation. The second statement declares a virtual attribute *Cnt* to be the number of tuples in a relation. The third statement actualizes the virtual attributes *Total* and *Cnt* to be associated with a new relation called *Total_Marks*. The *Final_Marks* and *Total_Marks* relations are shown below.

Final_Marks :

Student	Final
Joe	70
Mary	50
Tom	80

Total_Marks :

Student	Final	Total	Cnt
Joe	70	200	3
Mary	50	200	3
Tom	80	200	3

The attributes *Total* and *Cnt* have the same value in each tuple. Although such actualization is redundant, it is consistent to the whole approach of the domain algebra. We can also use the result of a reduction in horizontal operations of the domain algebra. For instance, the statements

> $>$ **let** *Average* **be if** *Final* $>$ *Total* / *Cnt*
>
> **then** "*above*" **else** "*below*" ;
>
> $>$ *Average_Marks* **<−** [*Student, Final, Average*] **in** *Final_Marks* ;

actualize the virtual attributes *Average* which uses the results of *Total* and *Cnt* in an horizontal operation. Note that the value of *Average* in a tuple is evaluated in terms of the values of *Total* and *Cnt* in the same tuple. The *Final_Marks* and *Average_Marks* relations are shown below.

```
Final_Marks :
----------------
Student    Final
----------------
Joe        70
Mary       50
Tom        80
----------------
```

```
Average_Marks :
-----------------------
Student Final   Average
-----------------------
Joe     70      above
Mary    50      below
Tom     80      above
---------------------------
```

Instead of producing extra virtual attributes we can combine vertical operations and horizontal operations in a domain expression. We can use the statements

> $>$ **let** *Average* **be if** *Final* $>$ *(red + of Final) / (red + of 1)*
>
> **then** "*above*" **else** "*below*" ;
>
> $>$ *Average_Marks* **<−** [*Student, Final, Average*] **in** *Final_Marks* ;

to produce the same result.

Equivalence

Equivalence is similar to reduction but produces a different result for different classes of tuples in a relation. Instead of calculating from all the tuples in a relation, equivalence operations group tuples into different "equivalence classes" specified by the "grouping" attributes. For example, in

> $>$ **let** *CSum* **be equiv** + **of** *Final* **by** *Course* ;
>
> $>$ *Class_Total_Marks* $<-$ [*Course, Student, Final, CSum*] **in** *Class_Marks* ;

the first statement declares a virtual attribute *CSum* which is the sum of the values of *Final* in each group grouped by *Course*. The *Class_Marks* and *Class_Total_Marks* relations are shown below.

Class_Marks :

Course	Student	Final
CS304	Ann	80
CS304	Peter	65
CS304	Sam	80
CS304	Sue	85
CS612	Joe	70
CS612	Mary	50
CS612	Tom	80

Class_Total_Marks :

Course	Student	Final	CSum
CS304	Ann	80	310
CS304	Peter	65	310
CS304	Sam	80	310
CS304	Sue	85	310
CS612	Joe	70	200
CS612	Mary	50	200
CS612	Tom	80	200

Functional Mapping

Functional mapping processes the tuples in the order of the specified "controlling" attributes. For example, in

> $>$ **let** *OSum* **be fun** + **of** *Final* **order** *Student*;
>
> $>$ *Order_Total_Marks* $<-$ [*Student, Final, OSum*] **in** *Final_Marks* ;

the first statement declares a virtual attribute *OSum* which is the accumulated sum of the values of *Final* in the order of *Student*. The relations are shown below.

Final_Marks :

Student	Final
Joe	70
Mary	50
Tom	80

Order_Total_Marks :

Student	Final	OSum
Joe	70	70
Mary	50	120
Tom	80	200

Partial Functional Mapping

Partial functional mapping performs functional mapping within each equivalence class specified by the "grouping" attributes. For example, in

> \> **let** *PSum* **be par** + **of** *Final* **order** *Student* **by** *Course* ;
>
> \> *Class_Order_Total_Marks* <− [*Student, Final, PSum*] **in** *Class_Marks* ;

the first statement declares a virtual attribute *PSum* which is the accumulated sum of the values of *Final* in the order of *Student* for each group grouped by *Course*. The relations are shown below.

Class_Marks :

Course	Student	Final
CS304	Ann	80
CS304	Peter	65
CS304	Sam	80
CS304	Sue	85
CS612	Joe	70
CS612	Mary	50
CS612	Tom	80

Class_Order_Total_Marks :

Course	Student	Final	PSum
CS304	Ann	80	80
CS304	Peter	65	145
CS304	Sam	80	225
CS304	Sue	85	310
CS612	Joe	70	70
CS612	Mary	50	120
CS612	Tom	80	200

Formal Syntax

The formal syntax for domain expressions with vertical operations is described below.

```
<vertical_operation>          :=  <reduction>
                                  | <equivalence>
                                  | <functional_mapping>
                                  | <partial_functional_mapping>
<reduction>                   :=  ‘red’ <red_opr> ‘of’ <domain_expression>
<red_opr>                     :=  ‘+’ | ‘*’ | ‘&’ | ‘|’ | ‘min’ | ‘max’
<equivalence>                 :=  ‘equiv’ <red_opr> ‘of’ <domain_expression>
                                  ‘by’ <vertical_list>
<vertical_list>               :=  <domain_expression> (‘,’ <domain_expression>)*
<functional_mapping>          :=  ‘fun’ <fun_opr> ‘of’ <domain_expression>
                                  ‘order’ <vertical_list>
<fun_opr>                     :=  <red_opr> | ‘-’ | ‘/’ | ‘mod’ | ‘**’
                                  | ‘cat’ | ‘pred’ | ‘succ’
<partial_functional_mapping>  :=  ‘par’ <fun_opr> ‘of’ <domain_expression>
                                  ‘order’ <vertical_list> ‘by’ <vertical_list>
```

2.6 Update Statements

Update statements are special forms of relational algebra statements. Update operations do not extend the capabilities of relational algebra, but they are easier to use and to understand. In Relix, an update statement has formal syntax

```
<update_statement>  :=  ‘update’ <relation_name> <update_operation> ‘;’
```

where:

- <relation_name> is the name of the “update relation” being updated; and

- <update_operation> is one of the three update operations (add, delete, and change) presented below.

2.6.1 Add

The add operation is equivalent to an increment operation in relational algebra statements. It adds the new tuples in the operand relation to the update relation. The update relation and the operand relation must be defined on the same attributes. For example, there are two relations in the database as shown below.

```
TA:                           NewTA:
-----------------             -----------------
Student     Course            Student     Course
-----------------             -----------------
Joe         CS102             Peter       CS102
Mary        CS314             Sue         CS355
Tom         CS102             -----------------
Tom         CS243
-----------------
```

The update statement

> **update** *TA* **add** *NewTA* **;**

adds the tuples in *NewTA* to *TA*, and the updated *TA* relation is shown below.

```
TA:
-----------------
Student     Course
-----------------
Joe         CS102
Mary        CS314
Peter       CS102
Sue         CS355
Tom         CS102
Tom         CS243
-----------------
```

Below is the formal syntax of add operation.

<update_add> := 'add' <relational_expression>

2.6.2 Delete

The delete operation works like the "left difference join" operation in relational algebra statement. For example, we have the *TA* and *OldTA* relations.

TA:

Student	Course
Joe	CS102
Mary	CS314
Tom	CS102
Tom	CS243

OldTA:

Student
Tom

The update statement

$>$ **update** *TA* **delete** *OldTA* ;

and the relational algebra statement

$>$ *TA* $<-$ *TA* **djoin** *OldTA* ;

produce the same result as as shown below

TA:

Student	Course
Joe	CS102
Mary	CS314

Below is the formal syntax of delete operation.

<update_delete> := 'delete' <relational_expression>

2.6.3 Change

The change operation modifies the values of the specified attributes in selected tuples of the update relation. Domain algebra is used to specify how the values of the specified attributes are changed. For example, the *TA* relation has data

```
TA:
-----------------
Student    Course
-----------------
Joe        CS102
Mary       CS314
Tom        CS102
Tom        CS243
-----------------
```

and the update statement

> **update** *TA* **change** *Course* **<−** *Course cat* "*A* " **;**

adds an "A" to the end of each *Course* in the *TA* relation. The updated *TA* relation is shown below.

```
TA:
-----------------
Student    Course
-----------------
Joe        CS102A
Mary       CS314A
Tom        CS102A
Tom        CS243A
-----------------
```

The above update statement changes all the tuples in the *TA* relation. Changes can also be restricted to selected tuples of the update relation. Relational algebra is used to specify which tuples are to be changed. For example, given the relations

```
TA:                              Fall:
-----------------                ------
Student    Course                Course
-----------------                ------
Joe        CS102                 CS102
Mary       CS314                 CS243
Tom        CS102                 CS256
Tom        CS243                 ------
-----------------
```

the update statement

> **update** *TA* **change** *Course* **<−** *Course cat* "*A* " **using** *Fall* **;**

changes only the tuples in result of the expression

TA **ijoin** *Fall*

or in other words, the above update statement adds an "A" to the end of the *Fall* courses only. The updated *TA* relation is shown below.

```
TA:
-----------------
Student    Course
-----------------
Joe        CS102A
Mary       CS314
Tom        CS102A
Tom        CS243A
-----------------
```

The formal syntax for the change operation is given below.

<update_change>	:=	'change' <change_list> {<optional_using_clause>}
<change_list>	:=	<change_pair> (',' <change_pair>)*
<change_pair>	:=	<attribute_name> '< −' <domain_expression>
<optional_using_clause>	:=	'using' {<μjoin_operator>}<relational_expression>

2.7 If-then-else Statements

The if-then-else statement is a Relix control statement whose execution is to be based on the result of a given condition. The if-then-else statement has formal syntax

<if_statement>	:=	'if' <if_cond> <then_part> { <else_part> } ';'
<then_part>	:=	'then' <statement_part>
<else_part>	:=	'else' <statement_part>
<statement_part>	:=	<one_statement>
		| '{' <one_statement> (';' <one_statement>)* '}'
<one_statement>	:=	<domain_declaration>
		| <relation_declaration>
		| <relational_statement>
		| <domain_statement>
		| <update_statement>

where <if_cond> is a relational expression, which gives a boolean singleton scalar relation as result.

When we discussed projection in Section 2.4.1, we mentioned that the resulting relation of the empty projection expression

[ ] **in** <relational_expression>

was a boolean singleton scalar relation. The above expression can be read as *"some tuples in ... "*. If there are some tuples in the operand relation the returned value is *true*. If the operand relation is empty the return value is *false*. Hence, the empty projection expression can be used as the <if_cond> in the if-then-else statement.

If the returned value in the <if_cond> is true, the statements in <then_part> are executed. Otherwise, the statements in <else_part>, if given, are executed. For example, the statement

> **if** [] **in** *TA*
> **then** *TAO* **< −** *TA* **ijoin** *Office* **;**

will execute the *ijoin* operation if there are some tuples in the *TA* relation.

2.8 Computations

Computations implement procedural abstraction for domain algebra in Relix with the following concepts.

- Computations are symmetrical functions. Whereas a function has specific input parameters and a specific output parameter, a computation has specific parameters, of which any subset can be inputs, with the complement of the subset being output.
- Computations are implicit relations. Whereas an explicit relation contains physical data and finite number of tuples, a computation contains a description of its tuples and possibly infinitely many tuples. Moreover, a computation can be called in s election, 'ijoin', or 'icomp' operation.
- Computations can have internal states where the output of the computation depends not only on the inputs but also on the internal state.

Computations relate to the work of this thesis in two ways:

1. the general forms of procedure declaration conforms to the general form of computation declaration to retain the simplicity of the syntax of the language; and
2. the concept of procedure also includes symmetry.

In this section we will use examples to discuss the general form of computation declaration and the concept of symmetrical functions. The basis of computations can be found in [Mer93] and a complete documentation of its implementation can be found in [Sut94].

2.8.1 Computation Declaration

The formal syntax for computation declaration is

<computation_declaration> := ‘comp’ <computation_name>
‘(’ <parameter_list> ‘)’ ‘is’
{‘def’ optional_predicate_clause ‘;’ }
‘{’ <block> ‘}’
(‘alt’ ‘{’ <block> ‘}’)* ‘;’

and the definitions are given below.

- <computation_name> is a unique identifier among the set of relation and computation names in the database.
- <parameter_list> is a comma-separated list of parameters which must have already been declared as domains.
- <optional_predicate_clause> is an optional boolean expression which gives the relationship among all the parameters. When all the parameters are supplied as

input, the predicate clause is evaluated, and the computation returns a boolean result.

- Each <block> is a single function. It contains statements that evaluate a set of output parameters from the rest of the parameters . A single statement assigns ($<-$) an expression to a given output parameter. An expression is a domain algebra expression or a computation call.

- Blocks are separated by the *alt* keyword. Each block in a computation must be unique regarding the set of the input and output parameters. The curly braces ({ }) around a *block* can be omitted if the block contains only one statement.

For example, $I = P * i$ is a formula for simple interest. I is the amount of interest, P is the amount of principle, and i is the interest rate. The statements

> **domain** *I real*;
> **domain** *P real*;
> **domain** *i real*;
> **comp** *Int* (*I, P, i*) **is**
def $I = P * i$;
$I <- P * i$
alt
$P <- I / i$
alt
$i <- I / P$;

are needed to declare a computation for the above formula. The first three statements declare the attributes *I*, *P*, and *i* with data type real. The fourth statement creates a computation called *Int* with parameters I, P, and i. The computation *Int* contains a p redicate clause and three blocks. Each of the blocks contains only one statement.

2.8.2 Symmetrical Functions

Computations are symmetrical functions because a given computation can have more than one set of input parameters. When a computation is declared, the type of the computation is determined by the union of the type of the predicate clause and the types of the blocks. When a computation is called, Relix searches for the corresponding type, picks up the block of that type, and evaluates the statements in the block.

The type of a block is determined by the sets of input and output parameters of the block. Computation use the convention

$$[\text{input parameters}] - > [\text{output parameters}]$$

to describe the type of the block. For example, the first block of *Int* contains the statement

$$I \;\; <- \; P * i$$

with P and i being the input, and I being the output. Hence,

$$[\text{P, i}] - > [\text{I}]$$

is the type of the block. The "show computation" command with syntax

'sc!!'<computation_name>

can be used to view the type of a particular computation. For example,

> **sc!!***Int*

shows the type of the *Int* computation as shown below.

```
Int (I: real, P: real, i: real)
  [I P i ] -> []
  [P i ] -> [I ]
  [I i ] -> [P ]
  [I P ] -> [i ]
```

Suppose we supply values for I and P in the *Int* computation. Relix will find the type

$$[\mathrm{I\ P}\ ] -> [\mathrm{i}\ ]$$

pick up the block, and evaluate the value of i using the statement

$$i \quad <- \; I \,/\, P$$

in the block. Likewise, when we supply values for P and i, Relix will find the type

$$[\mathrm{P\ i}\ ] -> [\mathrm{I}\ ]$$

and use the statement

$$I \quad <- \; P * i$$

to evaluate the value of I. In case we supply only values for P, Relix cannot evaluate the computation because the type

$$[\mathrm{P}\ ] -> [\mathrm{I\ i}\ ]$$

is not defined in the computation, and an error message is returned.

Chapter 3

Users' Manual on Procedures

This chapter describes how to use procedures in Relix. Section 3.1 explains the basic concept of procedure in Relix. Section 3.2 presents the commands on procedures. Section 3.3 illustrates the formal syntax for procedure declaration and procedure call. Section 3.4 defines the types of procedures. Section 3.5 describes the types of errors generated by procedure declaration, procedure call, and procedure execution. Finally, we end the chapter by showing more practical examples using procedures.

3.1 Basic Concept of Procedure

A procedure is a user-defined object whose declaration has general form

proc <procedure_name> (<formal_parameters>) **is** { <body> } **;**

where

- <procedure_name> is an identifier to be associated with the procedure declaration;
- <formal_parameters> are identifiers used within the body of the procedure which will be replaced by actual parameters when the procedure is activated; and

- <body> consists of a sequence of statements and/or commands described in the previous chapter, which will be executed after the formal parameters are replaced by the corresponding actual parameters.

The basic concept of procedure in Relix covers procedural abstraction, parameter passing, and symmetry.

3.1.1 Procedural Abstraction

As opposed to entering and executing one statement at a time, a procedure associates an identifier to a sequence of statements and awaits for special instruction from the user to start execution. For instance, when the relational algebra statement

> > *TAO* **<-** *TA* **ijoin** *Office*;

is entered, the statement is executed and the relation *TAO* is created. On the other hand, the statement

```
>   proc  iTAO (  ) is
    {
            TAO  <- TA ijoin Office;
    };
```

creates a procedure with name *iTAO* to be associated with the body of the procedure enclosed by a pair of curly brackets ({ ...}). However, the statement in the procedure body is not executed until the procedure is called by the user. Hence, the *TAO* relation does not exist until we issue a procedure call statement

> > *iTAO* ( );

which tells Relix to execute the procedure body associated with the procedure name *iTAO*.

3.1.2 Parameter Passing

The previous *iTAO* procedure does not have any formal parameters. All the identifiers used in the body of the *iTAO* procedure: namely *TA*, *Office*, and *TAO*, are global variables. Global variables are relations or domains which can be accessed outside the procedure.

Formal parameters are identifiers that are used within the body of a procedure, and those identifiers will be replaced by actual parameters before the statements in the body are executed. Consider the procedure declaration

```
>   proc   ijoinTA ( Operand, Result ) is
    {
           Result  <- TA ijoin Operand;
    };
```

that declares a procedure *ijoinTA* with formal parameters *Operand* and *Result*. In the body of the procedure, there are two types of identifiers:

- formal parameters (*Operand* and *Result*), and
- global variable (*TA*).

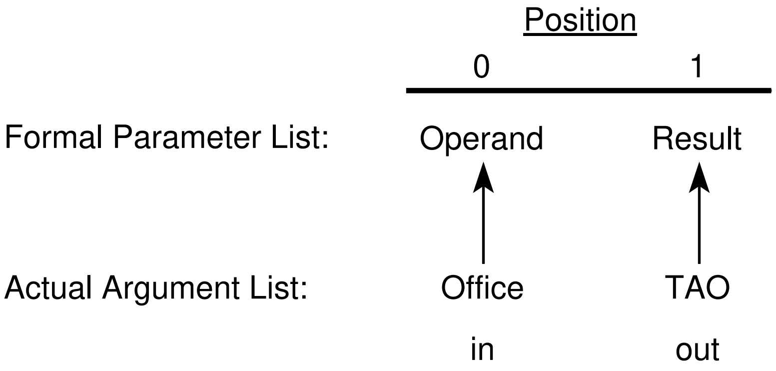
Among the formal parameters, *Operand* is the input to the statement in the procedure body

Result **<** **−** *TA* **ijoin** *Operand*;

and *Result* is the output. When we want to activate the *ijoinTA* procedure, we need to supply the actual parameters for the procedure. The actual parameters are mapped to the corresponding formal parameters by position. Before the body of the procedure is executed, the formal parameters in the procedure body are replaced by the corresponding parameters. The procedure call statement

$$> \quad ijoinTA\ (\ \mathbf{in}\ Office,\ \mathbf{out}\ \ TAO\ )\ ;$$

maps the actual parameter list to the formal parameter list as shown below.



The keywords **in** and **out** in front of each actual parameter indicate whether the corresponding actual parameter is input or output to the procedure. Such indication seems to be redundant here, but it is consistent with the call for symmetric procedures being described later in this section.

The statement in the procedure body

$$Result \quad <- \ TA\ \mathbf{ijoin}\ \ Operand;$$

is changed by replacing the formal parameter *Operand* to *Office*, and *Result* to *TAO*. The resulting statement

$$TAO \quad <- \ TA\ \mathbf{ijoin}\ \ Office\, ;$$

is executed.

Note that this is "pass by name" but not "pass by name and expression". Thus, the call statement

$$\textit{ijoinTA}\ (\ \textbf{in}\ A\ \textit{ijoin}\ B,\ \textbf{out}\ \ TAO\ )\,;$$

is invalid.

We can also supply only a subset of actual parameters. For instance, the procedure call statement

$$>\quad \textit{ijoinTA}\ (\ \textbf{in}\ \textit{Office},\ \textbf{out}\ )\,;$$

changes only the formal parameter *Operand* to *Office*. The statement in the procedure body becomes

$$\textit{Result}\ \ <-\ TA\ \textbf{ijoin}\ \textit{Office}\,;$$

and a relation named *Result* is created as a result of the execution.

Likewise, we can issue a procedure call statement

$$>\quad \textit{ijoinTA}\ (\ \textbf{in},\ \textbf{out}\ )\,;$$

which will not change the statement in the procedure body at all.

3.1.3 Symmetry

Procedures are symmetric because a given procedure may have more than one set of input and output parameters. The user can manipulate the actual parameters to select the block of statements to be executed at each procedure call. Like computations, the type of a procedure is the union of the types of its blocks. The type of a block specifies its input parameters and output parameters. We use the convention

$$[\text{input parameters}] -> [\text{output parameters}]$$

to denote the type of a block (block type). We will further discuss block types in Section 3.4. The block types are then used to determine the appropriate block to execute depending on which parameters are inputs. Consider the procedure declaration

```
>   proc LRU ( Left, Right, Union ) is
    {
            Union  <-   ([Student, Course] in Left) ujoin
                        ([Student, Room] in Right) ;
    } alt
    {
            Left   <-   ([Student, Course] where Course != dc in Union) ;
            Right  <-   ([Student, Room] where Room != dc in Union) ;
    } ;
```

wherein procedure *LRU* has three formal parameters and two blocks. Each block is separated by the keyword **alt**. The first block contains only one statement

```
Union  <-   ([Student, Course] in Left) ujoin  ([Student, Room] in Right) ;
```

with the formal parameters *Left* and *Right* being inputs, and the formal parameter *Union* being the output. Hence,

$$\text{[Left Right]} -> \text{[Union]}$$

is the type of the first block. The second block contains two statements

```
Left   <-   ([Student, Course] where Course != dc in Union) ;
Right  <-   ([Student, Room] where Room != dc in Union) ;
```

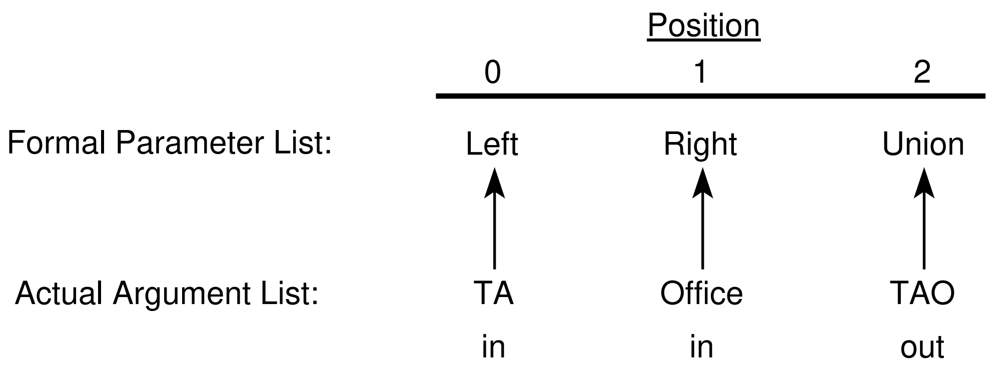
with the formal parameter *Union* being input, and the formal parameters *Left* and *Right* being outputs. Therefore, [Union] − > [Left Right]

is the type of the second block.

When a procedure is called, the system maps the actual parameter list to the formal parameter list, searches for the corresponding type, picks up the block of that type, changes the formal parameters to actual parameters in the block, and executes the block. In the actual parameter list, a keyword **in** or **out** must be supplied in front of each actual parameter to indicate whether the corresponding parameter is input or output. For example, the procedure call statement

> *LRU* (**in** *TA,* **in** *Office,* **out** *TAO*) ;

maps the actual parameter list to formal parameter list as shown below.



The system looks for the type

[Left Right] − > [Union]

and selects the first block. The statement in the first block is changed to

TAO < − ([*Student, Course*] **in** *TA*) **ujoin** ([*Student, Room*] **in** *Office*) ;

and then executed. Similarly, the procedure call statement

> $>$ *LRU* (**out** *TA*, **out** *Office*, **in** *TAO*) ;

maps the actual parameter list to formal parameter list as shown below.

	Position		
	0	1	2
Formal Parameter List:	Left	Right	Union
	↑	↑	↑
Actual Argument List:	TA	Office	TAO
	out	out	in

The system looks for the type

$$[\text{Union}] -> [\text{Left Right}]$$

and selects the second block. The statements in the second block are changed to

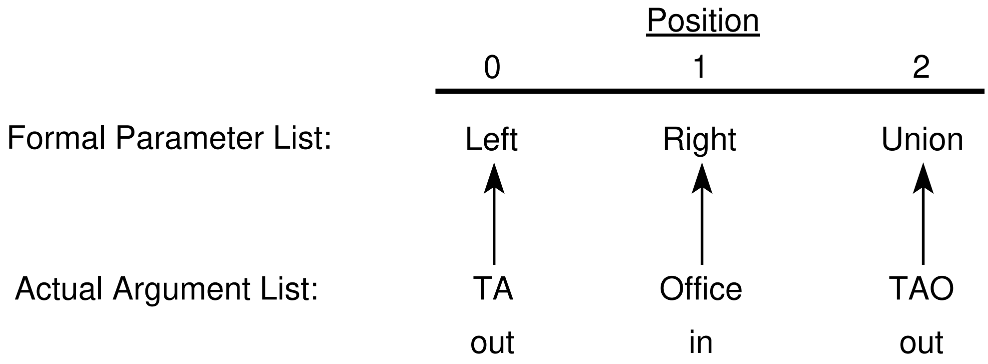
> *TA* < − ([*Student, Course*] **where** *Course* **!= dc in** *TAO*) ;
> *Office* < − ([*Student, Room*] **where** *Room* **!= dc in** *TAO*) ;

and then executed.

The procedure call statement

> $>$ *LRU* (**out** *TA*, **in** *Office*, **out** *TAO*) ;

maps the actual parameter list to formal parameter list as shown below.



The system searches for the type

$$[\text{Right}] -> [\text{Left Union}]$$

which is not defined in the procedure, and an error message is returned. We do not propose mechanisms to enforce consistency of the blocks in a procedure. It is up to the users to write code for each block and to relate them to one another. If a procedure is declared with more than one block of the same type, a warning mess age is displayed, and the corresponding procedure call will pick the first block of that type.

3.2 Commands on Procedures

Commands on procedures are designed to be similar to the ones on relations. The "print relation" (**pr!!**), "show relation" (**sr!!**), and "delete relation" (**dr!!**) commands will take an existing procedure name as argument. A new command "show procedure" (sp!!) is designed to show the type of a specified procedure.

3.2.1 Print

A "print relation" command displays the data associated with a particular relation. For procedures, the "print relation" command displays the source code of a particular procedure. Recall that the syntax for the "print relation" command is

<print_relation> := 'pr!!'<relation_name>

with a relation name as argument. For example, the procedure *LRU* has previously been declared, and the command

> **pr!!***LRU*

displays the source code of *LRU* as shown below.

```
proc LRU (Left, Right, Union) is
{
  Union <- ([Student, Course] in Left) ujoin
           ([Student, Room] in Right);
} alt
{
  Left <- [Student, Course] where Course != dc in Union;
  Right <- [Student, Room] where Room != dc in Union;
};
```

3.2.2 Show

The syntax for the "show relation" command is

<show_relation> := 'sr!!'{<relation_name>}

where a relation name is optional. If an existing procedure name is given as an argument, the "show relation" command displays the associated system information for the specified procedure. For instance, the command

> **sr!!***LRU*

produces a display as shown below.

```
Database: "sample"  (relation ".rel")
  Name              Arity Rank  Ntuples     Sort status
--------------------------------------------------------
  LRU                 0    -3    -3         procedure
```

The fields in the display have the following meanings.

- `Name` is the name of the relation.
- `Arity` is the number of attributes in the relation. Since procedures have no attributes, the value of `Arity` is 0.
- `Rank` is the number of sorted attributes in the relation. Since procedures do not have any attributes, the value of `Rank` is -3 which means that the information is not used.
- `Ntuples` is the number of tuples in the relation. We use this attribute to indicate whether the procedure is being protected from deletion. The value of `Ntuples` is -3 if no one is executing the procedure and the procedure is not protect ed.
- `Sort status` indicates the type of sorting applied for the relation. We use this field to specify that the relation is a procedure.

3.2.3 Delete

The syntax for the "delete relation" command is

<delete_relation> := 'dr!!'<relation_name>

with a relation name as argument. If an existing procedure name is specified as argument, the "delete relation" command removes the declaration and code of the specified procedure from the database provided that the specified procedure is not protecte d. For example, the command

> **dr!!***LRU*

removes the procedure *LRU* from the database if no one is executing the procedure.

3.2.4 Show Procedure

The "show procedure" command displays the formal parameters and the type of a particular procedure. The formal syntax for the "show procedure" command is

<show_procedure> := 'sp!!'<procedure_name>

with an existing procedure name as argument. The command

> **sp!!***LRU*

displays the type of *LRU* with the type of each block in the order of declaration as shown below.

```
LRU (Left, Right, Union)
  [ Left  Right  ] -> [ Union ]
  [ Union  ] -> [ Left  Right ]
```

3.3 Formal Syntax

A procedure declaration or procedure call statement can be

- entered alone at the Relix prompt, or
- used as a statement in the <if_statement> described in Section 2.7.

In this section, we illustrate the formal syntax for procedure declaration and procedure call using BNF notation described in Table 2.1.

3.3.1 Procedure Declaration

Procedure declarations can not be nested. All procedures are defined at the top level and are global to the whole program. The syntax for procedure declaration is designed to be similar to the one for computations. The formal syntax for procedure declaration is show below.

<procedure_declaration>	:=	'proc' <procedure_name>
		'(' {<procedure_parameter_list>} ')'
		'is' <procedure_body> ';'
<procedure_name>	:=	<identifier>
<procedure_parameter_list>	:=	<identifier> (',' <identifier>)*
<procedure_body>	:=	<procedure_block> ('alt' <procedure_block>)*
<procedure_block>	:=	'{' (<procedure_one_statement>)+ '}'
<procedure_one_statement>	:=	<procedure_command>
		| <procedure_statement> ';'

<procedure_command>	:=	<print_relation>
		| <show_relation>
		| <show_domain>
		| <show_relation_domain>
		| <delete_relation>
		| <delete_domain>
<procedure_statement>	:=	<domain_declaration>
		| <relation_declaration>
		| <relational_statement>
		| <domain_statement>
		| <update_statement>
		| <procedure_if>
		| <procedure_call>
<procedure_if>	:=	'if' <if_condition>
		'then' <procedure_statement_part>
		{'else' <procedure_statement_part>}
<procedure_statement_part>	:=	<procedure_statement>
		| '{' <procedure_statement>
		(';' <procedure_statement>)* '}'

where:

- <procedure_name> must be unique among the set of relation, computation, and procedure names in the same database;
- an optional <procedure_parameter_list> is enclosed by a pair of brackets;
- <procedure_parameter_list> is a list of unique identifiers representing the formal parameters of the procedure with a maximum number of 30 formal parameters;

- <procedure_body> contains a sequence of <block> separated by the keyword **alt**;
- <procedure_block> is enclosed by a pair of curly brackets ({ ... }), and contains at least one <procedure_one_statement>;
- <procedure_command> is any command described in Section 2.3;
- <procedure_statement> can be
  - <domain_declaration>, described in Section 2.2.1,
  - <relation_declaration>, described in Section 2.2.2,
  - <relational_statement>, described in Section 2.4,
  - <domain_statement>, described in Section 2.5,
  - <update_statement>, described in Section 2.6.
  - <procedure_if>, similar to the <if_statement> described in Section 2.7 except that only procedure statements are allowed in the <procedure_statement_part>,
  - <procedure_call>, described below.

3.3.2 Procedure Call

Procedure call is a statement with formal syntax:

<procedure_call>	::=	<procedure_name>
		'(' {<procedure_actual_parameter_list>} ')' ';'
<procedure_actual_parameter_list>	::=	<procedure_actual_parameter>
		(',' <procedure_actual_parameter>)*
<procedure_actual_parameter>	::=	'in' {<identifier>}
		| 'out' {<identifier>}

where:

- <procedure_name> is the name of a declared procedure;
- an optional <procedure_actual_parameter_list> is enclosed by a pair of brackets;
- the number of <procedure_actual_parameter> in <procedure_actual_parameter_list> must correspond to the number of formal parameters in the formal parameter list declared with the procedure;
- each <procedure_actual_parameter> must begin with the keyword **in** or **out**, but the <identifier> of the actual parameter can be omitted.

3.4 Type of Procedure

In Section 3.1.3 we define that the type of a procedure is the union of the types of its blocks. The type of a block specifies its input parameters and output parameters which are defined when the procedure is declared. In this section, we present in detail how to define the type of a block.

3.4.1 Inputs and Outputs of Statements in Procedures

A block consists of a sequence of procedure statements. Hence, the inputs and outputs of a block are based on the inputs and outputs of its statements. When defining the inputs and outputs of a statement, we focus on whether the declarations and/or values of the identifiers are required to carry out the execution of the statement. Therefore, we generally define the inputs and outputs of a statement as below.

> If the declaration and/or value of an identifier is required to execute the statement successfully; then the identifier is an input of the statement. Otherwise, the identifier is an output of the statement.

Let us apply the above general definition to each type of statements and commands allowed in procedures.

- Domain Declaration Statements

  Form: **domain** <attribute_name> <data_type> ;

  Inputs: NONE

  Outputs: <attribute_name>

- Relation Declaration Statements

  Form: **relation** <relation_name> (<attribute_list>) { **< −** <init_value> } ;

  Inputs: - All the identifiers in the <attribute_list>

  - All the identifiers in the <init_value>

  Outputs: - <relation_name>

- Relational Algebra Statements

  Assignment: <LHS> **< −** <RHS> ;

  Inputs: - All the identifiers in the <RHS>

  Outputs: - All the identifiers in the <LHS> which are not in the <RHS>

  Increment: <LHS> **<** + <RHS> ;

  Inputs: - All the identifiers in the statement

  Outputs: - NONE

- Domain Algebra Statements

  Form: **let** <attribute_name> **be** <domain_expression> ;

  Inputs: - All the identifiers in the <domain_expression>

  Outputs: - <attribute_name>

- Update Statements

  Form: **update** <relation_name> <update_operations> ;

  Inputs: - All the identifiers in the statement

  Outputs: - NONE

- Basic Commands

  Form: <command_name>**!!**<argument>

  Inputs: - <argument>

  Outputs: - NONE

- Procedure If Statement

  Form: **if** <if_cond> **then** <then_procedure_statement_part>
  { **else** <else_procedure_statement_part> }

  Inputs: - All the identifier in <if_cond>
  - The set of input identifiers required to process <then_procedure_statement_part> in sequential order
  - The set of input identifiers required to process <else_procedure_statement_part> in sequential order

  Outputs: - The set of output identifiers produced by <then_procedure_statement_part> or <else_procedure_statement_part> which are not inputs of the if statement

- Procedure Call

  Form: <procedure_name>**(**<actual_argument_list>**) ;**

  Inputs: - <procedure_name>
  - All the input arguments

  Outputs: - All the output arguments

The table below gives examples for each type of statements allowed in the body of procedures.

Procedure Statement	Inputs	Outputs
domain a integer;		a
relation A (a, b) <- B;	a, b, B	A
A <- [a, b] in B;	a, b, B	A
C <- C ijoin D;	C, D	
E <+ F;	E, F	
let b be a * 10;	a	b
update A change a <- b using B;	A, a, b, B	
pr!!A	A	
dr!!B	B	
if [ ] in A ijoin B	A, B,	
then C <- D ujoin E	D, E	C
else F <- G ujoin H	G, H	F
A (in a, out b, in c);	A, a, c	b

3.4.2 Global Variables and Formal Parameters

There are two types of identifiers in each statement:

1. global variables, the identifiers that **are not** in the formal parameter list; and
2. formal parameters, the identifiers that **are** in the formal parameter list.

We do not use the global variables to define the type of a block because global variables are not associated with procedure call statements. Moreover, we do not require the existence of the input global variables when the procedure is declared. The input global variables can be declared or created any time before the statement is executed. If any input global variable is not available when a statement is executed, the system will generate a severe error message and stop execution of the statement. Errors will be discussed in the next section.

Unlike computations, the formal parameters of a procedure do not need any declaration before the procedure is declared. Formal parameters are neither attributes nor relations. In Relix, an attribute and a relation can have the same identifier. Therefore, we cannot define whether a formal parameter is an attribute or a relation when the procedure is declared. As far as a procedure declaration is concerned, formal parameters are just identifiers, and the scope of the formal parameters is the corresponding procedure body. For example, the procedure declaration

```
>   proc   ProjectTA ( Project ) is
    {
           Result1  <- [Project] in TA ;
           Result2  <- [Room] in Project;
    };
```

is perfectly legal. The first statement uses the formal parameter *Project* as an identifier for an attribute in the *TA* relation, but the second statement use *Project* as an identifier for a relation.

3.4.3 Types of Blocks

Using the set of formal parameters to define the types of blocks, a procedure can have up to $2^{number\ of\ formal\ parameters}$ different combinations of input parameters and output parameters. Hence, to its full potential a procedure can have $2^{number\ of\ formal\ parameters}$ distinct types of blocks.

When defining the type of a block, we focus on whether the declarations and/or values of the formal parameters are required to carry out the sequential execution of the statements within the block. We cannot simply take the union of the input (output) parameters of its statements as the input (output) parameters of the block because:

1. the input of a statement may be produced by the output of the previous statements;
2. the output of a statement may be required as input of the previous statements; and
3. some formal parameters may not be used in the block.

We generally define the input parameters and output parameters of a block as below.

> If the declaration and/or value of a formal parameter is required to execute the statements of block sequentially; then the formal parameter is an input parameter of the block. Otherwise, the formal parameter is an output parameter of the block.

This implies that if a formal parameter is not used in the block, the formal parameter is an output parameter of the block.

The statements within the block are analyzed sequentially. For each statement, the inputs and outputs of the statement are identified. Each input formal parameter of the statement is added to the set of input parameters of the block if it is not in the set of output parameters of the block. Then, each output formal parameters in the statement is added the set of output parameters of the block if it is not in the set of input parameters of the block. After all the statements are analyzed, all the unused formal parameters are added to the s et of output parameters of the block. The algorithm is summarized in Figure 3.1.

```
IB  = the set of input parameters of the block    = empty
OB = the set of output parameters of the block = empty
foreach statement in the block
begin
    Find out the inputs and outputs of the statement
    foreach  I = input formal parameter in the statement
    begin
              if I is NOT in OB
              then
                  Add I to IB
    end for
    foreach  O = output formal parameter in the statement
    begin
              if O is NOT in IB
              then
                  Add O to OB
    end for
end for
Add all the unused formal parameters to OB
```

Figure 3.1: Algorithm for Computing the Type of a Block

3.4.4 Examples

The examples below show how the algorithm in Figure 3.1 works. To represent the type of a block, we use the convention

$$[\text{ IB }] -> [\text{ OB }]$$

where IB is the set of input parameters of the block, and OB is the set of output parameters of the block. The inputs and outputs of each statement are shown in the column "Statement Inputs Outputs". "Block Type" is updated after the analysis of eac h statement. "Procedure Type" is the union of the block types.

Procedure Declaration	Statement Inputs	Statement Outputs	Block Type	Procedure Type
proc Ex1 (A, B) is				
{				
let A be 1;		A	[ ] -> [A]	
let B be red+ of A;	A	B	[ ] -> [A B]	
} ;				[ ] -> [A B]

The procedure *Ex1* above shows that A is the input of the second statement but the output of the block.

Procedure Declaration	Statement Inputs	Statement Outputs	Block Type	Procedure Type
proc Ex2 (A, B) is				
{				
dd!!A;	A		[A] -> [ ]	
let A be red+ of B;	B	A	[A B] -> [ ]	
} ;				[A B] -> [ ]

The procedure *Ex2* shows that A is the output of the second statement but the input of the block.

Procedure Declaration	Statement Inputs	Statement Outputs	Block Type	Procedure Type
proc Ex3 (A, B, C) is { A <+ B; } ;	A, B		[A B] -> [ ]	[A B]->[C]

The procedure *Ex3* shows that the unused formal parameter *C* is added to the set of output parameters of the block.

Procedure Declaration	Statement Inputs	Statement Outputs	Block Type	Procedure Type

proc Ex4 (A, B) is
{
B <- R ijoin A;
}
	R, A	B	[A] -> [B]	[A] -> [B]

alt { A <- B ijoin S; } ;	B, S	A	[B] -> [A]	[B] -> [A]

The procedure *Ex4* shows that global variables, *R* and *S*, are ignored when computing the types of the blocks. The union of the types of its blocks

$$[\,\mathrm{A}\,] -> [\,\mathrm{B}\,]$$
$$[\,\mathrm{B}\,] -> [\,\mathrm{A}\,]$$

is the type of procedure *Ex4*.

3.5 Errors

When something goes wrong, the system reports an error message. There are two kinds of error messages: severe error messages and warning error messages.

Severe errors are usually syntax errors or run time errors. After displaying a severe error message, the system stops executing the current statement and goes to the next statement. A severe error message has general format

*** SEVERE ERROR ***
<function_name>: <message>

where

- <function_name> is the name of the function that issues the error message, and
- <message> describes the error.

A warning error message is usually caused by minor semantic errors. After displaying a warning error message, the system continues the execution of the current statement and attempts to produce the outputs of the statement. A warning error message has general format given below.

*** WARNING ERROR ***
<function_name>: <message>

In this section, we describe the types of severe and warning error messages, together with their causes, that may occur at procedure declaration, procedure call, and procedure execution. An alphabetical list of error messages described in this section can be found in Appendix B.

3.5.1 Procedure Declaration

When the user enters a procedure declaration, the system will not create the procedure declaration if any of the following severe error messages is displayed.

- Invalid token: "..."

  An invalid token is encountered when the system is parsing the user input. The user should check the syntax of the source code using the formal syntax described in Section 3.3.

- Domain "..." already appeared in this list

  A Duplicated identifier is found in the formal parameter list. The user should ensure that the identifiers in the formal parameter list are unique.

- Name "..." has been used

  The procedure name has been used by other procedures, relations, or computations. The "show relation" command (**sr!!**) can be used to display all the used names.

- "..." has too many parameters

  The procedure declaration has more than 30 formal parameters.

However, the system will create the procedure declaration even if any of the following warning error messages is displayed.

- Procedure "..." has more than one block

  The procedure has no formal parameters and attempts to declare more than one block. A procedure with no parameters should have 2^0 block. When the procedure is called, only the first block will be executed.

- Block "..." and block "..." have the same type

  The procedure declares two blocks with the same type. Each block in a procedure should have a unique type. When the procedure is called, only the first block of that type will be executed.

3.5.2 Procedure Call

When the user enters a procedure call statement, the system will not execute the procedure if any of the following severe error messages is displayed.

- Invalid token: ”...”

  An invalid token is encountered when parsing the procedure call statement.

- Cannot find "..."

  The procedure name given in the procedure call statement is not declared.

- Cannot execute "..."

  The procedure name given in the procedure call statement is not a procedure.

- Too many arguments in "..."

  The procedure call statement has more actual arguments than the number of formal parameters declared. The command **pr!!<procedure_name>** or **sp!!<procedure_name>** can be used to check the declared formal parameters of the procedure.

- Not enough arguments in "..."

  The procedure call statement has less actual arguments than the number of formal parameters declared.

- No such type in "..."

  The procedure call statement attempts to call an undefined block. The "show procedure" command (**sp!!<procedure_name>**) can be used to find out the defined types of blocks of the procedure.

3.5.3 Procedure Execution

When the system is executing the procedure statements in the selected block, the system will stop executing the current procedure statement if any of the following severe error messages is found.

- Procedure "..." is protected. Deletion failed

  A "delete relation" command (**dr!!<procedure_name>**) attempts to delete a caller procedure. A delete command will remove all the information and codes associated with the procedure. The system cannot locate the codes when the control is returned to a deleted procedure. Hence, we protect all the caller procedures from being deleted.

- Cannot execute "..."

  A statement attempts to use a procedure name in a relational expression.

- Cannot find "..."

  A statement attempts to use an undeclared object as input. All the inputs to a statement must be declared before the statement is executed.

- Domain "..." still used; cannot change type

  A domain declaration or domain algebra statement attempts to re-declare an existing attribute with another data type. Relix does not allow an attribute to change its data type if the attribute is used by any relation or virtual attribute.

- Domain "..." still used

  A "delete domain" command (**dd!!domain_name**) attempts to delete an existing attribute which is used by a relation or virtual attribute.

- Type mismatch

  A domain expression attempts to operate on two attributes of incompatible data type.

- No common attributes

  A relational algebra binary operation, except natural join and union join, attempts to operate on two relations with no common attributes, but the set of join attributes is not stated explicitly in the expression. The set of join attributes must be explicitly listed in the binary operation expression for relations with no common attributes.

- Attributes of "..." and "..." are not same

  A relational increment or update add statement attempts to add two relations defined on different attributes.

- If_cond is not a boolean singleton scalar

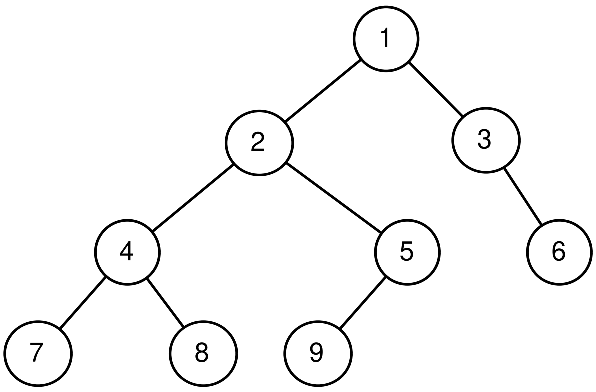
  The if_cond expression in the if statement does not produce a boolean singleton scalar.

3.6 More Examples

In this section we will present some practical examples using procedures.

3.6.1 Recursive Procedure

A tree can be represented by a relation defined on two attributes. For example,




```
Tree:
-----------------
Parent    Child
-----------------
  1         2
  1         3
  2         4
  2         5
  ...

  5         9
-----------------
```

the tree on the left can be stored as the *Tree* relation on the right. Each tuple in the *Tree* relation represents a link in the tree. To find out all the paths of a tree we can use a procedure:

> **proc** *Path* (*All, Tree, Parent, Child*) **is**
> {
> *temp* $<-$ *Tree* ;
> *All* $<-$ *Tree* ;
> *Path*(**in** *All,* **in** *Tree,* **in** *Parent,* **in** *Child*) ;
> } **alt**
> {
> *temp* $<-$ *temp* [*Child* **icomp** *Parent*] *Tree* ;
> **if** [] **in** *temp*
> **then**
> {
> *All* $<-$ *All* **ujoin** *temp* ;
> *Path*(**in** *All,* **in** *Tree,* **in** *Parent,* **in** *Child*)
> } ;
> } ;

The first block initializes the *temp* and *All* relations and calls the second block which performs the recursive operation. Note that the values of *temp* and *All* are updated with each execution of the second block. We can find out the type of the *Path* procedure

```
Path (All, Tree, Parent, Child)
  [ Tree  Parent  Child  ] -> [ All  ]
  [ All  Tree  Parent  Child  ] -> [ ]
```

using the show procedure command (**sp!!***Path*), The procedure call statement

> *Path*(**out** , **in** , **in** , **in**) **;**

starts the execution of the first block without changing the formal parameters in the first block. The solution of the *Path* procedure can be applied to other relations of the same type. For example,

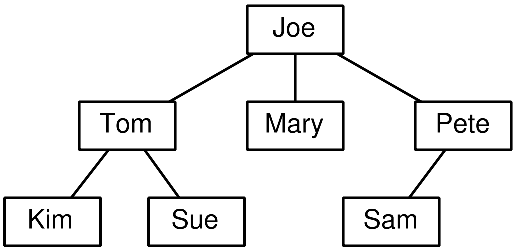


Chart:

Senior	Junior
Joe	Tom
Joe	Mary
Joe	Pete
Tom	Kim
Tom	Sue
Pete	Sam

the organization chart on the left is represented by the *Chart* relation on the right. The procedure call statement

> *Path*(**out** *AllChart* , **in** *Chart,* **in** *Senior,* **in** *Junior*) **;**

computes all the paths in the chart.

3.6.2 Event Handler

Event handlers are user defined procedures with special names. A complete documentation on event handlers in Relix can be found in [EK96]. Here we just give a simple illustration.

Event handlers are called by the system whenever the defined events occur. Suppose we want to recalculate all the paths whenever new tuples have been added to the *Chart* relation. We can define an event handler

> **proc** *post:add:Chart* **( ) is**

{

Path(**out** *AllChart* , **in** *Chart,* **in** *Senior,* **in** *Junior*) **;**

} ;

which will be called by the system automatically after the event

update *Chart* **add** ... **;**

is executed.

The name of an event handler has the formal syntax

<event_name>	:=	{<prefix>':'}<type>':'<relation_name>{'['<attribute_list>']'}
<prefix>	:=	'pre' | 'post'
<type>	:=	'add' | 'delete' | 'change'

where:

- 'pre' indicates that the event handler is to be executed prior to the execution of the event which triggered it;
- 'post' indicates that the event handler is to be executed after the triggering event has been executed; and
- if <prefix> is omitted then 'post' is assumed.

3.6.3 Domain Algebra Definition

In Relix, the definitions of virtual attributes are not stored permanently in the database. These definitions will be lost after Relix is terminated. Procedures can be used to restore the virtual attributes. For example, the *sumt* procedure declaration

> $>$ **proc** *sumt* ( ) **is**
>
> {
>
> **let***sumt***be** *red+ of t***;**
>
> } ;

defines a virtual attribute *sumt*. Since procedures are stored permanently in the database, calling the procedure

> $>$ *sumt*() ;

will restored the definition of *sumt*.

Chapter 4

Implementation of Procedures

This chapter is about the implementation of procedures. Section 4.1 overviews the implementation of Relix. Section 4.2 describes how procedures are represented. Section 4.3 and Section 4.4 present the implementation of procedure declaration and procedure call respectively. Section 4.5 presents the implementation of commands on procedures.

4.1 Implementation of Relix

Relix is an interactive multi-user system. It is written in C programming language, and is portable across different platforms running the UNIX operating system. Extensions in Relix require that the modules to be added are compatible with the existing code. Therefore, in this section we overview the implementation of Relix that is related to the work of this thesis. A complete documentation for its first implementation can be found in [Lal86].

4.1.1 System Relations

A relation is stored in a UNIX file whose name corresponds to the name of the relation. A database, which is a collections of relations, is equivalent to a UNIX directory. Every Relix database maintains a set of system relations which represents the data dictionary of the database and is stored permanently as UNIX hidden files.[1] Three basic system relations are used to store information about domains and relations in the database.

1. *.rel (.rel_name, .sort_status, .rank, .ntuples)*[2]

   The *.rel* system relation stores information about all the relations in the database.

   - .rel_name is the name of the relation,
   - .sort_status specifies the type of sorting for the relation,
   - .rank is the number of sorted attributes in the relation, and
   - .ntuples is the number of tuples in the relation.

2. *.dom (.dom_name, .type)*

   The *.dom* system relation stores information about all the domains in the database.

   - .dom_name is the name of the domain, and
   - .type is the data type of the domain.

3. *.rd (.rel_name, .dom_name, .dom_pos, .dom_count)*

   The *.rd* system relation stores information that links the relations with the domains on which they are defined.

[1] File names beginning with a period (.) are UNIX hidden files which are not normally listed under the UNIX list directory command.

[2] In Relix convention, names begin with a period (.) are system names.

- .rel_name is the name of the relation,
- .dom_name is the name of the domain,
- .dom_pos is the position of the domain in the relation, and
- .dom_count is the number of domains in the relation.

Computation uses five system relations to store the interface information for all the computations declared in the database. Two of them are related to the work of this thesis.

1. *.comp (.comp_name, .dom_pos, .dom_name, .type, .seq_attr)*

   The *.comp* system relation contains domain information of computations. The attributes are:

   - .comp_name,the name of the computation;
   - .dom_pos, the position of the domain in the computation;
   - .dom_name, the name of the domain;
   - .type, the data type of the domain; and
   - .seq_attr, the sequencing attribute status of the domain.

2. *.comp_type (.comp_name, .block, .block_type, .code_offset)*

   The *.comp_type* system relation contains information on types of computations. The attributes are:

   - .comp_name, the name of the computation;
   - .block, the sequence number of the block;
   - .block_type, the type of the block; and
   - .code_offset, the offset of intermediate code in the computation I-code file.

4.1.2 Lexical Analyzer, Parser, and Interpreter

Relix consists of three main modules: a lexical analyzer, a parser, and an interpreter. The lexical analyzer is generated by LEX [Les75] to scan the user input into tokens. The parser is generated by YACC [Joh75] to perform syntax analysis and generate intermediate code (I-code) for the input. The intermediate codes are generated by invoking a translator function with different parameters when a component of a sentence is recognized. Simulating a stack machine, the interpreter is a C function which reads instructions from the I-code and calls other C functions to perform the operations. Figure 4.1 summarizes the main flow of Relix.

An example for processing one statement will be shown. When the statement:

$$> \quad R \; <- S \; ;$$

is entered by the user, the lexical analyzer scans the input and matches the input into tokens specified in the LEX token description file. For example,

```
([a-z]|[A-Z]|[0-9]|[_'#])+   {return(IDENTIFIER);}
"<-"                         {return(ASSIGN);}
```

specifies the regular expressions for the tokens IDENTIFIER and ASSIGN respectively. Literal characters such as the semi-colon are also passed through the lexical analyzer and are also considered tokens. The tokens

IDENTIFIER ASSIGN IDENTIFIER ';'

are for the example statement. The parser performs syntax analysis and finds that the above stream of tokens represents a sentence of an <assignment_by_name> expression described in Section 2.4 with the following YACC rules:

```
assignment_by_name:
     name_valued_expression ASSIGN
```

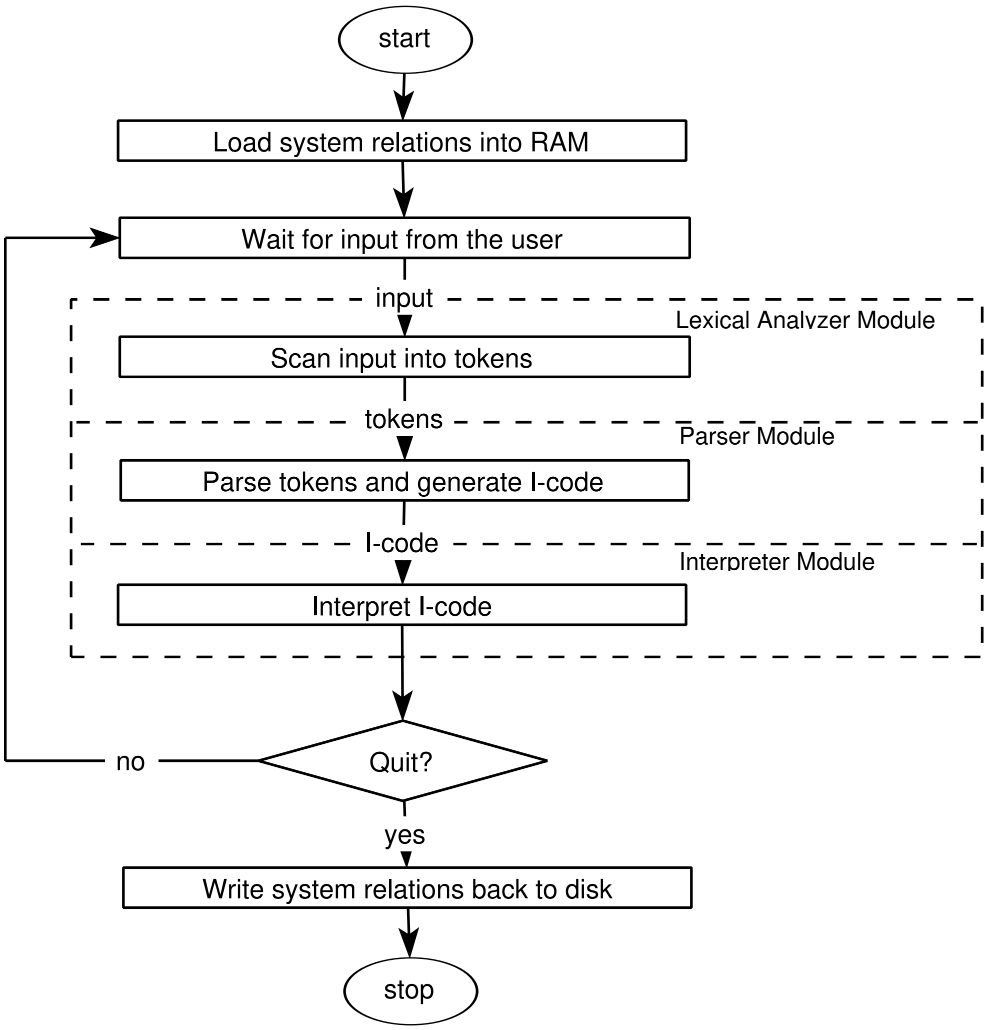


Figure 4.1: Relix Execution Flowchart

```
          relational_expression {translator(ASSIGN_SCALAR);} ';'
          ;
     name_valued_expression:
          identifier {strcpy( yytext, "name"); translator( IDENTIFIER);
                       translator( STRING); translator( CONSTANT_RELATION);}
          ;
     relational_expression:
          ...
          | identifier
          ;
     identifier:
          IDENTIFIER {translator(IDENTIFIER);}
          ;
```

where actions in YACC are C codes enclosed in a pair of curly brackets. The translator function is a C function which performs various tasks according to the actual parameter. The tasks of the translator function include:

- maintaining a scalar_stack for storing and retrieving identifiers;
- maintaining a set of flags and counters; and
- generating I-code.

For instance, the call

translator(IDENTIFIER);

pushes the value of the identifier onto the scalar_stack. Some of the parameters produce I-code. For example

parameter	I-code
STRING	`push-name x name`
CONSTANT_RELATION	`constant-relation`
ASSIGN_SCALAR	`push-name x assign-scalar`

where x is a string obtained by popping an item from the scalar_stack. The I-code for the example statement is shown below.

```
push-name          /* Push the next string onto the stack.*/
R
name
constant-relation  /* Call function constant_relation to
                      create a new relation using the name
                      on the stack.                      */
push-name
S
assign-scalar      /* Pop item A and B from the stack, and
                      call function assign_scalar to
                      assign item A to item B.           */
halt               /* Update system relations and return. */
```

The comments on the right hand side describe the interpreter actions for the corresponding I-codes. The interpreter maintains a stack for the purpose of storing and retrieving operands. `push-name` pushes an operand onto the stack. The '`constant-relation`' and '`assign-scalar`' are C functions that the interpreter needs to call with predefined arguments which are obtained by popping the operands from the stack. Note that '`halt`' is required at the end of the I-code for the interpreter to stop execution.

4.2 Representation Of Procedures

Procedures are implemented as special forms of relations because procedures are similar to computations which are special forms of relations. A procedure is represented as three components: a source file, an I-code file, and interface information. We will use the following procedure declaration to illustrate the representation.

$>$ **proc** *Sam* (*R, S*) **is**

{

S $<-$ R;

} **alt**

{

R $<-$ S;

};

4.2.1 Source File

A source file is a UNIX text file of the same name as the procedure. This contains the source code of the procedure and is kept only for the purpose of human reading. It does nothing in the execution process of the procedure. We generate a source file by storing all the characters that users type in when they declare a procedure. The source file of the procedure *Sam* is presented below.

```
proc Sam (R, S) is
{
     R <- S;
} alt
{
     S <- R;
};
```

4.2.2 I-code File

An I-code file is a UNIX text file with the name ".<procedure_name>.proc". This file contains intermediate codes of the procedure body and plays an important role in the execution process of the procedure. The I-code file of the *Sam* procedure is ".Sam.proc" containing I-code that follows below.

```
push-name
R
name
constant-relation
push-name
S
assign-scalar
p-stmts-delim      /* Procedure statement delimiter */
halt
push-name
S
name
constant-relation
push-name
R
assign-scalar
p-stmts-delim
halt
```

Note that the I-code file contains statements for all the blocks separated by the I-code 'halt'.

4.2.3 Interface Information

The interface information of a procedure includes its declaration, formal parameters, and type. Interface information is kept in the form of tuples in three system relations.

1. Each procedure has its declaration entry in relation *.rel*. The *.sort_status* of any procedure is set to a constant PROC_STATUS, which equals 32 in the current version, to be distinct from other types of relations. The *.rank* and *.ntuples* are set to RELIX_VOID, which equals -3, to indicate the unused state. The entry

.rel	(.rel_name,	.sort_status,	.rank,	.ntuples)
	Sam	32	-3	-3

   is for procedure *Sam*.

2. Each procedure having formal parameters has its formal parameter information in relation *.comp*. The *.comp_name* is the name of the procedure. The *.dom_pos* is the position of the formal parameter starting from 0. The *.dom_name* is the name of the formal parameter. The *.type* of any formal parameter is set to a domain constant *any*, which equals 6, to indicate that it can be any type. The *seq_attr* of any formal parameter is set to 0 to indicate unused state. The entries

.comp	(.comp_name,	.dom_pos,	.dom_name,	.type,	.seq_attr)
	Sam	0	R	6	0
	Sam	1	S	6	0

   are for procedure *Sam*.

3. Each procedure having formal parameters has its type information in relation *.comp_type*. The *.block* is the number of the block starting from 0. The *.block_type* is a long integer (32 bits) which represents the type of the corresponding block. The bit corresponding to the position of input formal parameters is marked. Hence, output formal parameters are the complement of input formal parameters. For example, the positions of R and S are 0 and 1 respectively. The procedure type is shown below.

$$[\ S\ ] -> [\ R\ ]$$
$$[R\ ] -> [\ S\ ]$$

The *.block_type* of block 0 is $2^1 = 2$, and that of block 1 is $2^0 = 1$. The *.code_offset* indicates the offset of intermediate codes in the I-code file. The entries

.comp_type	(.comp_name,	.block,	.block_type,	.code_offset)
	Sam	0	2	0
	Sam	1	1	80

are for procedure *Sam*.

4.3 Declare

In this section, we present the implementation for procedure declaration. We first describe the I-code for procedure declaration. Then the algorithm to declare a procedure is presented.

4.3.1 I-code for Procedure Declaration

When users declare a procedure, the lexical analyzer scans the input into a stream of tokens. Using the formal syntax described in Section 3.3.1, we add the rules for procedure declaration to the existing parser module. Appendix A lists all the required addition to the existing parser. The parser checks the syntax of the declaration statement, detects duplicated identifiers in the formal parameter lists, and translates the statement into I-code. Below is the structure of the I-code for procedure declaration.

```
push-name       /* Push the next string onto the stack.*/
...             /* first formal parameter             */
push-name       /* more formal parameters             */
...
```

```
...
push-count
...             /* number of formal parameters         */
push-name
...             /* procedure name                      */
p-dec           /* beginning of procedure declaration  */
p-blockbgn      /* beginning of a block                */
push-name-left
...             /* the input identifier of a statement */
push-name-right
...             /* the output identifier of a statement*/
...             /* operator of a statement  or command */
p-stmts-delim   /* statement delimiter                 */
...             /* more statements                     */
...
...
p-blockend      /* end of a block                      */
...             /* more blocks                         */
...
...
p-dec-off       /* end of procedure declaration        */
```

To examplify this, the I-code for the declaration of procedure *Sam* described in Section 4.2 is provided below.

```
push-name
R
push-name
```

```
S
push-count
2
push-name
Sam
p-dec
p-blockbgn
push-name-left
R
name
constant-relation
push-name-right
S
assign-scalar
p-stmts-delim
p-blockend
p-blockbgn
push-name-left
S
name
constant-relation
push-name-right
R
assign-scalar
p-stmts-delim
p-blockend
p-dec-off
halt
```

4.3.2 Algorithm to Declare a Procedure

When the interpreter meets the instruction '`p-dec`' in the above I-code, it calls the function `declare_proc` using the procedure name (P_name) and the formal parameter list (P_list) as arguments. The algorithm of the function declare_proc is shown below.

1. If P_name is in system relation .rel, report error and return.

2. If the number of formal parameters exceeds 30, report error and return.

3. Insert the procedure entry into the system relation .rel. as described in Section 4.2.3.

4. Create a source file for the procedure as described in Section 4.2.1.

5. Create an I-code file for the procedure as described in Section 4.2.3.

6. If the procedure has no formal parameters, write the I-code within the first block (p-blockbgn ... p-blockend) to the I-code file. If there is more than one block, issue a warning and skip the rest of the I-code.

7. If the procedure has formal parameters:

   (a) for each formal parameter in P_list create an entry in the system relation .comp as described in Section 4.2.3;

   (b) for each block

      i. compute the type of the block using the algorithm in Figure 3.1,

      ii. if the type of the block already exists in the system relation .comp_type, issue a warning,

iii. create an entry for the block in the system relation .comp_type as described in Section 4.2.3,

iv. write the I-code within each block (p-blockbgn . . . p-blockend) to the I-code file,

v. at the end of each block, add the I-code 'halt' to the I-code, and

vi. update the offset of the block in the system relation .comp_type.

8. Write the source program into the source file.

9. Return.

If the interpreter receives any error from the function `declare_proc`, it will skip the I-code until '`p-dec-off`' or '`halt`'. In this case, the system relations are not updated.

4.4 Call

In this section, we describe the implementation of the procedure call. We first look at the I-code for procedure call. Then we present the algorithm.

4.4.1 I-code for Procedure Call

The structure of the I-code for procedure call is given below.

```
push-name
...             /* procedure name         */
p-call          /* procedure call         */
p-actual-in
...             /* actual input argument  */
p-actual-out
```

```
...             /* actual output argument */
...             /* more arguments         */
...
p-call-off      /* end of procedure call  */
```

Example 1

The I-code for the procedure call without parameters,

Ex1();

is shown below

```
push-name
Ex1
p-call
p-call-off
halt
```

Example 2

The I-code for the procedure call without actual arguments,

Ex2(in, out);

is shwon below.

```
push-name
Ex2
p-call
p-actual-in
^A    /* a null string */
p-actual-out
^A
p-call-off
```

```
halt
```

Example 3

The I-code for the procedure call with actual arguments,

Ex3(out A, in B, in C);

is given below.

```
push-name
Ex3
p-call
p-actual-out
A
p-actual-in
B
p-actual-in
C
p-call-off
halt
```

4.4.2 Algorithm for Procedure Call

When the interpreter reaches the instruction '`p-call`' in the I-code, it calls the function `proc_call` with the procedure name (P_name) as argument. The algorithm for the function `proc_call` is given below.

1. Initialize code offset to 0, initialize an order list for the actual arguments to empty, initialize an order list for the formal parameters to empty, initialize the type to 0, initialize the argument position to 0.

2. If P_name is not in the system relation .rel, report error and return.

3. If P_name is not a procedure, report error and return.

4. Parse the actual arguments.

   (a) Fetch an I-code.

   (b) If the I-code is 'p-actual-in' or 'p-actual-out'

      i. if the I-code is 'p-actual-in', mark the bit of the type corresponding to the argument position;

      ii. fetch the argument and insert it into the order list for the actual arguments; and

      iii. increment the argument position.

   (c) If the I-code is not 'p-call-off', goto step 4a.

5. If the procedure has entries in the system relation .comp, add the formal parameters declared to the order list for the formal parameters;

6. If the number of items in the order list for the actual arguments does not match the number of items in the order list for the formal parameters, report the error and return.

7. If the number of actual arguments is not 0 and the type is not defined in the system relation .comp_type, report error and return. Otherwise, get the code offset of that type.

8. Create a temporary file with name ".P_name.proc.x.y" where x is the value of a static local counter and y is the process-id.

9. Load the I-code from the I-code file of the procedure, and replace formal parameters:

   (a) set the file pointer to the position of the code offset of the I-code file of the procedure;

(b) fetch an I-code from the I-code file of the procedure;

(c) if the I-code is a formal parameter and the corresponding argument is not a null string, write the argument to the temporary file; otherwise, write the I-code to the temporary file; and

(d) if the I-code is not 'halt' goto step 9b.

10. Protect the procedure from being deleted by incrementing the .ntuples of the procedure entry in the system relation .rel[3].

11. Call the interpreter to execute the I-code in the temporary file.

12. Decrement the .ntuples of the procedure entry in the system relation .rel.

13. Delete the temporary file.

14. Return.

4.5 Commands on Procedures

In this section, the implementation of commands on procedures described in Section 3.2 is presented. The implementation of the show relation command will not be presented here because it is not changed.

4.5.1 Print

The I-code

```
push-name
...  /* relation name */
print-rel
```

[3]The method for protection takes advantage of the existing concurrency control mechanism.

is for the "print relation" command. When the interpreter reaches the I-code '`print-rel`', it will call the function `print_rel` using the relation name (R_name) as argument. The algorithm of `print_rel` is given below.

1. If R_name is not a null string, set one_rel to true; otherwise, set one_rel to false.
2. If one_rel is false, ask the user for R_name.
3. If R_name is a null string or 'e!', return.
4. If R_name is a not in the system relation .rel, report error and goto step 8.
5. If R_name is a computation name, call `print_comp(R_name)` and goto step 8.
6. If R_name is a procedure name, call `print_proc(R_name)` and goto step 8.
7. Display the content of the relation.
8. If one_rel is false, goto step 2.
9. Return.

The algorithm of `print_proc` is shown below.

1. Open the source file of the procedure.
2. Display the content of the source file.

4.5.2 Delete

The I-code

```
push-name
...  /* relation name */
del-rel
```

is for the "delete relation" command. When the interpreter reaches the I-code '`del-rel`', it will call the function `del_proc` using the relation name (R_name) as argument if the relation name is a procedure name. The algorithm of `del_proc` is given below.

1. If the procedure is protected, report error and return.
2. Call `del_comp(R_name)` to delete its entries in the system relations .rel, .comp, and .comp_type.
3. Remove the source file, I-code file, and any associated files of the procedure.
4. Return.

4.5.3 Show Procedure

The I-code

```
push-name
...  /* procedure name */
p-show-proc
```

`is for the ''show procedure'' command.`

When the interpreter reaches the I-code '`p-show-proc`', it will call the function `show_proc` using the procedure name (P_name) as argument. The algorithm of `show_proc` is given below.

1. If P_name is not a null string, set only_one to true; otherwise, set only_one to false.
2. If only_one is false, ask the user for P_name.
3. If P_name is a null string or 'e!', return.

4. If P_name is a not a procedure name , report error and goto step 6.

5. Get the type information of the procedure from the system relation .comp_type, and display the type of each block.

6. If only_one is false, goto step 2.

7. Return.

Chapter 5

Conclusion

Procedure facilities have been integrated into Relix, and procedure constructs have been added to Aldat. Two programming language concepts, namely procedural abstraction and parameter passing, have been integrated into an existing database system. Moreover, the notion of symmetric procedures demonstrates that fundamental concepts in databases and programming languages can be unified.

Database systems have been criticized for lack of expressive power in [AB87]. While maintaining the simplicity of the language, procedures allow users to build new complex operations using simple ones. Moreover, procedures can be used as fundamental tools to integrate other programming concepts to Relix. For example, procedures have been used to define event handlers in [EK96].

In this final chapter, the work of this thesis is summarized in Section 5.1. Suggestions for future work are then given in Section 5.2.

5.1 Summary

Procedures have been built to represent procedural abstraction in Relix. While computations abstract only horizontal domain algebra expressions, procedures provide the users with more expressive power by abstracting a sequence of statements such as relation and domain declarations, relational and domain algebra statements, and commands.

Procedure declaration is a statement that specifies the name of the procedure, an optional list of formal parameters, and a procedure body containing one or more blocks of statements. Each block has a distinct type which is inferred by the system using the set of input and output formal parameters of the block. The type of the procedure is then defined to be the union of the types of its blocks. Once a procedure is declared, it is compiled and stored in the database persistently.

The invocation of a procedure uses a stand-alone statement which is written as the procedure name with a list of actual parameters. Each actual parameter is preceded by a keyword **in** or **out** which is used to compute the type of the block to be activated. Formal parameters and actual parameters are paired by position, and actual parameters are passed by name. If the name of the actual parameter is not given, the name of the formal parameter is used as default. Hence, the actual parameter must be an identifier or empty.

The activation of a procedure pushes a temporary i-code file, containing the block of statements to be executed, onto the run-time code stack. The i-code file is popped from the stack when the last statement of the procedure is executed. This implementation allows procedures to call themselves or other procedures.

Users can use the procedure commands to examine and manipulate declared procedures. The source code of the procedure can be reviewed any time using the print command. The system information of the procedure and the type of the procedure can be displayed using the show commands. Finally, the declaration of a procedure can be removed from the database explicitly using the delete command.

5.2 Future Work

In order to increase the expressive power of Relix, the language must support representation and manipulation of complex objects. This brings us to the suggestion of incorporating another programming language concept, namely abstract data type (ADT), into Relix.

In programming language, an ADT is a collection of data structures and operations abstracted into a simple data type. Relix deals with only two types of data objects: relations and domains. Procedures have been built to abstract complicated operations. The objective is to use relations to specify new attribute types. For this we need to convert procedures to domain operations and relations to attributes.

5.2.1 Procedures To Domain Operations

Suppose we have the declarations below.

```
> domain A intg;
> domain B intg;
> relation R(A, B) <- { (1, 2), (2, 5), (3, 6) };
```

Recall that the domain algebra statement

```
> let C be A + B;
```

creates a virtual attribute C which can be actualized with a relation algebra statement such as

```
> S <- [A, B, C] in R;
```

and the actualization of C applies the definition, C = A + B, to every tuple in R. Hence, the value of C in a tuple is the sum of the value of A and the value of B in the same tuple. However, this definition will not work if A and B are relations. This problem can be solved by using the procedure below.

```
> proc Add(A, B, C) is
  {
     let C be A + B;
     C <- [C] in (([A] in A) ijoin ([B] in B));
  };
```

If A equals 1 and equals 2, we can use the following statements to create S.

```
> relation A (A) <- {(1)};
> relation B (B) <- {(2)};
> Add(in A, in B, out C);
> S <+ (A ijoin B ijoin C);
```

Note that A, B, and C are scalar relations. Hence, domain algebra operations can be simulated by procedures. However, in order to simulate the statement

```
> S <- [A, B, C] in R;
```

one must enter the above four statements for each tuple in R.

We can solve the problem by including procedures in domain operations. For instance, the domain algebra statement

```
> let C1 be Add(in A, in B, out C1);
```

defines a virtual attribute C1. The statement

```
> S <- [A, B, C1] in R;
```

actualizes C1 by applying the definition of C1 to every tuple in R as follows.

1. Get the value of the attribute A and create a scalar relation A. For instance, the scalar relation A

   ```
   A(A)
     1
   ```

is created if the value of A is 1.

2. Get the value of the attribute B and create a scalar relation B.

3. Execute the definition: Add(in A, in B, out C1).

4. The value of the attribute C1 is the value in the scalar relation C1. For instance, if the scalar relation C1 is

```
C1(C1)
  3
```

the value of the attribute C1 is 3.

5.2.2 Relations to Attributes

We will use complex numbers as an example to illustrate our idea. Complex numbers can be represented by a relation:

```
> domain IPart real;
> domain RPart real;
> relation COMPLEX(RPart, IPart);
```

where COMPLEX is a relation with two attributes. In order to represent and manipulate COMPLEX as a new attribute type, we need to convert a relation to an attribute, and vise versa.

In the previous section, we have already shown how to convert scalar attributes to scalar relations. The similar idea can be used to convert ADT attributes to ADT relations. First the conversion is defined using a new construct

```
type cmpx is relation COMPLEX(RPart, IPart);
```

which defines a new type called *cmpx*. This statement has the following semantics.

- An attribute of type *cmpx* can be converted to a relation defined on two attributes: RPart and IPart.
- The relation COMPLEX is a hidden state relation which will be redefined by the system as COMPLEX(id, RPart, IPart). The attribute id is a system generated unique identifier which is used to link the relation to the host ADT attribute.

Suppose we have a *cmpx* type of relation with data as shown below.

```
(RPart, IPart)
    1       0
    1       2
```

If we want to convert the relation to an attribute CC whose type is *cmpx*, the system will perform the followings.

1. If the relation already has an id in COMPLEX, then use that id. Otherwise, generate a new unique id for the relation and add the relation to COMPLEX.
2. The value of the attribute CC is the id.

Now suppose the type of an attribute DD is *cmpx*, and the value of DD is 1234. The system can convert DD to a relation using a single Relix statement:

```
DD <- [RPart, IPart] where id=1234 in COMPLEX.
```

5.2.3 Abstract Data Type

We have already shown how to convert relations to attributes and procedures to domain operations. The conversions are invisible to the programmer, who writes ordinary procedures using relational and domain algebras. The conversion is specified by the adt construct, which is a special form of procedure which output one or more types and one or more procedures. For example, the adt declaration for complex number is shown below.

```
>adt COMPLEX_ADT (cmpx, Cwr, C+) is
 {
   type cmpx is relation COMPLEX(RPart, IPart);

   proc Cwr(R, I, Z) is
   {
      let RPart be R;
      let IPart be I;
      Z <-([RPart] in R) ijoin ([IPart] in I);
   }
   alt
   {
      let R be RPart;
      let I be IPart;
      R <- [R] in Z;
      I <- [I] in Z;
   };

   proc C+(X, Y, Z) is
   {
     let XR be RPart;
     let XI be IPart;
     let YR be RPart;
     let YI be IPart;
     let IPart be XI + YI;
     let RPart be XR + YR;
     Z <- [RPart, IPart] in (([XR, XI] in X)
                       ijoin ([YR, YI] in Y));
```

```
        };
    };
```

To use the adt, we issue a procedure call like statement

```
> COMPLEX_ADT( out cmpx, out Cwr, out C+);
```

which performs the followings:

1. create the state relation and register the conversion for type *cmpx*;
2. declare a procedure Cwr;
3. declare a procedure C+.

Moreover, the above statement implies that changing types and procedures names within an ADT is possible.

Appendix A

Modification of the Parser Module

In this appendix, we list all the YACC rules and translator parameters to be added to the existing parser module of Relix.

A.1 Parser Rules for Procedure

In this section, we list all the rules to be added to the existing parser module of Relix in alphabetical order. Any rule that is already in Relix will not be listed here.

```
procedure_actual_parameter:
  IN identifier  {translator( P_ACTUAL_IN);}
       |
  OUT identifier {translator( P_ACTUAL_OUT);}
       |
  IN
  {strcpy( yytext, FILLER_s); translator( IDENTIFIER);
   translator( P_ACTUAL_IN);}
       |
  OUT
  {strcpy( yytext, FILLER_s); translator( IDENTIFIER);
   translator( P_ACTUAL_OUT);}
       ;
procedure_actual_parameter_list:
  procedure_actual_parameter
       |
  procedure_actual_parameter_list ',' procedure_actual_parameter
       ;
procedure_actual_parameter_option:
  /* empty */
```

```
      |
  procedure_actual_parameter_list
      ;
procedure_block:
  {translator( P_BLOCKBGN); translator( PUSH_NAME_L);}
  '{' procedure_statements '}'
  {translator( P_BLOCKEND);}
      ;
procedure_body:
  procedure_block
      |  procedure_body 'alt' procedure_block
      ;
procedure_call:
  identifier {translator( PUSH_NAME_R); translator( P_CALL);}
  '(' procedure_actual_parameter_option ')' {translator( P_CALL_OFF);}
      ;
procedure_command:
  DEL_DOM {translator( PUSH_NAME_R); translator( DEL_DOM);}
      |
  DEL_REL {translator( PUSH_NAME_R); translator( DEL_REL);}
      |
  PRINT_REL {translator( PUSH_NAME_R); translator( PRINT_REL);}
      |
  SHOW_DOM  {translator( PUSH_NAME_R); translator( SHOW_DOM);}
      |
  SHOW_REL  {translator( PUSH_NAME_R); translator( SHOW_REL);}
      |
  SHOW_RD   {translator( PUSH_NAME_R); translator( SHOW_RD);}
      ;
procedure_declaration:
  'proc' procedure_name {translator( P_PARAM_OPTION);}
  '(' procedure_parameter_list ')' {translator( P_DEC);}
  'is' procedure_body {translator( P_DEC_OFF);}
      ;
procedure_definition_statement:
  LET  identifier BE {translator( LET); translator(PUSH_NAME_R);}
  domain_expression {translator( BE);}
      |
  LET  identifier INITIAL {translator( LET);  translator(PUSH_NAME_R);}
  domain_expression BE {translator( ELSE);}
  domain_expression {translator( LET_WITH_INITIAL);}
```

```
        ;
procedure_domain_declaration:
  DOMAIN_DEC  identifier {translator( DOMAIN_DEC);
  translator(PUSH_NAME_R);}
  TYPE {translator( IDENTIFIER); translator( TYPE);}
        ;
procedure_else_part:
/* empty */
       |
ELSE {translator( PUSH_NAME_L); translator( IF_ELSE); }
procedure_statement_part
       ;
procedure_executable_statement:
  identifier {translator( EXECUTION);}
        |
  identifier '{'{translator( LEFT_PARTIAL_FIT);}
  constant_list '}' {translator( LEFT_PARTIAL_OFF);}
  ASSIGN {translator(PUSH_NAME_R);}
  relational_expression{translator( ASSIGN);}
        |
  name_valued_expression ASSIGN
  {translator( ELSE); translator(PUSH_NAME_R);}
  relational_expression {translator( ASSIGN_SCALAR);}
        |
  name_valued_expression INCREMENT
  {translator( ELSE); translator(PUSH_NAME_R);}
  relational_expression {translator( INCREMENT_SCALAR);}
        |
  renaming_increment_left ASSIGN
  {translator( ELSE); translator( RENAME_OPTION);
   translator(PUSH_NAME_R);}
  domain_option ']' relational_expression {translator( RENAME);}
        |
  renaming_increment_left INCREMENT
  {translator( ELSE); translator( RENAME_OPTION);
    translator(PUSH_NAME_R);}
  domain_option ']' relational_expression
  {translator( RENAME_INCREMENT);}
        |
  name_valued_expression nest_operator
  {push_short_stack( join_flag, &join_stack);
```

```
   translator( NULL_DOMAIN_OPTION); translator(PUSH_NAME_R);}
  identifier {translator( pop_short_stack( &join_stack));}
        |
  renaming_increment_left domain_option nest_operator
  {push_short_stack( join_flag, &join_stack);
   translator( DOMAIN_OPTION_1); translator(PUSH_NAME_R);}
  domain_option ']' identifier
  {translator( pop_short_stack( &join_stack));}
        ;
procedure_if:
  IF { translator( PUSH_NAME_R);}
  no_if_relational_expression {translator( IF_THEN); }
  THEN {translator(PUSH_NAME_L);} procedure_statement_part
  procedure_else_part
  {translator( PUSH_NAME_L); translator( IF_STMT_END); }
        ;
procedure_name:
  identifier
        ;
procedure_one_statement:
  procedure_command
  {translator( P_STMTS_DELIM); translator( PUSH_NAME_L);}
        |
  procedure_statement ';'
  {translator( P_STMTS_DELIM); translator( PUSH_NAME_L);}
        ;
procedure_parameter_list:
  /* empty */
        |
  procedure_parameters
        ;
procedure_parameters:
  identifier {translator( P_PARAM_LIST);}
        |
  procedure_parameters ',' identifier {translator( P_PARAM_LIST);}
        ;
procedure_relation_declaration:
  RELATION_DEC identifier
 {translator( DOMAIN_OPTION); translator(PUSH_NAME_R);}
 '(' attribute_list ')' rcp_ln_option {translator( RELATION_DEC);}
        ;
```

```
procedure_sequence_of_statements:
procedure_statement ';'
{translator( PUSH_NAME_L); translator( STMTS_DELIM ); }
procedure_body_of_statements
      |
procedure_statement
{translator( PUSH_NAME_L); translator( STMTS_DELIM ); }
      ;
procedure_statement:
  procedure_domain_declaration
       |
  procedure_relation_declaration
       |
  procedure_definition_statement
       |
  procedure_executable_statement
       |
  procedure_update_statement
       |
  procedure_call
       |
  procedure_if
       ;
procedure_statements:
  procedure_one_statement
       |
  procedure_statements procedure_one_statement
       ;
procedure_statement_part:
procedure_statement
{translator( PUSH_NAME_L); translator( STMTS_DELIM ); }
      |
'{' procedure_sequence_of_statements '}'
      ;
procedure_update_first:
  UPDATE {translator( PUSH_NAME_R);}  identifier
  {translator( UPDATE_RELATION);}
       ;
procedure_update_statement:
  procedure_update_first update_last_1
       |
```

```
procedure_update_first '[' {translator( DOMAIN_OPTION);}
domain_option update_last_2
      ;
```

A.2 Translator Parameters for Procedure

In the section, we list all the translator parameters for procedure, together with the desired I-code, to be added to the translator function.

Parameter	I-code
P_ACTUAL_IN	p-actual-in x
P_ACTUAL_OUT	p-actual-out x
P_BLOCKBGN	p-blockbgn
P_BLOCKEND	p-blockend
P_CALL	push-name x p-call
P_CALL_OFF	p-call-off
P_DEC	push-name ... push-count n push-name x p-dec
P_DEC_OFF	p-dec-off
P_PARAM_OPTION	
P_PARAM_LIST	
P_STMTS_DELIM	p-stmts-delim
PUSH_NAME_L	push-name-left x
PUSH_NAME_R	push-name-right x
SHOW_PROC	push-name x p-show-proc

where x is a string obtained by popping an item from the scalar_stack, and ' ... ' is a list of identifiers obtained by popping the list from the list_stack.

Appendix B

List of Error Messages

The error messages described in Section 3.5 are summarized alphabetically below.

- "..." has too many parameters
  The procedure declaration has more than 30 formal parameters.
- Attributes of "..." and "..." are not same
  A relational increment or update add statement attempts to add two relations defined on different attributes.
- Block "..." and block "..." have the same type
  The procedure declares two blocks with the same type.
- Cannot execute "..."
  The procedure name given in the procedure call statement is not a procedure, or a statement attempts to use a procedure name in a relational expression.
- Cannot find "..."
  A statement attempts to use an undeclared object as input.
- Domain "..." already appeared in this list
  Duplicated identifier is found in the formal parameter list.
- Domain "..." still used
  A delete domain command (**dd!!domain_name**) attempts to delete an existing attribute which is used by a relation or virtual attribute.
- Domain "..." still used; cannot change type
  A domain declaration or domain algebra statement attempts to re-declare an existing attribute with another data type.

- If_cond is not a boolean singleton scalar
  The if_cond expression in the if statement does not produce a boolean singleton scalar.

- Invalid token: "..."
  An invalid token is encountered when the system is parsing the user input.

- Name "..." has been used
  The procedure name has been used by other procedures, relations, or computations.

- No common attributes
  A relational algebra binary operation, except natural join and union join, a ttempts to operate on two relations with no common attributes, but the set of join attributes is not stated explicitly in the expression.

- No such type in "..."
  The procedure call statement attempts to call an undefined block.

- Not enough arguments in "..."
  The procedure call statement has less actual arguments than the number of formal parameters declared.

- Procedure "..." has more than one block
  The procedure has no formal parameters and attempts to declare more than one block.

- Procedure "..." is protected. Deletion failed
  A delete relation command (**dr!!<procedure_name>**) attempts to delete a caller procedure.

- Too many arguments in "..."
  The procedure call statement has more actual arguments than the number of formal parameters declared.

- Type mismatch
  A domain expression attempts to operate on two attributes of incompatible data type.

Bibliography

[A+76] M. M. Astrahan et al. System R: Relational approach to database management. *ACM Transactions on Database Systems*, 1(2):97–137, June 1976.

[AB87] M. P. Atkinson and O. P. Buneman. Types and persistence in database programming languages. *ACM Surveys*, 19(2):106–190, June 1987.

[ABC+83] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshott, and R. Morrison. An approach to persistent programming. *Computer Journal*, 26(4):408–419, Nov. 1983.

[ACC81] M. P. Atkinson, K. J. Chisholm, and W. P. Cockshott. Ps-algol: An algol with a persistent heap. *SIGPLAN Notices*, 17(7):24–31, July 1981.

[ACO85] A. Albano, L. Cardelli, and R. Orsini. Galileo: A strongly typed interactive conceptual language. *ACM Transactions on Database Systems*, 10(2), June 1985.

[AM88] M. P. Atkinson and R. Morrison. Types, bindings and parameters in a persistent environment. In *Data types and Persistence*, pages 3–20. Springer-Verlag, New York, 1988.

[Ame66] Ameriican National Standards Institute, New York. *American National Standard Programming Language FORTRAN*, 1966.

[AS85] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, Massachusetts, 1985.

[Atk78] M. P. Atkinson. Programming languages and databases. In S. B. Yao, editor, *The 4th International Conference on Very Large Data Bases*, pages 408–419, Berlin, West Germany, Sept. 1978.

[Atk85] M. P. Atkinson. Procedures as persistent data objects. *ACM Transactions o Programming Languages and Systems*, 7(4):539–559, Oct. 1985.

[Atk89] M. P. Atkinson. Questioning persistent types. In *Proceedings of the Second International Workshop on Database Programming Languages*, pages 2–24, Gleneden Beach, Oregon, June 1989.

[CM82] A. J. Cole and R. Morrison. *An introduction to Programming with S-algol*. Cambridge University Press, New York, 1982.

[COD68] CODASYL COBOL Committee. *COBAL Journal of Development*, 1968.

[Deu90] O. Deux. The story of O2 system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, Mar. 1990.

[DJ95] H. L. Dershem and M. J. Jipping. *Programming Languages: Structures and Models*. PWS Publishing Company, Boston, MA, 1995.

[EK96] A. Z. El-Kays. Implementing Event Handlers in a Database Programming Language. Master's thesis, McGill University, Montreal, Canada, 1996.

[GM88] G. Graefe and D. Maier. Query optimization in object-oriented database systems. In K.R. Dittrich, editor, *International Workshop on Object-Oriented Database Systems*, pages 358–363, Bad Munster, September 1988.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Mass., 1983.

[Joh75] S. C. Johnson. Yacc: Yet another compiler-compiler. Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[KR78] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, N.J., 1978.

[L+77] B. W. Lampson et al. Report on the programming language Euclid. *SIGPLAN Notices*, 12(2):1–79, Feb. 1977.

[Lal86] N. Laliberté. Design and Implementation of a Primary Memory Version of Aldat. Master's thesis, McGill University, Montreal, Canada, 1986.

[Les75] M. E. Lesk. Lex: a lexical analyzer generator. Technical Report 39, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10), Oct. 1991.

[LLPS91] G. M. Lohman, B. Lindsay, Hamid Pirahesh, and K. B. Schiefer. Extensions to Starburst: Objects, types, functions, and rules. *Communications of the ACM*, 34(10):94–109, Oct. 1991.

[Mer77] T. H. Merrett. Relations as programming language elements. *Information Processing Letters*, 6(1):29–33, Feb. 1977.

[Mer84] T. H. Merrett. *Relational Information Systems*. Reston Publishing Company, Reston, Virginia, 1984.

[Mer93] T. H. Merrett. Computations: Constraint programming with the relational algebra. In *International Symposium on Next Generation Database Systems and their Applications*, pages 12–17, Fukuoka, Japan, September 1993.

[Mil84] R. Milner. A proposal for standard ML. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Progrmming*, pages 184–197, Austin, Texas, August 1984.

[ML87] M. Marcotty and H. Ledgard. *The World of Programming Languages*. Springer-Verlag, New York, 1987.

[ML89] T. H. Merrett and N. Laliberté. Including scalars in a programming language based on the relational algebra. *IEEE Transactions on Software Engineering*, 15(11):1437–1443, Nov. 1989.

[Mnu92] E. Mnushkin. Inheritance in a Relational Object-oriented Database System. Master's thesis, McGill University, Montreal, Canada, 1992.

[MS89] F. Matthes and J. W. Schmidt. The type system of DBPL. In *Proceedings of the Second International Workshop on Database Programming Languages*, pages 219–225, Gleneden Beach, Oregon, June 1989.

[MS91] T. H. Merrett and H. Shang. Unifying programming languages and databases: Scoping, metadata, and process communication. In *The Third International Workshop on Database Programming Languages: Bulk Types and Persistent Data*, pages 139–148, Nafplion, Greece, Aug. 1991.

[MSOP86] D. Maier, J. Stein, A. Otis, and A. Purdy. Development of an Object-oriented DBMS. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 472–482, Portland, OR, Sept. 1986.

[RS79] L. A. Rowe and K. A. Shoens. Data abstraction views and updates in RIGEL. In P.A. Berstein, editor, *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 71–81, Boston, Mass., May-June 1979.

[SAH87] M. R. Stonebraker, J. Anton, and E. Hanson. Extending a database system with procedures. *ACM Transactions on Database Systems*, 6(3), Sept. 1987.

[SAHR84] M. R. Stonebraker, E. Anderson, E. Hanson, and B. Rubenstein. Quel as a data types. In *Proceedings of the ACM-SIGMOD International conference on Management of Data*, Boston, Mass., June 1984.

[Sch77] J. W. Schmidt. Some high level language constructs for data of type relation. *ACM Transactions on Database Systems*, 2(3), Sept. 1977.

[Sho79] J. E. Shopiro. Theseus—a programming language for relational databases. *ACM Transactions on Database Systems*, 4(4), Dec. 1979.

[SR86] M. Stonebraker and L. A. Rowe. The design of POSTGRES. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, Washington, D.C., May 1986.

[Sto86] M. Stonbraker. Inclusion of new types in relational data base systems. In *Second International Conference on Data Base Engineering*, Los Angeles, Ca., Feb. 1986.

[Str86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Mass., 1986.

[Sut94] N. Sutyanyong. Procedural Abstraction in a Relational Database Programming Language. Master's thesis, McGill University, Montreal, Canada, 1994.

[SWKH76] M. R. Stonebraker, E. Wong, P. Kreps, and G. D. Held. The design and implementation of INGRES. *ACM Transactions on Database Systems*, 1(3), Sept. 1976.

[Ten81] R. D. Tennent. *Principles of Programming Languages*. Prentice-Hall, New York, 1981.

[U.S80] U.S. Department of Defense, Washington. *Reference Manual for the Ada Programming Language*, 1980.

[W+75] A. Van Wijngaarden et al. Revised report on the algorithmic language Algol 68. *Acta Informationca*, 5:1–236, 1975.

[Was79] A. I. Wasserman. The data management facilities of PLAIN. In P.A. Berstein, editor, *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 60–70, Boston, Mass., May-June 1979.

[Wir71] N. Wirth. The programming language PASCAL. *Acta Informatica*, 1(1), May 1971.

[Wir77] N. Wirth. Modula: A language for modular multiprogramming. *Software—Practice and Experience*, 7, 1977.

[Wir83] N. Wirth. *Programming in Modula-2.* Springer-Verlag, 1983.

[WWG51] M. V. Wilkes, D. J. Wheeler, and S. Gill. *The Preparation of Programs for a Digital Computer.* Addison-Wesley, New York, 1951.