Text Operators in a Relational Programming Language

Jiantao Xie

School of Computer Science
McGill University, Montréal

January 2005

A thesis submitted to McGill University
in partial fulfilment of the requirements of the degree of
Master of Science in Computer Science.

Contents

Abstract

JRelix is an implementation of a relational database system which provides a significantly powerful database programming language and which is especially adept with complex data. This thesis documents an enhancement of JRelix which provides intuitive descriptions and efficient manipulations for textual information in the database.

Coupled with the relational and domain algebra in JRelix, the new database system supports rapid textual information retrieval, flexible text mining, structured text schema discovery, relational operations on text, transformation between text and relation, and powerful pattern matching in structured or unstructured data. These endow JRelix with the capacity of handling complicated textual information from heterogeneous data sources (e.g., data from the web), and also enrich its searching power on vast bodies of electronic data as a text/relational database management system.

Résumé

JRelix est une implémentation d'un système de base de données relationnelle. JRelix fournit un langage de programmation de base de données expressif et est particulièrement adepte à traiter des données complexes. Cette thèse propose des améliorations à JRelix qui permet des descriptions intuitives et la manipulation efficace de données textuelles. JRelix est capable de traiter des types de données texte, incluant le texte d'origine et des données de type langage SGML, d'une façon efficace et significative.

Grâce à l'intégration avec l'algèbre relationnelle et l'algèbre de domaine de JRelix, le nouveau système de base de donnée supporte le recouvrement rapide de données textuelles, l'exploration flexible de texte, la découverte de schémas de textes structurés, les opérations relationnelles sur le texte, la transformation entre formats textuels et relationnels et le filtrage expressif de données structurées ou non-structurées. Ces fonctions permettent JRelix de supporter des données textuelles complexes venant de sources hétérogènes (p. ex. du Web), et aussi d'enrichir le pouvoir de recherche sur une vaste quantité de données électroniques en tant que système de gestion de base de données relationnelle/textuelle.

Acknowledgements

First and foremost, I would like to express my gratitude to my thesis supervisor, Professor Tim H. Merrett, for his attentive guidance, enthusiastic encouragement and endless patience throughout the research for and the preparation of my thesis. Without his invaluable ideas, piercing insight, copious experience and generous financial support, the way through my graduate studies and life in a foreign country would not have been so successful and fruitful.

I am grateful to my colleagues in the Aldat laboratory, with a special mention of our JRelix system's coordinator Zhongyan Wang who gave me her unstinting assistance on the usage of facilities in the laboratory and her noteworthy instructions on the enhancement of the system. I would also like to thank each member of the staff of the School of Computer Science for their administration and technical support.

I extend my special thanks to my friend John C. Owen for his willingness and devotion in prove-reading each page of the thesis and in addition for his generous help in my daily life during the course of my studies. Furthermore, I wish to thank François Pepin for the translation into French of the thesis abstract and for his constructive suggestions.

Lastly but by no means least I must acknowledge my most sincere appreciation to my parents. They supported my decision to leave my hometown and pursue a Master's degree overseas, they gave me unremitting encouragement and priceless advice in my studies, and without question, they unhesitatingly sponsored me during the complete time of my University endeavours. It would have been impossible for me to have achieved so much without their love and endorsement.

Chapter 1

Introduction

Text, as a special form of electronic information, nowadays dominates various kinds of data sources, e.g. the data on the web and in the file systems. Based on the work undertaken on text research, we are now proposing to integrate text manipulation into JRelix, which is a relational database system adopting Aldat as its database programming language. In section 1.1, we will address the research background and the previous achievements which have been attained in this area. In section 1.2, we will further present the motivation for the introduction of text manipulation into JRelix. In the last section, we will outline the content of the thesis.

1.1 Background and Related Work

The application of conventional database technology is becoming increasingly important in academic research and business enterprises. While the existing design and development of database systems are sound for traditional data management, the particular techniques used for manipulating vast bodies of information embedded in text are repeatedly found to be necessary as a complement to the conventional database systems. In the last few years, a considerable amount of research and implementations

have been conducted in pattern match, text structure, text database modeling and text schema discovery, etc.

1.1.1 Forms of Text

The forms of text vary substantially among databases and file systems. It is essential to understand the diverse nature of text before we start to design our text algebra and model our text database systems.

- **Plain Text**

Plain text is the simplest form of text. It can be viewed as a sequence of characters, words or other bounded strings (i.e. lines) [Tomp97]. Each character in the text can be identified by its position within the character sequence, and each substring can be identified by its starting and ending positions. Text matching, thus requires finding substrings that satisfy the given criteria for exact match or approximate match, and returns the positions or the values of the matches.

As a sequence of words, text can be parsed into tokens. Substrings can now be identified by the positions of their first and last words, where position means the number of words from the beginning of the text , called word sequence or word offset. The problem with this view is to define the boundaries of the words, which can be white space, punctuation marks, line-feeds or no boundary at all, as in some Asian languages. An interesting model of text parsing is presented in the PAT system [Salm94]. Text is considered to be a collection of overlapping indexed elements:

```
Text is a sequence of words
is a sequence of words
a sequence of words
sequence of words
of words
words
```

The starting positions can be at any character position, and in this example they are

chosen to be at the beginning of the words. Text storage and search problems in this model can also be solved with tries [Shan95].

Many applications require additional structure to be imposed on text, i.e. lines. Under a line-based model, the text operations, e.g. line matching and line retrieval, are constrained to within one line. Arbitrary imposing of line boundaries on text makes it inefficient to retrieve all desired information in multiple lines, e.g. searching with the Unix file operators *grep* and *diff*.

- **Marked Up Text**

Text is usually a rich combination of content and form [Tomp97]. The forms of text are typically recorded using some sort of markup [Coom87], which is represented by special strings of characters, e.g. tags, embedded in the text. A tag is usually distinguished from the content of the text by beginning with a reserved character that may be disallowed in the pure text, e.g. angle brackets in SGML [ISO86] and backslashes in LaTeX [Lamp94].

Marking up text with *x*ML tags, where *x* stands for different kinds of markup standards, provides more elaborate attributes for the text [Merr03]. As we will discuss in the next section, markup tags can also be used to encode hierarchical and other complex structured text.

- **Structured Text**

Structured text is any text that has an identifiable internal structure [Brown98]. This structure may be explicitly established by the inclusion of appropriate electronic markups [Coom87, Tomp89], possibly complemented by an external document type definition (DTD) or it may be implied by the language contained within the text. The following example shows an XML file and its associated grammar.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE note [
    <!ELEMENT note (to, from, heading, body)>
    <!ELEMENT to (#PCDATA)>
    <!ELEMENT from (#PCDATA)>
    <!ELEMENT heading (#PCDATA)>
    <!ELEMENT body (#PCDATA)>
]>
<note>
    <to>Tove</to>
    <from>Jani</from>
    <heading>Reminder</heading>
    <body>Don't forget me this weekend!</body>
</note>
```

Figure 1.1 An XML file and its associated grammar

There are mainly two structured model types [Tomp97]. The first characterizes text as a collection of documents, each of which is a simple character string. The search for the matching strings results in the retrieval of a set of documents containing those strings rather than making direct references to the strings themselves. The second type of model divides documents into named fields, for the purpose of formatting and of restricting searches. For example, a bibliographic record is subdivided into text fields named "author", "title" and "biblio" etc. The new OED project in the University of Waterloo takes the second model, which identifies and delimits its dictionary components by introducing tags into a text stream [Tomp92].

The structured text schema, normally expressed as a grammar, is a strong parallel to the schemas for conventional database records, which are far less useful in managing text than in conventional databases [Gonn87]. Grammars are constraints on sets of text that help to record the structure, determine the validity of new data, and guide query optimizers. When the text structure is defined by a context-free grammar, text can be modeled and manipulated as trees [Salm96]. This provides the basis for an algebra in which common needs are conveniently expressed.

- **Semi-structured Text**

In comparison with structured text, some data sources are designed with non-rigid structures for convenience, e.g. the ACeDB genome database [Thie92], or their schemas are normally contained within the data, which is usually self-describing, e.g. the HTML data on the web. They are named semi-structured data. There is no separate schema in some forms of semi-structured data, and in others it exists but only applies loose constraints on the data. Semi-structured data has recently emerged as an important topic because of the growing data sources on the Web, its flexible format for data exchange between disparate database and its help in the processing of structured data [Bune97]. Different data models have been proposed to reconcile the semi-structured data sources and rigid structured data sources, e.g. database systems [Abit99].

The general representation of the semi-structured data is kind of a rooted, directed and labeled graph, such as in the model of the Lore project and the Lorel query language [McHu97, Abit97b]. Although cycles should be allowed in the data, strategically we also refer to these graphs as trees with links or mutual definitions of elements [Merr03].

1.1.2 Text/relational Databases

In recent years, text as one form of electronic information is becoming more and more important for academic research and business enterprises. However, it is stored, accessed and manipulated in ad hoc fashions, and text management is not federated into the conventional database approach in many database systems. Typical text management systems which specify text access languages are incapable of simultaneously processing conventional data [Wein85]. Alternatively, long data fields are introduced into conventional databases to provide text storage. However, text manipulations and structured text are not usually supported by these systems.

If text is to be embedded within conventional databases, the design of text/relational

database has to be able to define views over texts that are accessible from those systems, without destroying the texts themselves and without undesired duplication of data [Brown98]. What a federated database system initially needs is a simple text framework that is compatible with the rest of the database system's model and encapsulates most of the significant properties of structured/semi-structured text. The database query language must then be coupled with operations on text that performs effective query, retrieval and update of selected text fragments [Raym96].

To combine text manipulation into relational database management, the work in the Centre for the New OED and Text Research at the University of Waterloo has come up with a text/relational database management system (T/RDBMS) model and an extended SQL that provides access to the structured text described by SGML [Blak94]. As potential clients may depend on the conventional relational database, the proposed T/RDBMS model is essentially an integration of commercial relation database systems and commercial full-text search engines. The SQL extension is capable of handling SGML-based data in a simple way, extracting from highly structured text into relations and preserving the integrity of complex text units, but it is not designed to address the needs of semi-structured data and is unable to perform text update. The follow-up research introduced a structured text ADT for object-relational database, which allows text to be queried, retrieved, marked up, associated with a schema and integrated with relational information [Brown98]. The proposed text extensions yield easy definitions and dynamic constructions of relational views of structured text derived from hierarchically structured text, marked subtexts, and extracted subtexts.

1.1.3 Text Schemas

For a text document in which structural elements are identified with markups, it is always necessary to attain its schema which contains element nesting and ordering structures. Nowadays, there are two types of text formats dominant in the text database systems. One is the highly structured data, typical of SGML data which is restricted by a

Document Type Declaration (DTD); another is the semi-structured data, typical of Web sources, e.g. XML and HTML data. A considerable amount of research has been carried out in the applications of grammars on texts, or the discovery of schemas within texts.

The highly structured data is normally associated with a grammar to describe its content. Queries involving the data may use this grammar, which must therefore be made known to the database. A good example of highly structured data is the New Oxford English Dictionary, which is converted from its original printed form to a computerized form with explicitly tag-structured elements [Kazm86]. Text data constrained by a given grammar can be considered as a parse tree. It results in the convenience of extending traditional query languages, such as SQL, to include operations on structured text. But on the other hand, it limits the variability of the structure of text and fails to handle the irregularity in text databases. Even the approach of allowing absence of subtexts or introducing optional pattern matching is not satisfactory [Brow98].

Rather than having a fixed grammar that describes all possible forms of the data, the semi-structured data appears more flexible and thus dominates the data sources on the Web. To combine the semi-structured data with the traditional data in database systems, the necessary arises to discover its variable schema. The Lore system has been designed to generate the “DataGuide” from the semi-structured data, which plays in a traditional schema role and helps optimize and formulate queries [McHu97]. Another approach of schema discovery is also proposed based on stochastic grammatical inference [Matt00]. It introduces dynamic grammar into semi-structured data management, and is able to scale on large data sets and handle complicated text structure. Adopting the schema discovery tools as an important component in data management, more and more researches are directed towards semi-structured data extraction and integration from heterogeneous data sources by defining the architecture of a common metadata repository and a comprehensive framework [Shet90, Litw90, Gao99, Kori99, Calv01].

1.1.4 Text Mining

Another significant area of research on text is on knowledge discovery, a non-trivial process of identifying valid, novel, potentially useful and ultimately understandable patterns in data [Fayy96]. Differing from generic data mining in structured databases, text mining operates on the textual information normally in unstructured forms, and it discovers and uses the implicit structure (e.g. grammatical structure) of the texts. It is subdivided into research on linguistic processes, document classification, and automated extraction from textual data sources.

Natural language analysis is one of the most fruitful fields of research on text mining. The early work focused at the word or tag level, e.g. to infer association rules between individual words [Lin01, Silv98, Coop97]. The more recent work proposed to perform text mining at the term (or phrase) level, e.g. to extract and analyze terms from a text collection to construct a term taxonomy [Feld98], or to associate the extracted phrase with the time factor to identify trends in large text databases [Lent97].

Textual document classification and categorization is one practical research area of text mining due to the recent increase of documents in the digital form. Machine learning techniques are predominantly applied in text classification systems, e.g. a general inductive process automatically builds a classifier by learning, from a set of previously classified documents, the characteristics of one or more categories [Seba02]. The fruit of the text classification research is used in many applicative contexts, ranging from the e-mail automated categorization [Agra99], to the World-Wide-Web navigation [Chaf00, Midd01] and in general any application requiring document selection or organization.

There are several specialized automated extraction tools implemented in comprehensive database systems. They are mostly designed for retrieval of biomedical data from electronic literature sources, e.g. the MuteXt system [Horn03] which is interested in point mutations, and the KEX [Fuku98] and the ABGENE [Tana02] systems which

identify protein names. Generally, the techniques of supervised learning, pattern matching and part-of-speech tag marking are used in the extraction processes.

1.1.5 Aldat and JRelix

The relational model of database was first proposed by Dr. E. F. Codd in the classical paper "A Relational Model of Data for Large Shared Data Banks" in 1970. In Codd's relational model, a collection of tables called relations is used for data storage. Nowadays, his model is widely accepted as a standard in database systems. However, the industrial implementations of relational database usually lack of expressive computation power and fail to meet the needs of complex data manipulation. This leads to research in the field of database programming languages (DBPL). DBMS are capable of handling large amounts of persistent data, while DBPL provide well-proven and powerful techniques for generating, manipulating and optimizing data on the databases.

Aldat, as a database programming language based on extended algebra, is designed to address the needs of massive operations on relations with the secondary storage approach. In Aldat, each relation is viewed as a table consisting of rows and columns, which are termed "tuples" and "attributes" respectively. The term "domain" refers to the set of legal values that attributes can contain, i.e. the data type of an attribute.

There are two kinds of algebra essential in Aldat, namely relational algebra and domain algebra. Relational algebra, which extends that proposed by Codd, consists of a set of operations applied on relations for information retrieval. There is no operation performed on individual tuples. Every relational operator takes relations as operand and produces a relation as the result, which gives programmers a high level of programming thought. Domain algebra, introduced as operations on the attributes, provides treatment of attributes independent of relations [Merr84]. That is to say, it allows programmers to create new domains called virtual domains, from existing attributes without regard to where the attributes reside. A virtual domain is only actualized and given a value when an

operand relation is specified through the relational algebra.

JRelix, as the incarnation of Aldat, is a relational database system developed at the Aldat lab in the School of Computer Science at McGill University. In addition to relational and domain operations, it provides computations for relational programming [Lui96, Bake98], nested relations for complex data construct [He97, Hao98], relational states for object-oriented computing [Zhen02] and event handlers for active database programming [Sun00], etc. The functionalities of relational OLAP, data mining and multi-database are also implemented in JRelix.

1.2 Motivation

The federated text/relational databases address the dependence of business enterprises on the traditional database systems and the needs of text, especially structured text, storage and manipulation. However, the highly structured text model adopted by these systems also causes limitations.

The imposition of a schema on the text makes it incapable of catering for the irregularity in the structure of structured text databases [Abit97a]. Although in some developed models the absence of subtexts could be allowed, accepting the optional components of the text in the model produces two phases of extraction and results in unacceptable efficiency [Brown98]. It is also difficult to decide in advance on a single appropriate schema, because the data structure and element types may evolve rapidly. Theses characteristics result in frequent schema modifications, a common headache for database administration.

Another limitation of the model is its inability to handle the enormous amount of semi-structured data on the World Wide Web which is varied and irregular. Although considerable effort has been expended on reconciling data from multiple, heterogeneous data sources into well-structured data conforming to a single uniform schema [Comp91,

Gao99, Kori99, Calv01], the problems caused by the change of source structure or the addition of new sources can not be addressed satisfactorily [McHu97].

In most existing text/database models, the capability of text operations focuses on marked-up text management. However, plain text is more subtle as a sequence of characters without special strings indicating its structure. If we capture the pattern of its natural elements such as words, sentences and paragraphs, we might be able to analyze its structure, and discover the hidden knowledge inside the text. As an example, the following text gives a description of three computer science courses:

```
COMP 575 - Fundamentals of Distributed Algorithms
Study of a collection of algorithms that are basic to the world of concurrent
programming...
Prerequisite: COMP 310
Instructor: Carl Tropper

COMP 617 - Information Systems
Seminar course. A major area of application of the techniques covered in 308-612
is discussed...
Prerequisite: COMP 612
Instructor: Timothy Merrett

COMP 642 - Numerical Estimation
Efficient and reliable numerical algorithms in estimation and their
applications...
Prerequisites: MATH 323, MATH 324 and COMP 350
Instructor: Xiao-Wen Chang
```

Figure 1.2 A text of course description

This is typical unstructured data on the Web. As we can see from the text, the information of each course is limited within one paragraph. Inside each paragraph, the course *title*, *description*, *prerequisite* and *instructor* appear on separate lines and in a constant sequence. Furthermore the *title*, *prerequisite* and *instructor* start with the key words *COMP*, *Prerequisite* and *Instructor* respectively, which help to distinguish them from the *description*. With a plain text mining tool, we would be able to extract the desired

information from the text, and convert it into a relation, as follows:

Courses			
(Title	Description	Prerequisite	Instructor)
COMP 575 - Fund...	Study of a col...	COMP 310	Carl Tropper
COMP 617 - Info...	Seminar course...	COMP 612	Timothy Merrett
COMP 642 - Nume...	Efficient and ...	MATH 323, MATH 324 and ...	Xiao-Wen Chang

Figure 1.3 Courses relation

To query the semi-structured/unstructured data in a relational/text database, traditional query languages are inappropriate, because the structure of the data is unpredictable, and in some cases there is no explicit structure at all. Even if a structure can be imposed on the unstructured data, the structure is irregular, and may evolve rapidly, e.g. the data of *prerequisite* in our example may be absent for some elementary courses. Aldat is an expressive database language for querying such data effectively. It does not require the data schema to be fully known to the system before the performance of data extraction, and it provides powerful path expressions which permit a flexible form of navigational access and are particularly suitable when the details of the structure are not clear or the desired data is dispersed on a complex structure. Figure 1.4 gives a family tree as an example of semi-structured data.

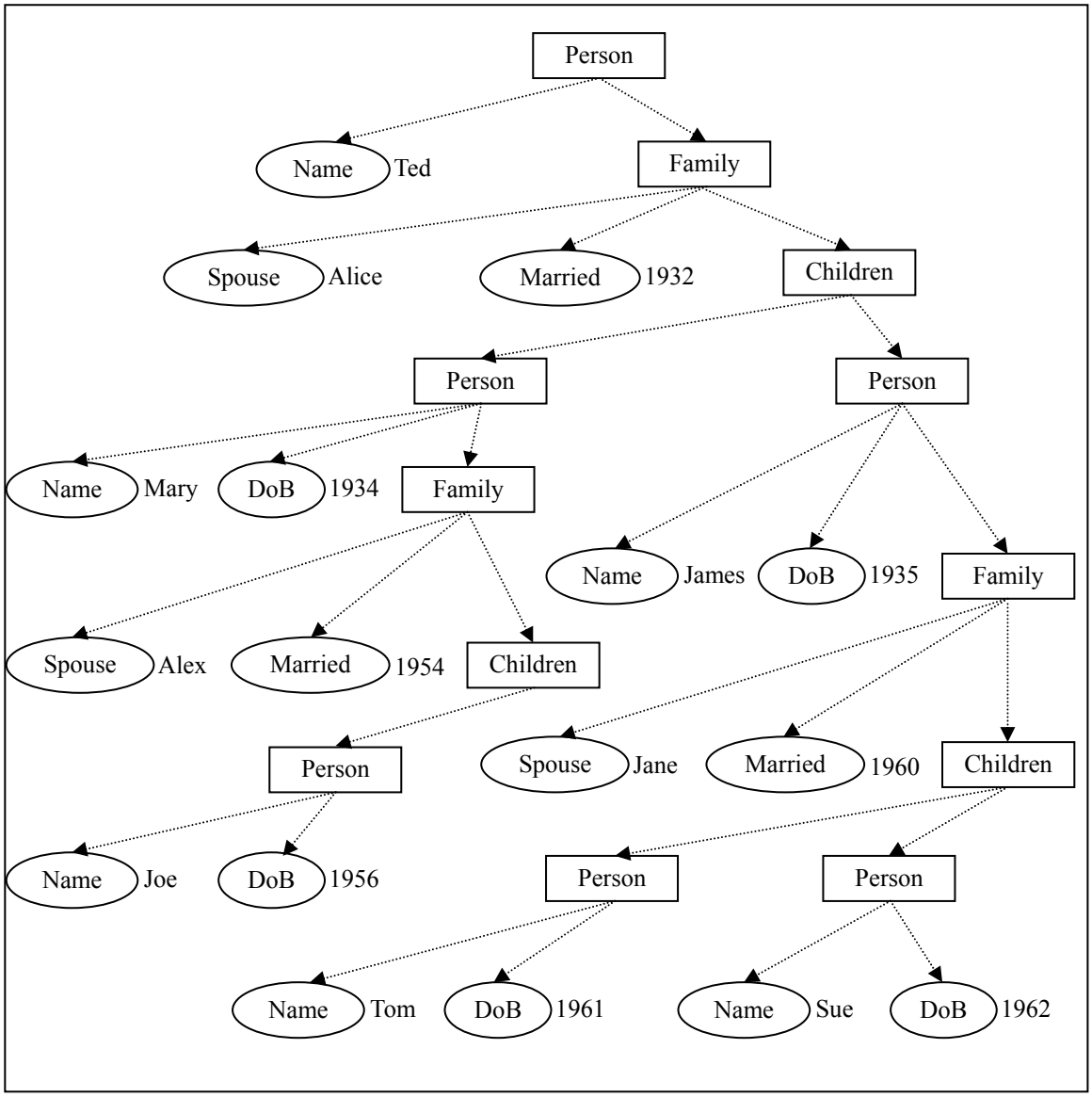

Figure 1.4 A Semi-structured Family Tree

When querying semi-structured data, we can not expect users to be fully aware of the data structure. Thus, it is important not to require complete knowledge of the structure to express meaningful queries [Abit97b]. In Aldat, the following path expression is able to answer the query "find all members' names in the family".

```
Person/(./)*Name
```

At the same time, Aldat also allows the retrieval of specific data if the precise position is known, e.g. to extract the name of Ted's spouse by specifying the path

```
Person/Family/Spouse
```

Therefore, extending Aldat to manipulate text can exploit its expressive power in complicated data inquiry and computation, and will address the needs of semi-structured/unstructured data manipulation in the text/relational database.

1.3 Thesis Outline

This thesis introduces an Aldat extension for text management, and its implementation on JRelix. In the first chapter, we will describe our research background on text/relational database systems, and present the motivation of integrating text manipulation into the JRelix system which is an incarnation of Aldat. In the second chapter, a brief tutorial of JRelix will be given. It does not cover every aspect of the system, instead, it provides the fundamental knowledge of JRelix operations, which is a base for understanding the discussions in the following chapters. In the third chapter, we will introduce the extended operations in JRelix in the form of a user's manual for database programmers. In the fourth chapter, we will present their implementation details and in the final chapter, a summary will be given for the thesis, together with the prospect of future research.

Chapter 2

JRelix System Overview

The purpose of this chapter is to provide a brief tutorial of the JRelix system, on which our enhancement extension will be based. Section 2.1 helps the users to get started by presenting the running, the element declarations and some other useful commands in JRelix. Section 2.2 talks about the initializations and assignments. Section 2.3 introduces in brief the views in JRelix. In sections 2.4 and 2.5, we navigate the readers through the usage of relational algebra and domain algebra. In the rest of the chapter, we describe a newly implemented syntactic sugar which simplifies the queries on the nested relations.

2.1 Getting Started

2.1.1 Starting and Terminating JRelix

JRelix is implemented in Java. It runs in a directory which it uses for data storage. To start JRelix in the directory, the following command is typed on the command line of the operating system.

```
% java JRelix
```

If the JRelix system is successfully booted, the copyright information will be displayed in its run-time environment and a prompt (“>”) will be given for user input.

```
Starting stand alone JRelix.
+-------------------------------------------+
|            Relix Java version 0.90        |
| Copyright (c) 1997 -- 2003 Aldat Lab      |
|         School of Computer Science        |
|             McGill University             |
+-------------------------------------------+
>
```

Figure 2.1 JRelix startup

To terminate the JRelix system, the command is “quit;”. Consequently, the JRelix environment will be safely shut down, and control will be returned to the operating system.

2.1.2 Element Declarations

- Domain Declaration

Atomic domains in JRelix are declared with the keyword *domain* in the following syntax:

Syntax: domain <*id_list*> <*type*>;

Parameters:

id_list – A list of domains to be declared. Domains in the list are separated by commas.

type – The type of these domains to be declared. There are nine primary domain types valid in JRelix, i.e. *integer*, *short*, *long*, *double*, *float*, *string*, *boolean*, *attribute*, *universal* and *numeric*. They are consistent with the data types of the same names in Java, should they exist.

In addition to the atomic domain types, a complex domain type namely the relation

domain, can be defined in JRelix. A relation domain works as a domain and a relation at the same time, and is a useful tool to build nested relations. It takes the following syntax in declaration.

Syntax: domain <*domain_name*>(<*id_list*>);

Parameters:

domain_name – The name of the relation domain to be declared.

id_list – The domains in the new relation domain.

Figure 2.2 gives some examples of domain declarations and their explanations.

```
>domain capacity integer;            // integer domain
>domain credits float;               // float domain
>domain title, instructor string;    // string domains
>domain Assts(asstId, mark);         // relation domain
```

Figure 2.2 Examples of domain declarations

- Relation Declaration

Relations are defined on domains (or called attributes) which have already been declared. The domains in relation can be atomic or complex domains. The declaration of relation follows the syntax below,

Syntax: relation <*rel_name*>(<*id_list*>) [<*initialization*>];

Parameters:

rel_name – The name of the relation to be declared.

id_list – A list of domains on which the relation is defined.

initialization – The statement of relation initialization. It adopts the curly bracket syntax in which relations start and end with curly brackets ("{" and "}"), while each tuple is surrounded by round brackets ("(" and ")") and different tuples are separated by commas (",").

Figure 2.3 gives an example of relation declarations which generate the relation *Courses*

shown in figure 2.4.

```
>domain name, title, instructor strg;
>domain capacity intg;
>domain credits float;
>relation Courses(name, credits, title, capacity, instructor) <-
 {("COMP102", 3.0, "Computers and Computing", 45, "Ciaravola"),
  ("COMP208", 3.0, "Computers in Engineering", 100, "Ratzer"),
  ("COMP251", 3.0, "Data Structures", 45, "Crepeau"),
  ("COMP302", 3.0, "Programming Languages", 68, "Friedman"),
  ("COMP310", 3.0, "Computer Systems", 70, "Maheswaran"),
  ("COMP540", 3.0, "Matrix Computations", 25, "Chang"),
  ("COMP642", 4.0, "Numerical Estimation", 15, "Chang"),
  ("COMP644", 4.0, "Pattern Recognition", 25, "Aloupis")
 };
```

Figure 2.3 An example of relation declarations

```
Courses
(name       credits     title                    capacity    instructor)
 COMP102    3.0         Computers and Computing  45          Ciaravola
 COMP208    3.0         Computers in Engineering 100         Ratzer
 COMP251    3.0         Data Structures          45          Crepeau
 COMP302    3.0         Programming Languages    68          Friedman
 COMP310    3.0         Computer Systems         70          Maheswaran
 COMP540    3.0         Matrix Computations      25          Chang
 COMP642    4.0         Numerical Estimation     15          Chang
 COMP644    4.0         Pattern Recognition      25          Aloupis
```

Figure 2.4 The *Courses* relation

The relations which contain other relations as attributes are called nested relations. Figure 2.5 gives an example of nested relation declarations and initialization.

```
>domain studId, asstId strg;
>domain mark float;
>domain Assts(asstId, mark);
>relation StudAsst(studId, Assts) <-
{("001", {("asst1", 22.0), ("asst2", 25.0), ("asst3", 26.0)}),
 ("002", {("asst1", 24.0), ("asst3", 24.5)}),
 ("003", {("asst2", 27.0), ("asst3", 23.0)}),
 ("004", {("asst1", 23.5), ("asst2", 25.0)})
};
```

Figure 2.5 An Example of nested relation declarations and initializations

In this example, domain *Assts* is a relation domain defined on two atomic domains *asstid* and *mark*. It is adopted as an attribute in the relation *StudAsst* and is assigned the values during the declaration and initialization of *StudAsst*. Figure 2.6 shows the nested relation *StudAsst* as the result.

```
StudAsst
(StudId         Assts                )
                (asstId         mark )
 001             asst1          22.0
                 asst2          25.0
                 asst3          26.0
 002             asst1          24.0
                 asst3          24.5
 003             asst2          27.0
                 asst3          23.0
 004             asst1          23.5
                 asst2          25.0
```

Figure 2.6 The nested relation *StudAsst*

2.1.3 Commonly Used Commands

Now we know the way to declare basic elements in JRelix. In this section we will list some commonly used commands for further operations on the database elements.

Command	Operation
trace;	To turn log on/off.
pr <expr>;	To print out the result of the expression or relation.
sr [<rel>];	To display the description of a given relation, or to list all relations in the system if the argument is omitted.
sd [<dom>];	To display the description of a given domain, or to list all domains in the system if the argument is omitted.
dr <rellist>;	To delete relations in the relation list.
dd <domlist>;	To delete domains in the domain list.
srd;	To show the dependence of relations on domains.

2.2 Assignments

It is useful to be able to create new relations from the old ones. JRelix provides two assignment operators, namely replacement assignment ("<-") and incremental assignment ("<+"). The replacement operator completely replaces the left-hand relation, without regard to whether or not it exists. The data in the right-hand relation is copied into the left-hand relation, and the old data in the left-hand relation is destroyed. The incremental operators add new tuples to the left-hand relation, and the attributes of the right-hand relation must be compatible with those of the left-hand relation. The assignment operation allows renaming attributes, but all attributes of the left-hand relation must be specified in the list.

Syntax: <*rel_A*> <- <*rel_B*>; or
<*rel_A*> <+ <*rel_B*>; or
<*rel_A*> [<*attr_list_A*> <- <*attr_list_B*>] <*rel_B*>; or
<*rel_A*> [<*attr_list_A*> <- <*attr_list_B*>] <*rel_B*>;

The usage of assignments in the relation initialization has been shown in figures 2.3 and 2.5. Figure 2.7 gives an example of renaming attributes in the relation.

```
>domain courname, courtitl, courinst strg;
>domain courcred float;
>domain courcapa intg;
>CourDesc [courname, courcred, courtitl, courcapa, courinst <- name, credits, title,
capacity, instructor] Courses;
>pr CourDesc;
+------------+---------------+----------------------+-------------+--------------+
| courname   | courcred      | courtitl             | courcapa    | courinst     |
+------------+---------------+----------------------+-------------+--------------+
| COMP102    | 3.0           | Computers and Comput | 45          | Ciaravola    |
| COMP208    | 3.0           | Computers in Enginee | 100         | Ratzer       |
| COMP251    | 3.0           | Data Structures      | 45          | Crepeau      |
| COMP302    | 3.0           | Programming Language | 68          | Friedman     |
| COMP310    | 3.0           | Computer Systems     | 70          | Maheswaran   |
| COMP540    | 3.0           | Matrix Computations  | 25          | Chang        |
| COMP642    | 4.0           | Numerical Estimation | 15          | Chang        |
| COMP644    | 4.0           | Pattern Recognition  | 25          | Aloupis      |
+------------+---------------+----------------------+-------------+--------------+
relation CourDesc has 8 tuples
```

Figure 2.7 An example of renaming attributes

2.3 Views

While the assignments allow the result of the expression to be evaluated and stored in another relation, it is however, also useful to be able to defer the process until later. The mechanism for this is called a *view* in JRelix, which is notated by replacing the assignment operators with the keyword *is*. The syntax of the view definition is given below,

Syntax: <*rel_name*> [initial <*rel_expr_init*>] is <*rel_expr*>;

Parameters:

rel_name – The name of the relation to be assigned.

rel_expr_init – The expression to be assigned to the view as initial value. This statement is optional and is only useful in recursive view executions.

rel_expr – The expression to be assigned to the view during the evaluation.

No assignment is performed in the declaration of a view. The expression is evaluated and

assigned to the target relation only when a subsequent assignment, or other operation such as *print*, is executed. Figure 2.8 gives an example of view declaration.

```
>CourView is [name, title] in Courses;
```

Figure 2.8 An example of view declaration

In this example, a view *CourView* is declared on the attributes *name* and *title* in the *Courses*. It is assigned by the result of the expression following the key word *is* only when the view is accessed by other operations.

2.4 Relational Algebra

The operators in the relational algebra family work on one or more relations and produce a relation as the result. The operation provides a closure of relations and makes it possible for expressions to be constructed at arbitrary length and in arbitrary complexity. The relational operators and domain operators as well are functional, which means the operation of the relation does not change the value of the relation. According to the number of operands an operator takes, relational algebra is divided into two categories, namely unary operators and binary operators.

2.4.1 Unary Operators

Unary operators take single relations as inputs. In this section we will introduce three types of most useful operators, i.e. *projection*, *selection*, and *T-selection*.

- Projection

Projection creates a new relation on a specified subset of the attributes of the operand. Duplicates will be removed in the result. It takes the following syntax,

Syntax: [<*id_list*>] in <*rel_expr*>;

Parameters:

id_list – The attributes to project on.

rel_expr – The input relational expression.

Please note that if the *id_list* is empty, the operation will return a boolean value, in which false means the result is empty, and vice versa.

- Selection

Selection picks out tuples of a relation according to a boolean condition on the tuple values. The boolean condition can involve arbitrary operations on any attributes of the relation or on any constant values, but it must be able to be evaluated on individual tuples.

It takes the following syntax,

Syntax: where <*bool_cond*> in <*rel_expr*>;

Parameters:

bool_cond – The boolean expression providing criteria for the selection.

rel_expr – The input relational expression.

- T-Selection

Projection and *selection* can be combined in a single operation, called *T-selection*. The syntax is,

Syntax: [<*id_list*>] where <*bool_cond*> in <*rel_expr*>;

Parameters:

id_list – Attributes to be projected on.

bool_cond – Selection criteria.

rel_expr – The input relational expression.

In the following example we apply *T-selection* in the *Courses* relation to the search for the course(s) instructed by Prof. *Chang*.

```
>ChangCour <- [name, title] where instructor = "Chang" in Courses;
>pr ChangCour;
+--------------------+--------------------+
| name               | title              |
+--------------------+--------------------+
| COMP540            | Matrix Computations |
| COMP642            | Numerical Estimation |
+--------------------+--------------------+
relation ChangCour has 2 tuples
```

Figure 2.9 An example of *T-Selection*

2.4.2 Binary Operators

Binary operators accept two relations as input and join them together. There are two categories of binary operators, i.e. μ-joins and σ-joins. Both categories of join operators take the similar syntax:

Syntax: <*left_expr*> <*join_operator*> <*right_expr*>; or

<*left_expr*> [<*id_list*> :<*join_operator*>: <*id_list*>] <*right_expr*>;

Parameters:

join_operator – Join operator.

right_expr – Relational expressions as right-hand operand.

left_expr – Relational expressions as left-hand operand.

id_list – Specified attribute list on which to join.

Following the first syntax, the common attributes of the left and the right hand relations are used as join attributes. In the case when the two relations do not share any common attributes, the user must specify the attributes on which to join, and the second syntax should be taken. We will discuss two categories of join operators separately in the rest of the section,.

➢ μ-joins

The μ-join operations extend the mathematical operations on sets, which include intersection, union and difference. The results of their applications contain the union of the attributes from two operand relations, except difference joins which sometimes lead to the disappearance of certain attributes containing all null values.

In general, μ-join operators can be defined in terms of three components in the result, i.e. the *center*, the *left wing* and the *right wing*. They are defined as the following,

- For relation R(X, Y) and S(Y, Z) sharing a common attribute set,Y
  center(R, S) ≡ {(x, y ,z)|(x, y) ∈ R ∧ (y, z) ∈ S}
  left(R, S) ≡ {(x, y, *DC*)|(x, y) ∈ R ∧ ∀z, (y, z) ∉ S}
  right(R, S) ≡ {(*DC*, y, z)|(y, z) ∈ S ∧ ∀x, (x, y) ∉ R}
- For relation R(W, X) and S(Y, Z) sharing no common attribute set
  center(R, S) ≡ {(w, x, y, z)|(w, x) ∈R ∧ (y, z) ∈ S ∧ x = y}
  left(R, S) ≡ {(w, x, y, *DC*)|(w, x) ∈ R ∧ x = y ∧ ∀z, (y, z) ∉ S}
  right(R, S) ≡ {(*DC*, x, y, z)|(y, z) ∈ S ∧ x = y ∧ ∀x, (x, y) ∉ R}

The symbol *DC* stands for *don't care*, which is a null value defined in JRelix.

Now we are able to give the definition of μ-join operators as shown below,

Operator	**Definition**	**Description**
R **ijoin** S	center(R, S)	intersection join
R **ujoin** S	left(R, S) ∪ center(R, S) ∪ right(R, S)	union join
R **ljoin** S	left(R, S) ∪ center(R, S)	left join
R **rjoin** S	center(R, S) ∪ right(R, S)	right join
R **djoin** S	left(R, S)	difference join
R **drjoin** S	right(R, S)	right difference join
R **sjoin** S	left(R, S) ∪ right(R, S)	symmetric difference join

The following example applies *ijoin* on the relations *Courses* and *ProfOffice* to find out the offices of the course professors.

```
>pr ProfOffice;
+----------------------+----------------------+
| professor            | office               |
+----------------------+----------------------+
| Aloupis              | MC 421               |
| Chang                | MC 303               |
| Ciaravola            | MC 201               |
| Crepeau              | MC 101               |
| Friedman             | MC 411               |
| Kemme                | MC 102               |
| Maheswaran           | MC 106               |
| Ratzer               | MC 215               |
+----------------------+----------------------+
relation ProfOffice has 8 tuples
>CourOffice <- Courses [instructor :ijoin: professor] ProfOffice;
>pr CourOffice;
+------------+---------+---------+----------------------+----------+------------+--------+
| instructor | name    | credits | title                | capacity | professor  | office |
+------------+---------+---------+----------------------+----------+------------+--------+
| Aloupis    | COMP644 | 4.0     | Pattern Recognition  | 25       | Aloupis    | MC 421 |
| Chang      | COMP540 | 3.0     | Matrix Computations  | 25       | Chang      | MC 303 |
| Chang      | COMP642 | 4.0     | Numerical Estimation | 15       | Chang      | MC 303 |
| Ciaravola  | COMP102 | 3.0     | Computers and Comput | 45       | Ciaravola  | MC 201 |
| Crepeau    | COMP251 | 3.0     | Data Structures      | 45       | Crepeau    | MC 101 |
| Friedman   | COMP302 | 3.0     | Programming Language | 68       | Friedman   | MC 411 |
| Maheswaran | COMP310 | 3.0     | Computer Systems     | 70       | Maheswaran | MC 106 |
| Ratzer     | COMP208 | 3.0     | Computers in Enginee | 100      | Ratzer     | MC 215 |
+------------+---------+---------+----------------------+----------+------------+--------+
relation CourOffice has 8 tuples
```

Figure 2.10 An example of μ-join

- σ-joins

The σ-join operations extend the truth-valued comparison operations on sets to relations by applying them to each set of values of the join attribute for each of the other values in the two relations. The result contains symmetric difference of the attributes from two

operand relations.

We can define the σ-joins using the following notation. In relations R(W, X) and S(Y, Z), R_w is the set of values X associated by R with a given value, w, of W, and S_z is the set of values of Y associated by S with a given value, z, of Z. If W and X are disjoint sets of the attributes of R, and Y and Z are disjoint sets of the attributes of S, we can give the definitions as below. X and Y are allowed to be the same set of attributes, but at least they must be compatible attribute sets.

Operator	**Definition**	**Name**	**Description**
R **icomp** S	$\{(w, z)\|R_w \cap S_z \neq \emptyset\}$	natural composition	overlap
R **sep** S	$\{(w, z)\|R_w \cap S_z = \emptyset\}$	empty intersection join	not overlap
R **sup** S	$\{(w, z)\|R_w \supseteq S_z\}$	greater than or equal join	superset
R **gtjoin** S	$\{(w, z)\|R_w \supset S_z\}$	greater than join	proper superset
R **lejoin** S	$\{(w, z)\|R_w \subseteq S_z\}$	less than or equal join	subset
R **ltjoin** S	$\{(w, z)\|R_w \subset S_z\}$	less than join	proper subset
R **eqjoin** S	$\{(w, z)\|R_w = S_z\}$	equal join	equal

To find out the office of the professor of each course, we can also use *icomp*. In the following example, the result relation does not contain the attribute *instructor* nor the attribute *professor* because of the symmetric difference operation of *icomp* applied on the input attributes.

```
>CourOffice <- Courses [instructor :icomp: professor] ProfOffice;
>pr CourOffice;
+------------+----------+----------------------+-------------+----------+
| name       | credits  | title                | capacity    | office   |
+------------+----------+----------------------+-------------+----------+
| COMP102    | 3.0      | Computers and Comput | 45          | MC 201   |
| COMP208    | 3.0      | Computers in Enginee | 100         | MC 215   |
| COMP251    | 3.0      | Data Structures      | 45          | MC 101   |
| COMP302    | 3.0      | Programming Language | 68          | MC 411   |
| COMP310    | 3.0      | Computer Systems     | 70          | MC 106   |
| COMP540    | 3.0      | Matrix Computations  | 25          | MC 303   |
| COMP642    | 4.0      | Numerical Estimation | 15          | MC 303   |
| COMP644    | 4.0      | Pattern Recognition  | 25          | MC 421   |
+------------+----------+----------------------+-------------+----------+
relation CourOffice has 8 tuples
```

Figure 2.11 An example of σ-join

2.5 Domain Algebra

Independent of relational algebra, domain algebra addresses the needs of computation on attributes, e.g. arithmetic, which relational algebra is unable to meet. There are two main types of domain algebra, namely scalar operations and aggregate operations. Scalar operations work within individual tuples while aggregate operations work across tuples. Thus, they can also be thought of as horizontal operations and vertical operations, respectively.

Domain algebra associates a virtual domain with an expression in declaration. The associated expression can be constants, relational expressions, other domain expressions or any valid combination of these elements. Recursive definition of a virtual domain is not permitted.

A virtual domain may appear anywhere an actual domain is expected. It is actualized only when it is referred through the relational algebra. It takes the following syntax in definition,

Syntax: let <*domain_name*> be <*expr*>;

Parameters:

domain_name – The Name of the virtual domain to be defined.

expr – The expression that will actualize the domain.

We will show some examples, which give the general idea of domain algebra in two categories in the rest of the section. To go deep into the operations, please refer to previous work on Aldat [Yuan98].

2.5.1 Horizontal Operations

Horizontal operations combine attribute values in each tuple using scalar operators. For example, we can concatenate the fields *name* and *title* in the *Courses* relation into one field with the following commands.

```
>let course be name cat " " cat title;
>CompCour <- [course, credits, capacity, instructor] in Courses;
>pr CompCour;
+---------------------------------+----------+-------------+-----------------+
| course                          | credits  | capacity    | instructor      |
+---------------------------------+----------+-------------+-----------------+
| COMP102 Computers and Computing | 3.0      | 45          | Ciaravola       |
| COMP208 Computers in Engineering| 3.0      | 100         | Ratzer          |
| COMP251 Data Structures         | 3.0      | 45          | Crepeau         |
| COMP302 Programming Languages   | 3.0      | 68          | Friedman        |
| COMP310 Computer Systems        | 3.0      | 70          | Maheswaran      |
| COMP540 Matrix Computations     | 3.0      | 25          | Chang           |
| COMP642 Numerical Estimation    | 4.0      | 15          | Chang           |
| COMP644 Pattern Recognition     | 4.0      | 25          | Aloupis         |
+---------------------------------+----------+-------------+-----------------+
relation CompCour has 8 tuples
```

Figure 2.12 An example of horizontal operation

2.5.2 Vertical Operations

Vertical operations work across tuples and come up with summaries or orders of the tuples in the relation. They are divided into *reduction* and *functional mapping* operations. The difference between these two types is whether the tuple order is needed. Each type is further divided into two subtypes, i.e. one with grouping functions and another without.

The following example sums the number of courses by credit in the *Courses* relation.

```
>let numcour be equiv + of 1 by credits;
>CourByCred <- [credits, numcour] in Courses;
>pr CourByCred;
+---------------+-------------+
| credits       | numcour     |
+---------------+-------------+
| 3.0           | 6           |
| 4.0           | 2           |
+---------------+-------------+
```

Figure 2.13 An example of vertical operation

2.6 Syntactic Sugar for Nested Queries

Previous research has presented different approaches to nested relation operations, such as query, update and level lifting [Merr84, He97]. As we can see in the preceding sections, nested relations are subsumed as domains in nesting. Recent work introduced a syntactic sugar for nested relation queries, which regards projection domain and relational expression as paths towards the low level elements. The syntactic sugar can further couple with regular expressions and provides an intuitive means of handling complex structured data.

Path expression takes the following syntax,

Syntax: [<*rel_name*>/] (<*nested_path_expr*>/)* [<*domain_name*>];

Parameters:

rel_name – Top level relation name.

nested_path_expr – Nested path expression, which can be regular expressions.

domain_name – Query domain name.

The regular expression adopted by path expression provide Kleene star ("*") which means any occurrence, plus sign ("+") which means one or more occurrence, question mark ("?") which means zero or one occurrence and period (".") which matches any path. For example, "(/name)*" matches a null expression or any repeat of the path expression "/name", "(/name)+" matches more than one repeat of "/name", "(/name)?" matches a null expression or "/name" and ".*" matches any length of the path expression.

In the rest of the section, we will now present some examples based on the relation *StudAsst* that we have already seen in section 2.1.2. In the first example, we retrieve all assignment ids in the nested relation *Assts*. As we can see, the level of the attribute *asstId* is raised by the path expression and it becomes an attribute in a top level relation.

StudAsst		
(StudId	Assts	)
	(asstId	mark)
001	asst1	22.0
	asst2	25.0
	asst3	26.0
002	asst1	24.0
	asst3	24.5
003	asst2	27.0
	asst3	23.0
004	asst1	23.5
	asst2	25.0

Figure 2.14 The relation StudAsst

```
>AsstIds <- Assts/asstId in StudAsst;
>pr AsstIds;
+----------------------+
| asstId               |
+----------------------+
| asst1                |
| asst2                |
| asst3                |
+----------------------+
relation AsstIds has 3 tuples
```

Figure 2.15 Nested retrieval with path expression

In the second example, we query the marks for assignment 1. It gives us the idea that we can operate a nested relation as a top-level relation with the powerful path expressions.

```
>Asst1Mark <- [mark] where asstId = "asst1" in StudAsst/Assts;
>pr Asst1Mark;
+---------------+
| mark          |
+---------------+
| 22.0          |
| 23.5          |
| 24.0          |
+---------------+
relation Asst1Mark has 3 tuples
```

Figure 2.16 A nested query with path expression

Chapter 3

Users' Manual for Text Operations

The feature of handling text is added into the JRelix system by text operators. These operators accept plain text as the input and generate a new text or relation as the output. In this chapter, the users' manual of text operators in the JRelix system is presented. Section 3.1 describes the types of valid input and basic text operations. It gives the details of the text definition, initialization, information retrieval and deletion. Section 3.2 introduces six basic binary grep operators on the text, namely *igrep*, *ugrep*, *dgrep*, *sgrep* or *diff*, *lgrep* and *rgrep*. Section 3.3 presents another operator, called *text2attr*, which breaks a text into tokens and turns them into tuples in a relation. Section 3.4 ends the chapter by introducing the *mu2nest* operator which is used to handle marked-up text, e.g. XML and HTML.

3.1 Basic Text Operations

Compared with relations, text is a sequence of elements without regular repetitions which can be captured and converted into tuples. The valid input to the *text* type element in the JRelix system is plain text, in other words, textual data in the ASCII format. The plain text on Unix or Linux with the line-feed "\n", and that on the MS Windows systems with

line-feed "\r\n" are both acceptable. JRelix unifies the text formats with line-feed "\n" before it becomes a JRelix text.

3.1.1 An Example

In this example, we are going to import a text file *person.txt* into JRelix, which resides under the path "*/home/user/jxie4/thesis/person.txt*". The format of *person.txt* is ASCII plain text. We define a JRelix text named *person* and initialize it with the *person.txt* file.

```
>text person <- "/home/user/jxie4/thesis/person.txt";
```

The text *person* is imported to JRelix.

```
>st;
---------------------------------- Text Table ----------------------------------
Name                Length              Content
--------------------------------------------------------------------------------
person              203                 Ted married Alice in 1932. Their childre
--------------------------------------------------------------------------------
>pr person;
--------------------------------------------------------------------------------
                                   Text person
--------------------------------------------------------------------------------
Ted married Alice in 1932. Their children, Mary (1934) married Alex in 1954 (Joe
was born to Mary and Alex in 1956) and James (1935) married Jane in 1960 (James
and Jane had Tom in 1961 and Sue in 1962).
--------------------------------------------------------------------------------
Text person has 203 characters
```

3.1.2 Text Definition and Initialization

A JRelix text must be defined and initialized before using. A text can only be initialized when it is defined. Its content can be assigned by a plain text file on the disk, by a relation or by another text.

3.1.2.1 Define a Text

Syntax: text <*text_name*>;

The simplest way to define a text is to define a text without initialization. With the above command, JRelix defines a text called *text_name* and sets its length to 0.

Example:

```
>text person;
>st;
---------------------------------- Text Table ----------------------------------
Name                Length              Content
--------------------------------------------------------------------------------
person              0
--------------------------------------------------------------------------------
```

3.1.2.2 Initialize with a Plain Text File

JRelix also allows users to initialize a text with an ASCII file on the local disk. An absolute file pathname should be given and surrounded by double quotation marks.

Syntax: text <*text_name*> <- <*full_file_path*>;

full_file_path is a pathname string surrounded by quotation marks. The text file is copied from the given path and the file length will be calculated.

Note: JRelix accepts absolute file pathnames on both Unix platforms and MS Windows platforms. For UNIX platforms, the prefix of an absolute pathname is "/". For Windows platforms, the prefix of an absolute pathname is encoded as "\\".

Example:

Unix -

```
>text person <- "/home/user/jxie4/thesis/person.txt";
```

Windows -

```
>text person <- "C:\\JRelix\\person.txt";
```

3.1.2.3 Initialize with another Text or Relation

A new text can also be assigned by a JRelix text or relation that already exists. JRelix transforms the format of the input text/relation and assigns it to the new text.

Syntax: text <*text_name*> <- <*other_text_or_relation*>;

Example:

In the first example we initialize a text *newperson* with the text *person*.

```
>text newperson <- person;
>pr newperson;
--------------------------------------------------------------------------------
                                Text newperson
--------------------------------------------------------------------------------
Ted married Alice in 1932. Their children, Mary (1934) married Alex in 1954 (Joe
was born to Mary and Alex in 1956) and James (1935) married Jane in 1960 (James
and Jane had Tom in 1961 and Sue in 1962).
--------------------------------------------------------------------------------
Text newperson has 203 characters
```

In the second example we create a text *BoMtext* from the relation *BoM*.

```
>pr BoM;
+--------------------+------------+--------------------+
| assembly           | qty        | subassembly        |
+--------------------+------------+--------------------+
| cover              | 1          | plate              |
| cover              | 2          | screw              |
| fixture            | 2          | plug               |
| fixture            | 2          | screw              |
| plug               | 1          | mould              |
| plug               | 2          | connector          |
| wallplug           | 1          | cover              |
| wallplug           | 1          | fixture            |
+--------------------+------------+--------------------+
relation BoM has 8 tuples
>text BoMtext <- BoM;
>pr BoMtext;
--------------------------------------------------------------------------------
                                Text BoMtext
--------------------------------------------------------------------------------
cover   1       plate
cover   2       screw
```

```
fixture 2       plug
fixture 2       screw
plug    1       mould
plug    2       connector
wallplug        1       cover
wallplug        1       fixture
-------------------------------------------------------------------------------
Text BoMtext has 132 characters
>st;
---------------------------------- Text Entry ----------------------------------
Name                Length              Content
-------------------------------------------------------------------------------
person              203                 Ted married Alice in 1932. Their childre
newperson           203                 Ted married Alice in 1932. Their childre
BoMtext             132                 cover 1 plate cover 2 screw fixture 2 pl
-------------------------------------------------------------------------------
```

3.1.3 Text Information Retrieval and Deletion

3.1.3.1 Print Command

For the convenience of JRelix programmers, it is possible to use the same command *pr* to print out the content of the texts as the relations. The print text command, *pt*, is also acceptable.

Syntax: pr 〈*text_name*〉;

or pt 〈*text_name*〉;

Note: Some JRelix operators accept both relations and texts as parameters. To avoid the ambiguity in using these operators, it is NOT allowed to define relations and texts with the same name. Thus using command *pr* to print out the content of the texts is not ambiguous to the programmers or to the JRelix system.

3.1.3.2 Show Text Command

In the same way that the command *sr* retrieves the information of the relations in JRelix, *sr* or *st* has the same effect on texts.

Syntax: sr <*text_name*>;

or st [<*text_name*>];

JRelix prints out the name, the length and part of the content of the text. The *text_name* parameter in the command *st* is optional. If it is absent, JRelix will retrieve the information of all texts in the system.

Note: If we use the command *sr* without giving the *text_name*, JRelix will only list all the relations instead of texts in the system.

3.1.3.2 Delete Text Command

Unlike relations which are composed of domains and relations, texts are not attributes of any other elements. A programmer can delete or redefine a text at any time. The syntax of deleting a text is

Syntax: dr <*text_name*>;

or dt <*text_name*>;

Note: It is not necessary to delete a text before defining a new one with the same name. JRelix automatically replaces the old one with the definition of the new text.

3.2 Regular Expressions

3.2.1 Introduction

Text operators are integrated into JRelix to retrieve interesting information from text. Before starting to discuss text operators, it is essential to introduce regular expressions which will be used in our text operators.

Basically, regular expressions are a way to describe a set of strings based on common characteristics shared by each string in the set. They can be used as a tool to search, edit or manipulate text or data. For example, the expression "Andrew" matches any substring

"Andrew" and the expression "A.*w" matches any substring starting with the letter "A" and ending with letter "w".

There are many different formulations of regular expressions we can choose from, such as grep, Perl, JDK, Python and PHP. The one adopted herein is Java Regular Expression. Its syntax is similar to the Unix grep. Compared with the line-based pattern matching in Unix grep, Java regular expressions have higher flexibility. A match can be individual characters, words, sentences across lines or the whole text.

Note:

1. In the implementations of the eight text operators, grep can be performed across lines and paragraphs without specifying "\n" in their patterns. In other words, they can match any characters with the wildcard "." and do not need to worry whether the matching string is separated by line-feed or not.
2. The pattern-matching in the text operators is case sensitive.
3. Although a match can extend across lines, the expressions ^ and $ still match just after or just before, respectively, a line terminator or the end of the input sequence.

3.2.2 Regular Expression Syntax

Now that a general idea of what regular expressions are and why they are used has been established, it is important to explain their syntax more clearly. The syntax of Java regular expressions is similar to that of Unix grep. Most of the grep expressions we already known can be applied on the patterns of the text operators. It is only necessary to briefly mention the regular expression syntax which the programmers are likely to use and which is different from Unix grep.

- **Characters**

Construct	Matches
x	The character x
\\	The backslash character
\t	The tab character
\n	The newline (line feed) character
\r	The carriage-return character

The character expressions are used to match individual characters.

Node: The carriage-return character (\r) is eliminated from the text by JRelix before it is imported into the system. Thus the pattern "\r" will match nothing in the texts.

- **Character classes**

Construct	Matches
[abc]	a, b, or c (simple class)
[^abc]	Any character except a, b, or c (negation)
[a-zA-Z]	a through z or A through Z, inclusive (range)
[a-d[m-p]]	a through d, or m through p: [a-dm-p] (union)
[a-z&&[def]]	d, e, or f (intersection)
[a-z&&[^bc]]	a through z, except for b and c: [ad-z] (subtraction)
[a-z&&[^m-p]]	a through z, and not m through p: [a-lq-z](subtraction)

- **Predefined character classes**

Construct	Matches
.	Any character (also match the line terminators in our implementation)
\d	A digit: *[0-9]*
\D	A non-digit: *[^0-9]*
\s	A whitespace character: *[\t\n\x0B\f\r]*
\S	A non-whitespace character: *[^\s]*
\w	A word character: *[a-zA-Z_0-9]*
\W	A non-word character: *[^\w]*

Predefined character classes offer the wildcards that match a group of characters which are commonly used. For example, "\w+" matches any words and "\d+" matches any sequence of digits in the text.

♦ **POSIX character classes (US-ASCII only)**

Construct	Matches
\p{Lower}	A lower-case alphabetic character: *[a-z]*
\p{Upper}	An upper-case alphabetic character:*[A-Z]*
\p{ASCII}	All ASCII:*[\x00-\x7F]*
\p{Alpha}	An alphabetic character:*[\p{Lower}\p{Upper}]*
\p{Digit}	A decimal digit: *[0-9]*
\p{Alnum}	An alphanumeric character:*[\p{Alpha}\p{Digit}]*
\p{Punct}	Punctuation: One of !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
\p{Graph}	A visible character: *[\p{Alnum}\p{Punct}]*
\p{Print}	A printable character: *[\p{Graph}]*
\p{Blank}	A space or a tab: *[\t]*
\p{Cntrl}	A control character: *[\x00-\x1F\x7F]*
\p{XDigit}	A hexadecimal digit: *[0-9a-fA-F]*
\p{Space}	A whitespace character: *[\t\n\x0B\f\r]*

♦ **Boundary matchers**

Construct	Matches
^	The beginning of a line
$	The end of a line

♦ **Logical operators**

Construct	Matches
XY	X followed by Y
X|Y	Either X or Y
(X)	X, as a capturing group

- **Quantifiers**

Greedy	Reluctant	Possessive	Meaning
x?	*x??*	*x?+*	x, once or not at all
*x**	*x*?*	*x*+*	x, zero or more times
x+	*x+?*	*x++*	x, one or more times
x{n}	*x{n}?*	*x{n}+*	x, exactly n times
x{n,}	*x{n,}?*	*x{n,}+*	x, at least n times
x{n,m}	*x{n,m}?*	*x{n,m}+*	x, at least n but not more than m times

More convenient than the Unix grep, Java regex returns character sequences of any length instead of a whole line. Quantifiers make it possible to specify the length of the sequence to be matched. There are three kinds of quantifiers, namely greedy, reluctant, and possessive quantifiers.

Greedy quantifiers eat the entire input string prior to attempting the first match. If the attempt to match the entire input string fails, the matcher backs off the input string character by character until a match is found or there are no more characters left from which to back off.

In contrast to the greedy quantifiers, the reluctant quantifiers start at the beginning of the input string and then reluctantly eat one character at a time looking for a match. The last thing they try is the entire input string.

In the same manner as the greedy quantifiers, the possessive quantifiers read in the whole input string and try to find a match. Unlike the greedy quantifiers, they do not back off if they fail to match and return with finding nothing.

The most useful quantifiers to the programmers are the greedy quantifiers and the reluctant quantifiers, which return the longest and shortest matches respectively. For

example, the greedy expression "<.*>" matches the whole XML file and the reluctant expression "<.*?>" matches the individual XML tags.

- **Quoting**

The backslash character ("\") serves to introduce escaped constructs as well as to quote characters which otherwise would be interpreted as unescaped constructs. Thus the expression "\\" matches a single backslash, "\." and "\?" match a full stop (".") and a question mark ("?") respectively, and pattern ".*?(\.|\?|!)" matches a sentence.

Some general ideas about the regular expressions to be used in the text operators have now been made known. Similarly, the most useful patterns for the JRelix programmers in handling the text and the new Java regular expression syntax beyond Unix grep have been discussed. More examples will be shown when we introduce the individual operators in the following sections.

3.3 Binary Grep Operators

A new set of relational operators are suggested by the need of textual pattern matching. It would be opportune to be able to find a substring or a match for a more general regular expression in plain text. The first six operators are called binary grep operators. They supply a matching pattern and defined attributes for the result relation, which binary greps associate with the proper values according to the types of the attributes. Basically, the input of a binary grep is two relevant texts, and the output is a joined relation containing the information retrieved from the texts by the matching pattern.

The six members in the binary grep family, namely the *igrep*, *ugrep*, *dgrep*, *sgrep*, *lgrep* and *rgrep*, share similar names. Like the names of the join operators, the first letters of the names of the binary operators indicate the types of relational operations being done on the result of the grepping. For example, *igrep* means the result is the intersection of two greps, and *sgrep* means the result is the symmetric difference of two greps.

The six binary grep operators follow the similar syntax which is given below.

Syntax: 〈*text1*〉 xgrep(〈*pattern*〉 (, 〈*attr*〉)+) 〈*text2*〉

x stands for *i*, *u*, *d*, *s*, *l* or *r*, respectively.

Parameters:

text – The input texts as the source of the grep.

pattern – The match pattern, which will be applied on both texts.

attr - The domains predefined by the user, which will be used as the attributes in the result relation. Users can specify two kinds of attributes, namely value and position, for the results from two texts. *Value*s are the string-type attributes used for the matched strings in the texts and *position*s are the integer-type attributes used for the starting position of the strings. The positions of the attribute parameters within their types determine the source of their value and the sequence of their occurrence in the result relation. That is to say, the first attributes of string type and integer type will be used for the matches and their positions in the left-hand text and the second attributes will be used for the right-hand text. The second *position* parameter can be omitted. More attributes than the first two strings and the two integers will be ignored.

Some examples for the individual bi-grep operators will be shown now. Their usage and meaning will be discussed in the following sections.

3.3.1 Intersection Grep

The first binary grep, namely intersection grep or *igrep*, applies the pattern to the texts separately and presents the intersection of the results to the user. *i* means that an intersection operation is performed. It also serves to distinguish this binary operator from the unary grep and other operators in the bi-grep family. Please note the following example.

Given two texts, Jtext and Stext,

```
>pr Jtext;
--------------------------------------------------------------------------------
                                    Text Jtext
--------------------------------------------------------------------------------
On his way to work,
Joe met Sue. "Let's
go out tonight", he invited
her. After work, he met her
at her apartment and they
went to a movie
which he enjoyed a lot.
--------------------------------------------------------------------------------
>pr Stext;
--------------------------------------------------------------------------------
                                    Text Stext
--------------------------------------------------------------------------------
Sitting in a movie
with Joe, Sue
wondered why she had accepted his
invitation. She had just
started to paint her apartment
and did not really have time.
--------------------------------------------------------------------------------
>domain val1, val2 strg;
>domain pos1, pos2 intg;
>JStextlink <- Jtext igrep("\w+", pos1, val1, val2, pos2) Stext;
>pr JStextlink;
+------------+--------------------+--------------------+------------+
| pos1       | val1               | val2               | pos2       |
+------------+--------------------+--------------------+------------+
| 3          | his                | his                | 65         |
| 11         | to                 | to                 | 104        |
| 21         | Joe                | Joe                | 25         |
| 29         | Sue                | Sue                | 30         |
| 71         | her                | her                | 113        |
| 95         | her                | her                | 113        |
| 103        | her                | her                | 113        |
| 107        | apartment          | apartment          | 117        |
| 117        | and                | and                | 128        |
| 132        | to                 | to                 | 104        |
| 135        | a                  | a                  | 11         |
| 137        | movie              | movie              | 13         |
| 161        | a                  | a                  | 11         |
+------------+--------------------+--------------------+------------+
relation JStextlink has 13 tuples
```

igrep returns a relation containing the common words in two texts. The values of *val1* and *val2* come from *Jtext* (left-hand operant) and *Stext* (right-hand operant) respectively. *pos1* and *pos2* give the positions of the words in the texts. The sequence of the attributes in the relation comes out the same as specified by the programmer.

We noted that the pattern "\w+" is used to grep individual words. That is to say, "\w" is used to match any single letter, and "+" is to tell the operator to concatenate the consecutive letters together and return them as a word.

3.3.2 Union Grep

The remainder of the binary greps follows similar lines. The union grep, *ugrep*, applies the union operation on the grep results. It retains all grepped strings from two texts, even the strings that have no matches in another text. The positions of the unmatched strings are set to null values, *DC*, in the result.

The following is an example with the same input Jtext and Stext as in section 3.3.1.

```
>JStextunion <- Jtext ugrep("\w+", pos1, val1, val2, pos2) Stext;
>pr JStextunion;
```

pos1	val1	val2	pos2
dc	dc	She	82
dc	dc	Sitting	0
dc	dc	accepted	56
dc	dc	did	132
dc	dc	had	52
dc	dc	had	86
dc	dc	have	147
dc	dc	in	8
dc	dc	invitation	70
dc	dc	just	90
dc	dc	not	136
dc	dc	paint	107
dc	dc	really	140
dc	dc	she	48
dc	dc	started	96

dc	dc	time	152
dc	dc	why	44
dc	dc	with	20
dc	dc	wondered	35
0	On	dc	dc
3	his	his	65
7	way	dc	dc
11	to	to	104
14	work	dc	dc
21	Joe	Joe	25
25	met	dc	dc
29	Sue	Sue	30
35	Let	dc	dc
39	s	dc	dc
42	go	dc	dc
45	out	dc	dc
49	tonight	dc	dc
59	he	dc	dc
62	invited	dc	dc
71	her	her	113
76	After	dc	dc
82	work	dc	dc
88	he	dc	dc
91	met	dc	dc
95	her	her	113
100	at	dc	dc
103	her	her	113
107	apartment	apartment	117
117	and	and	128
121	they	dc	dc
127	went	dc	dc
132	to	to	104
135	a	a	11
137	movie	movie	13
144	which	dc	dc
150	he	dc	dc
153	enjoyed	dc	dc
161	a	a	11
163	lot	dc	dc

relation JStextunion has 54 tuples

Similarly to what we did in *igrep*, the pattern "\w+" is used to grep individual words into a relation. The values of *val1* and *val2* are necessarily the same because the union

operation is applied on the grepped strings. The value of the position attribute tells whether the string appears in the text: If the string appears, the position value gives the position of the string; otherwise, it gives a null value which indicates there is no match for the string in the text. Unlike the way igrep discards the unmatched strings, the result of ugrep contains all data retrieved from both texts.

3.3.3 Difference Grep

Similar to the *djoin* operator in the binary join family, the difference grep operator, called *dgrep*, returns the difference of the grepped strings from two texts. That is to say, dgrep subtracts the grepped string set of the right-hand text from that of the left-hand text, and presents it as the result.

In order to show the function of *dgrep*, the text Stext is revised a little bit and have another text Stext'. Please note that the difference between Stext and Stext' only exists in the second and third lines.

```
>pr Stext';
--------------------------------------------------------------------------------
                                  Text Stext'
--------------------------------------------------------------------------------
Sitting in a movie
with Joe, Sue wondered
why she had accepted his
invitation. She had just
started to paint her apartment
and did not really have time.
--------------------------------------------------------------------------------
```

And then we look for the difference between Stext and Stext' with the tool *dgrep*.

```
>Stextdgrep <- Stext dgrep(".*?$", pos1, val1, val2, pos2) Stext';
```

```
>pr Stextdgrep;
+-------------+----------------------+
| pos1        | val1                 |
+-------------+----------------------+
| 20          | with Joe, Sue        |
| 35          | wondered why she had |
+-------------+----------------------+
relation Stextdgrep has 2 tuples
```

The result is that the difference exists in the lines starting at the 20th character and the 35th character, that is to say, the second and the third lines. In the same manner as the *djoin* operator, dgrep does not show the attribute with all null values in the result, even if the programmer gives its name in the parameters. In this example, ".*?\n" can also be used as the match pattern because "\n" and "$" are alternatives in current implementation. The question mark ("?") in the pattern tells the pattern matcher to stop at the first occurrence of the line-feed ("\n" or "$"), and return the matched string before going on for another match.

If we do it the other way round, we will get the result presented by Stext', which actually tells the same thing.

```
>Stextdgrep' <- Stext' dgrep(".*?$", pos2, val2, val1, pos1) Stext;
>pr Stextdgrep';
+-------------+----------------------+
| pos2        | val2                 |
+-------------+----------------------+
| 20          | with Joe, Sue wonder |
| 44          | why she had accepted |
+-------------+----------------------+
relation Stextdgrep' has 2 tuples
```

3.3.4 Symmetric Difference Grep

Operator *dgrep* only shows the matches which exist in the left-hand text but not in the right-hand text. Inspired by the *sjoin* operator, we implemented a symmetric difference grep, or *sgrep*, which returns the difference from both sides. In other words, it combines

two difference greps, text1 *dgrep* text2 and text2 *dgrep* text1.

```
>Stextdiff <- Stext sgrep(".*?\n", pos1, val1, val2, pos2) Stext';
>pr Stextdiff;
+------------+--------------------+--------------------+------------+
| pos1       | val1               | val2               | pos2       |
+------------+--------------------+--------------------+------------+
| dc         | dc                 | why she had accepted | 44         |
| dc         | dc                 | with Joe, Sue wonder | 20         |
| 20         | with Joe, Sue      | dc                 | dc         |
| 35         | wondered why she had | dc                 | dc         |
+------------+--------------------+--------------------+------------+
relation Stextdiff has 4 tuples
```

The result is the symmetric difference of the lines in both texts. It comes out the same as the union of the two *dgreps* which can be found in section 3.3.3. As other binary grep operations, the values of *val1* and *val2* are equivalent, and the value of the position tells to which text the matched string belongs.

We can see from this example, that the *sgrep* operator has a similar function to the *diff* command on Unix, which is commonly used to compare and find out the difference between text documents. Thus the *sgrep* operator has an alias: *diff*. The following command will produce the same result as the example above.

```
Stextdiff <- Stext diff(".*?\n", pos1, val1, val2, pos2) Stext';
```

3.3.5 Left Grep

The following two grep operations are sometimes useful when we are concerned about the grep results on one side and the unmatched results on the other side can be left out. What the *lgrep* operator does is to retrieve the strings in the left-hand text and only the related strings in the right-hand text. It is equal to the union of *igrep* and *dgrep*.

We still use the Jtext and Stext to give the example.

```
>JStextleft <- Jtext lgrep("\w+", pos1, val1, val2, pos2) Stext;
```

```
>pr JStextleft;
```

pos1	val1	val2	pos2
0	On	dc	dc
3	his	his	65
7	way	dc	dc
11	to	to	104
14	work	dc	dc
21	Joe	Joe	25
25	met	dc	dc
29	Sue	Sue	30
35	Let	dc	dc
39	s	dc	dc
42	go	dc	dc
45	out	dc	dc
49	tonight	dc	dc
59	he	dc	dc
62	invited	dc	dc
71	her	her	113
76	After	dc	dc
82	work	dc	dc
88	he	dc	dc
91	met	dc	dc
95	her	her	113
100	at	dc	dc
103	her	her	113
107	apartment	apartment	117
117	and	and	128
121	they	dc	dc
127	went	dc	dc
132	to	to	104
135	a	a	11
137	movie	movie	13
144	which	dc	dc
150	he	dc	dc
153	enjoyed	dc	dc
161	a	a	11
163	lot	dc	dc

```
relation JStextleft has 35 tuples
```

We can find that the number of tuples in the result is exactly the number of words in Jtext. Compared to the *ugrep* operation, there is no null value in the position on the left side in

the *lgrep* operation, which means the unmatched strings on the right side are not included in the result.

3.3.6 Right Grep

The Right grep is the reverse operation of left grep: it returns all the grepped strings from the right-hand text and only the matched strings from the left-hand grep. It is equal to the union of the intersection grep and right difference grep.

Please see the following example:

```
>JStextright <- Jtext rgrep("\w+", pos1, val1, val2, pos2) Stext;
>pr JStextright;
```

pos1	val1	val2	pos2
dc	dc	She	82
dc	dc	Sitting	0
dc	dc	accepted	56
dc	dc	did	132
dc	dc	had	52
dc	dc	had	86
dc	dc	have	147
dc	dc	in	8
dc	dc	invitation	70
dc	dc	just	90
dc	dc	not	136
dc	dc	paint	107
dc	dc	really	140
dc	dc	she	48
dc	dc	started	96
dc	dc	time	152
dc	dc	why	44
dc	dc	with	20
dc	dc	wondered	35
3	his	his	65
11	to	to	104
21	Joe	Joe	25
29	Sue	Sue	30
71	her	her	113

```
| 95          | her                  | her                  | 113         |
| 103         | her                  | her                  | 113         |
| 107         | apartment            | apartment            | 117         |
| 117         | and                  | and                  | 128         |
| 132         | to                   | to                   | 104         |
| 135         | a                    | a                    | 11          |
| 137         | movie                | movie                | 13          |
| 161         | a                    | a                    | 11          |
+-------------+----------------------+----------------------+-------------+
relation JStextright has 32 tuples
```

As we have expected, the number of tuples in the result is equal to the number of words in Stext. It is useful when we want to focus on the words in the right-hand text and throw away the noisy words in the left-hand text.

3.4 Text-To-Attribute Operator

Basically, a text is a sequence of characters, whose meaning depends on the order of its elements. It does not have intrinsic repetition that can be caught and turned into tuples. However, if its order is captured by sequencing attributes, text can indeed seem to be a set of tuples. The decision has to be made as to what attributes are going to be used to give rise to the text, e.g. characters and their sequence, words and their sequence, or sentences and their sequence etc. This can be done by implementing a simple text mining tool, which is able to extract the internal structure of the text, and retrieve the desired information (e.g. gene and protein sequences) from the textual documents, as has been discussed in section 1.1.4.

The *text2attr* operator is an operator on text which creates a relation containing text elements and their sequence. It can also be used in domain algebra, which works on text attributes and produces a nested relation. *text2attr* takes parameters in groups. Each group allows the programmers to retrieve one level of the elements, such as characters, words or sentences. It contains patterns, strings, and integers as parameters, which provide the matching pattern, the name of the element attribute and the name of the

element sequence attribute to the operator respectively. Either, but not both, of the element name and element sequence name is optional, in the case where only the element or only the sequence is wanted. At least one group of parameters should be given.

Syntax: text2attr((<*pattern*>, <*attr*>[, <*attr*>])+) <*text*>

Example:

```
>domain sentseq intg;
>domain sent strg;
>sentences <- text2attr(".*?\.", sentseq, sent) Jtext;
>pr sentences;
+------------+-------------------------------------------------------------------+
| sentseq    | sent                                                              |
+------------+-------------------------------------------------------------------+
| 0          | On his way to work, Joe met Sue.                                  |
| 1          |  "Let's go out tonight", he invited her.                          |
| 2          |  After work, he met her at her apartment and they went to a movie |
+------------+-------------------------------------------------------------------+
relation sentences has 3 tuples
```

This is a simple example with only one group of parameters used to capture the sentences in the text. The pattern ".*?\." tells the matcher to grep a sequence of characters that ends with a full stop (“.”). The parameters *sentseq* and *sent* give the names of the element attribute and the element sequence attribute respectively. The *text2attr* operator returns three sentences it found in the text and gives their sequence. The attributes appear in the relation at the sequence specified by the programmer.

For elements that are hierarchically related, *text2attr* can take more than one group of parameters and join different levels of elements together, according to the sequence of the parameters. It is the programmer's responsibility to give the parameters in the correct sequence according to the natural structure of the text, and make the output meaningful. For example, the sequence of “characters, words, sentences” is more meaningful than “characters, sentences, words”.

Example:

```
>domain charseq, wordseq, sentseq intg;
>domain char, word, sent strg;
>hiertext <- text2attr(".", charseq, char, "\w+|\s+|\p{Punct}", wordseq, word, ".*?\.", sentseq,
sent) Jtext;
>pr hiertext;
```

charseq	char	wordseq	word	sentseq	sent
0	O	0	On	0	On his way to work,
1	n	0	On	0	On his way to work,
2		1		0	On his way to work,
3	h	2	his	0	On his way to work,
4	i	2	his	0	On his way to work,
5	s	2	his	0	On his way to work,
6		3		0	On his way to work,
7	w	4	way	0	On his way to work,
8	a	4	way	0	On his way to work,
9	y	4	way	0	On his way to work,
10		5		0	On his way to work,
11	t	6	to	0	On his way to work,
12	o	6	to	0	On his way to work,
13		7		0	On his way to work,
14	w	8	work	0	On his way to work,
15	o	8	work	0	On his way to work,
16	r	8	work	0	On his way to work,
17	k	8	work	0	On his way to work,
18	,	9	,	0	On his way to work,
19		10		0	On his way to work,
20		10		0	On his way to work,
21	J	11	Joe	0	On his way to work,
22	o	11	Joe	0	On his way to work,
23	e	11	Joe	0	On his way to work,
24		12		0	On his way to work,
25	m	13	met	0	On his way to work,
26	e	13	met	0	On his way to work,
27	t	13	met	0	On his way to work,
28		14		0	On his way to work,
29	S	15	Sue	0	On his way to work,
30	u	15	Sue	0	On his way to work,
31	e	15	Sue	0	On his way to work,
32	.	16	.	0	On his way to work,
33		17		1	"Let's go out tonig
34	"	18	"	1	"Let's go out tonig
35	L	19	Let	1	"Let's go out tonig
36	e	19	Let	1	"Let's go out tonig
37	t	19	Let	1	"Let's go out tonig
38	'	20	'	1	"Let's go out tonig
39	s	21	s	1	"Let's go out tonig
...	...	...	...	...	...
153	e	71	enjoyed	2	After work, he met

```
| 154        | n          | 71         | enjoyed       | 2          |  After work, he met  |
| 155        | j          | 71         | enjoyed       | 2          |  After work, he met  |
| 156        | o          | 71         | enjoyed       | 2          |  After work, he met  |
| 157        | y          | 71         | enjoyed       | 2          |  After work, he met  |
| 158        | e          | 71         | enjoyed       | 2          |  After work, he met  |
| 159        | d          | 71         | enjoyed       | 2          |  After work, he met  |
| 160        |            | 72         |               | 2          |  After work, he met  |
| 161        | a          | 73         | a             | 2          |  After work, he met  |
| 162        |            | 74         |               | 2          |  After work, he met  |
| 163        | l          | 75         | lot           | 2          |  After work, he met  |
| 164        | o          | 75         | lot           | 2          |  After work, he met  |
| 165        | t          | 75         | lot           | 2          |  After work, he met  |
| 166        | .          | 76         | .             | 2          |  After work, he met  |
+------------+------------+------------+---------------+------------+----------------------
+
relation hiertext has 167 tuples
```

The pattern "\w+|\s+|\p{Punct}" is used to capture a word, a sequence of writespace characters or a punctuation. Here *text2attr* captures three levels of elements and joins them into a relation according to the sequence of the parameter groups. The number of parameter groups is not limited in *text2attr*, but the result is meaningful only when the captured elements are hierarchically related.

The *text2attr* operator can be applied to text-type or string-type attributes in domain algebra. In the following example *text2attr* is used to parse the words in the attribute *sent*.

Syntax: text2attr((<*pattern*>, <*attr*>[, <*attr*>])+) <*attr_or_text*>

Example:

```
>let words be text2attr("\w+", wordseq, word) sent;
>sentences' <- [sentseq, sent, words] in sentences;
>pr sentences';
+------------+---------------------+---------------------+
| sentseq    | sent                | words               |
+------------+---------------------+---------------------+
| 0          | On his way to work,  | 64                  |
| 1          |  "Let's go out tonig | 65                  |
| 2          |  After work, he met  | 66                  |
+------------+---------------------+---------------------+
relation sentences' has 3 tuples
```

```
>pr .words;
+---------------------+-------------+---------------------+
| .id                 | wordseq     | word                |
+---------------------+-------------+---------------------+
| 64                  | 0           | On                  |
| 64                  | 1           | his                 |
| 64                  | 2           | way                 |
| 64                  | 3           | to                  |
| 64                  | 4           | work                |
| 64                  | 5           | Joe                 |
| 64                  | 6           | met                 |
| 64                  | 7           | Sue                 |
| 65                  | 0           | Let                 |
| 65                  | 1           | s                   |
| 65                  | 2           | go                  |
| 65                  | 3           | out                 |
| 65                  | 4           | tonight             |
| 65                  | 5           | he                  |
| 65                  | 6           | invited             |
| 65                  | 7           | her                 |
| 66                  | 0           | After               |
| 66                  | 1           | work                |
| 66                  | 2           | he                  |
| 66                  | 3           | met                 |
| 66                  | 4           | her                 |
| 66                  | 5           | at                  |
| 66                  | 6           | her                 |
| 66                  | 7           | apartment           |
| 66                  | 8           | and                 |
| 66                  | 9           | they                |
| 66                  | 10          | went                |
| 66                  | 11          | to                  |
| 66                  | 12          | a                   |
| 66                  | 13          | movie               |
| 66                  | 14          | which               |
| 66                  | 15          | he                  |
| 66                  | 16          | enjoyed             |
| 66                  | 17          | a                   |
| 66                  | 18          | lot                 |
+---------------------+-------------+---------------------+
relation .words has 35 tuples
```

The virtual attribute *words* is defined on the string type attributes *sent* and is actualized in the relation *sentences'*. It turns out to be a nested relation containing the words parsed by

text2attr and their sequences.

Application:

With the intrinsic ability to match the implicit patterns and extract the desired information from the text, the *text2attr* operator can also be used as a simple text mining tool. For example, the following text, *CourDesc*, contains descriptions of three courses as shown in figure 3.1.

```
COMP 575 - Fundamentals of Distributed Algorithms
Study of a collection of algorithms that are basic to the world of concurrent
programming...
Prerequisite: COMP 310
Instructor: Carl Tropper

COMP 617 - Information Systems
Seminar course. A major area of application of the techniques covered in 308-612 is
discussed...
Prerequisite: COMP 612
Instructor: Timothy Merrett

COMP 642 - Numerical Estimation
Efficient and reliable numerical algorithms in estimation and their applications...
Prerequisites: MATH 323, MATH 324 and COMP 350
Instructor: Xiao-Wen Chang
```

Figure 3.1 The *CourDesc* text

The text gives the course name, introduction, prerequisites and instructors for each course. The lines containing different information of the course start with different key words, e.g. *COMP*, *Prerequisite* and *Instructor* etc, which helps to predict their contents. And all information for one course is bounded within one paragraph, which helps to distinguish the different courses. The patterns of the text are obvious. We will try to do some text mining on the text and extract some useful information into a relation, which is ready for further computer process.

```
>domain name, course, description, prerequisite, instructor strg;
>CName <- text2attr("^COMP \d+ -.*?$", name, ".*?\n\n", course) CourDesc;
>CDesc <- text2attr("^[^(Prerequisite)(COMP)(Instructors)(\n)].+?$", description, ".*?\n\n",
course) CourDesc;
>CPrer <- text2attr("^Prerequisite.*?$", prerequisite, ".*?\n\n", course) CourDesc;
>CInst <- text2attr("^Instructors.*?$", instructor, ".*?\n\n", course) CourDesc;
>CourDescRel <- [name, description, prerequisite, instructor] in (CName ijoin CDesc ijoin CPrer ijoin
CInst);
>pr CourDescRel;
+----------------------+----------------------+----------------------+----------------------+
| name                 | description          | prerequisite         | instructor           |
+----------------------+----------------------+----------------------+----------------------+
| COMP 575 - Fundament | Study of a collectio | Prerequisite: COMP 3 | Instructors: Carl Tr |
| COMP 617 - Informati | Seminar course. A ma | Prerequisite: COMP 6 | Instructors: Timothy |
| COMP 642 - Numerical | Efficient and reliab | Prerequisites: MATH  | Instructors: Xiao-We |
+----------------------+----------------------+----------------------+----------------------+
relation CourDescRel has 3 tuples
```

In this example, different kinds of course information are extracted into intermediate relations, e.g. *CName*, *CDesc*, *CPrer* and *CInst*, by text-to-attribute operations. Each operation accepts two groups of parameters. The first group matches the desired information and gives it the attribute name, and the second group matches the whole paragraph that identifies the course to which it belongs. The ijoin operation following the extraction commands assemble different information of the same course into the same tuple. The result of the mining example is shown in the relation *CourDescRel*.

The *text2attr* operator can also be applied on texts in domain algebra and create a nested relation for each tuple in the actualized relation. It is similar to its application on attributes and further examples are not to be given in this section.

3.5 Markup-To-Nest Operator

3.5.1 Definition of Null-type Domain

Before considering the markup-to-nest operator, we would like to introduce the new feature of defining null type domain in JRelix, in other words, defining a domain without

specifying its type. It will be needed it in section 3.5.2 when we run the recursion with the *mu2nest* operator.

Syntax: domain <*domain_list*>;

With this command, the system is told to define domains in *domain_list* with an *undefined* type and to leave the domain types to be specified in further operations.

Example:

```
>domain Spouse, Married, Name, DoB;
>sd;
------------------------------ Domain Entry -------------------------------
Name        Type            NumRef     IsState   Dom_List
---------------------------------------------------------------------------
Name        undefined       0          false
Spouse      undefined       0          false
Married     undefined       0          false
DoB         undefined       0          false
---------------------------------------------------------------------------
```

In the example we define domain *Spouse*, *Married*, *Name* and *DoB* without giving their types. In the following section we can see that the *mu2nest* operator accepts the null type domains and assigns their domain types in the run-time. The null type domains are shown with type *undefined* in the domain list before the specification.

3.5.2 Markup-To-Nest

The operators introduced in the previous sections are designed for operations on plain text. Other than plain text, another kind of embedded text data is important, namely the marked-up text. Marking up the text with *xML* tags specified more elaborate information than the plain text. For example, by marking up the text *person* in section 3.1.1, we have a semi-structured text *personmu*:

```
<Person>
  <Name>Ted</Name> married
  <Family><Spouse>Alice</Spouse> in <Married>1932</Married>. Their children,
    <Children><Name>Mary</Name> (<DoB>1934</DoB>) married
      <Family><Spouse>Alex</Spouse> in <Married>1954</Married>
        (<Children><Name>Joe</Name> was born to Mary and Alex in <DoB>1956</DoB>
        </Children>)
      </Family>
      and <Name>James</Name> (<DoB>1935</DoB>) married
      <Family><Spouse>Jane</Spouse> in <Married>1960</Married>
        (<Children>James and Jane had <Name>Tom</Name> in
         <DoB>1961</DoB> and <Name>Sue</Name> in <DoB>1962</DoB>
        </Children>).
      </Family>
    </Children>
  </Family>
</Person>
```

The markup-to-nest operator, *mu2nest*, is implemented to handle the marked-up text. It works on a marked-up text or a marked-up string attribute and produces a nested relation.

Syntax: mu2nest(<*exclude_pattern*>, (<*attr*>)*) <*text*>

Parameters:

text – The source text.

exclude_pattern – The pattern indicating what is to be excluded from the text. The excluded strings will not appear in the *content* attribute. The usual things a programmer wishes to exclude in the text are the mark-up tags.

attr – There are 3 possible types of parameters that *mu2nest* takes in the following order: *content*, *start* and *length*. They are attributes of string-type, integer-type and integer-type predefined by the programmer. Similar to the *JRelix* computation, *mu2nest* adopts the positional convention and recognizes the parameters by the order of their appearance. That is to say, if the programmer wants to omit some of the parameters, the commas should be left in, expect that the omitted parameter is the last parameter.

1. *content* – The content of the element after excluding undesired strings matched by the pattern. The tags will be included in the *content* if the programmer gives a null pattern ("").
2. *start* – The absolute starting position of the element in the text, with tags excluded.

3. *length* – The content length of the element, with tags excluded.

The following is a simple example on the text *personmu*.

Example:

```
>domain content strg;
>domain start, length intg;
>personnest <- mu2nest("<.*?>", content, start, length) personmu;
>pr personnest;
+----------------------+---------+---------+------------------+------------------+
| content              | start   | length  | Name             | Family           |
+----------------------+---------+---------+------------------+------------------+
| Ted married Alice in | 0       | 203     | 20               | 21               |
+----------------------+---------+---------+------------------+------------------+
relation personnest has 1 tuple
>pr .Name;
+----------------------+----------------------+------------+------------+
| .id                  | content              | start      | length     |
+----------------------+----------------------+------------+------------+
| 20                   | Ted                  | 0          | 3          |
+----------------------+----------------------+------------+------------+
relation .Name has 1 tuple
>pr .Family;
+----------------------+-------------------------------------------------------------+
| .id                  | content                                                     |
+----------------------+-------------------------------------------------------------+
| 21                   | <Family .abspos=12><Spouse>Alice</Spouse> in <Married>1932 |
+----------------------+-------------------------------------------------------------+
relation .Family has 1 tuple
```

The value of *content* in relation *personnest* is the whole text with tags excluded. We can see that *mu2nest* extracts the specified attributes at the top level and generates nested relations for the child-elements. To the leaf child-element *Name*, *mu2nest* applies the similar rule to that of the top-level element, and retrieves the attributes *content*, *start* and *length* for the nested relation *Name*. Since *Family* recursively contains marked-up text, *mu2nest* does not descend into it. *mu2nest* only extracts its content as a marked-up string and creates a nested relation for it.

Note: In the case when a nested relation already exists with the same name before *mu2nest* creates one, *mu2nest* checks for its structure. If the old relation is compatible with the one *mu2nest* is going to create, i.e. they have identical attributes in the same order, new tuples will be appended to the old one. Otherwise, the operation will abort.

According to the positional convention used on the parameters, the invocations

mu2nest("<.*?>", content, , length) personmu

and

mu2nest("<.*?>", content) personmu

are both acceptable. The former generates *content* and *length*, the latter generates *content* as extracting attributes.

The *mu2nest* operator can also be applied in the domain algebra and extract structure information from string-type attributes in the relations. To go deeper in the text *personmu*, we now apply *mu2nest* to *Family* in *personnest*.

Example:

```
>let F be mu2nest("<.*?>", content, start, length) Family;
>personnest' <- [content, start, length, Name, F] in personnest;
>pr personnest';
+--------------------+------------+------------+----------------+----------------+
| content            | start      | length     | Name           | F              |
+--------------------+------------+------------+----------------+----------------+
| Ted married Alice in | 0          | 203        | 35             | 40             |
+--------------------+------------+------------+----------------+----------------+
relation personnest' has 1 tuple
>pr .F;
+--------+--------------------+---------+---------+----------+----------+----------+
| .id    | content            | start   | length  | Spouse   | Married  | Children |
+--------+--------------------+---------+---------+----------+----------+----------+
| 40     | Alice in 1932. Their | 12      | 191     | 37       | 38       | 39       |
+--------+--------------------+---------+---------+----------+----------+----------+
relation .F has 1 tuple
```

```
>pr .Spouse;
+--------------------+--------------------+------------+------------+
| .id                | content            | start      | length     |
+--------------------+--------------------+------------+------------+
| 37                 | Alice              | 12         | 5          |
+--------------------+--------------------+------------+------------+
relation .Spouse has 1 tuple
>pr .Married;
+--------------------+--------------------+------------+------------+
| .id                | content            | start      | length     |
+--------------------+--------------------+------------+------------+
| 38                 | 1932               | 21         | 4          |
+--------------------+--------------------+------------+------------+
relation .Married has 1 tuple
>pr .Children;
+--------------------+----------------------+
| .id                | content              |
+--------------------+----------------------+
| 39                 | <Children .abspos=43>< |
+--------------------+----------------------+
relation .Children has 1 tuple
```

With the help of domain algebra, *mu2nest* analyses the marked-up text in *Family* in a similar manner to what it did on *personmu*. Again it extracts the information at the top level, returns a single-tuple relation *.F* and creates nested relations for the elements one level down.

We noticed that *mu2nest* leaves the element *Children* with structures unanalyzed. To go further and expand all levels in the text *personmu*, we need recursion in domain algebra. Since this example nests *Children* inside *Family*, and *Family* inside *Children*, we define two virtual attributes for *Family* and *Children* mutually.

Example:

Step 1. Define null domains.

```
>domain Family, Children, Family', Children', Spouse, Married, Name, DoB;
```

In the previous examples, the child-element attributes, such as *Name*, *Spouse* and

Married, are generated by *mu2nest* automatically in the run-time. But in this example, we are going to define virtual domain *Family'* and *Children'* on these undefined domains. Thus we need to define them as null domains first to make JRelix accept the definition of *Family'* and *Children'*. Their domain types will be given in the run-time.

Step 2. Define recursive domains mutually.

```
>domain c strg;
>domain s, l intg;
>let Family' be [c, s, l, Spouse, Married, Children'] in mu2nest("<.*?>", c, s, l) Family;
>let Children' be [c, s, l, Name, DoB, Family'] in mu2nest("<.*?>", c, s, l) Children;
warning: domain 'Children'' is being referenced.
```

We can see that *Family'* is defined on *Children'*, and *Children'* is defined on *Family'*. They are also defined on the null domains we declared above.

Step 3. Run the recursion.

```
>personrecu <- [c, s, l, Name, Family'] in mu2nest("<.*?>", c, s, l) personmu;
>pr personrecu;
+---------------------+-------------+-------------+--------------+--------------+
| c                   | s           | l           | Name         | Family'      |
+---------------------+-------------+-------------+--------------+--------------+
| Ted married Alice in | 0           | 203         | 41           | 63           |
+---------------------+-------------+-------------+--------------+--------------+
relation personrecu has 1 tuple
>pr .Family';
+------+---------------------+-------+-------+------------+------------+------------+
| .id  | c                   | s     | l     | Spouse     | Married    | Children'  |
+------+---------------------+-------+-------+------------+------------+------------+
| 61   | Alex in 1954 (Joe wa | 63    | 52    | 49         | 50         | 57         |
| 61   | Jane in 1960 (James  | 141   | 62    | 52         | 53         | 60         |
| 63   | Alice in 1932. Their | 12    | 191   | 43         | 44         | 62         |
+------+---------------------+-------+-------+------------+------------+------------+
relation .Family' has 3 tuples
>pr .Spouse;
+---------------------+---------------------+-------------+-------------+
| .id                 | c                   | s           | l           |
+---------------------+---------------------+-------------+-------------+
| 43                  | Alice               | 12          | 5           |
```

```
| 49                  | Alex                | 63          | 4           |
| 52                  | Jane                | 141         | 4           |
+---------------------+---------------------+-------------+-------------+
relation .Spouse has 3 tuples
>pr .Married;
+---------------------+---------------------+-------------+-------------+
| .id                 | c                   | s           | l           |
+---------------------+---------------------+-------------+-------------+
| 44                  | 1932                | 21          | 4           |
| 50                  | 1954                | 71          | 4           |
| 53                  | 1960                | 149         | 4           |
+---------------------+---------------------+-------------+-------------+
relation .Married has 3 tuples
>pr .Children';
+------+---------------------+-------+-------+-----------+-----------+------------+
| .id  | c                   | s     | l     | Name      | DoB       | Family'    |
+------+---------------------+-------+-------+-----------+-----------+------------+
| 57   | Joe was born to Mary | 77    | 37    | 55        | 56        | dc         |
| 60   | James and Jane had T | 155   | 46    | 58        | 59        | dc         |
| 62   | Mary (1934) married  | 43    | 160   | 46        | 47        | 61         |
+------+---------------------+-------+-------+-----------+-----------+------------+
relation .Children' has 3 tuples
>pr .Name;
+---------------------+---------------------+-------------+-------------+
| .id                 | c                   | s           | l           |
+---------------------+---------------------+-------------+-------------+
| 41                  | Ted                 | 0           | 3           |
| 46                  | Mary                | 43          | 4           |
| 46                  | James               | 120         | 5           |
| 55                  | Joe                 | 77          | 3           |
| 58                  | Tom                 | 174         | 3           |
| 58                  | Sue                 | 190         | 3           |
+---------------------+---------------------+-------------+-------------+
relation .Name has 6 tuples
>pr .DoB;
+---------------------+---------------------+-------------+-------------+
| .id                 | c                   | s           | l           |
+---------------------+---------------------+-------------+-------------+
| 47                  | 1934                | 49          | 4           |
| 47                  | 1935                | 127         | 4           |
| 56                  | 1956                | 110         | 4           |
| 59                  | 1961                | 181         | 4           |
| 59                  | 1962                | 197         | 4           |
+---------------------+---------------------+-------------+-------------+
relation .DoB has 5 tuples
```

We noted that the recursion invoked by the mutual definitions of Family' and Children' extracts the structure of *Person* at all levels. The recursion comes to an end when it encounters a null value in *Family'*, and presents the result as recursive nested relations.

The text structure of *Person* explored by *mu2nest* is illustrated below. The numbers in the brackets are surrogates of nested relation, which would help in finding out the original data in the tables above. We also noticed that when *mu2nest* encounters two or more elements with the same name at the same level, it appends them as different tuples in the nested relation with the same surrogate.

```
Person
(Name   Family                                                          )
        (Spouse Married Children                                        )
                        (Name   DoB     Family                          )
                                (Spouse Married Children                )
                                                (Name   DoB     Family  )
 Ted    (63)
        Alice   1932    (62)
                        Mary    1934    (61)
                                        Alex    1954    (57)
                                                        Joe     1956    dc
                        James   1935    (61)
                                        Jane    1960    (60)
                                                        Tom     1961    dc
                                                        Sue     1962    dc
```

Chapter 4

Implementation of Text Operations

After presenting the users' manual of text operations, we go deeper into the implementation of these new features in the JRelix system. In this chapter, the implementation details are given. Sections 4.1 and 4.2 introduce the general development environment and the overview of the existing JRelix system architecture. In section 4.3, the strategy of text storage and the implementation of basic text operations are interpreted. In sections 4.4 to 4.6, the design and implementation of three groups of text operators, namely binary grep operator, text-to-attribute operator and markup-to-text operator, are presented in detail for both relation algebra and domain algebra.

4.1 Development Environment

The JRelix system is developed in Java. The new version of JRelix is implemented on JDK 1.4.0 or higher and runs on the platform of UNIX, Linux, MS Windows and any other operating system that supports Java.

The parser of the JRelix programming language is generated by two parser tools: JJTree and JavaCC. JavaCC, Java Compiler Compiler, is written in Java and is capable of

generating a parser from a high-level grammar specification in a text file and converting it into a Java program which recognizes matches in the grammar. As a complement to JavaCC, JJTree is a preprocessor of JavaCC which inserts parse tree building actions at various places in the JavaCC source. The output of JJTree is run through JavaCC to create the parser. JDK, JavaCC and JJTree together constitute the platform for the JRelix implementation.

4.2 JRelix Implementation Overview

The JRelix system contains four main components. They are the Parser, the Interpreter, the Execution Engine and the Data Storage. The parser and the interpreter work as the front-end processor and the interface between the user and the execution engine. The execution engine is invoked by the interpreter and fulfills the task passed from it. Data storage manages the data in the system as disk files or as run-time data in the RAM. The relations between the components are shown in Figure 4.1.

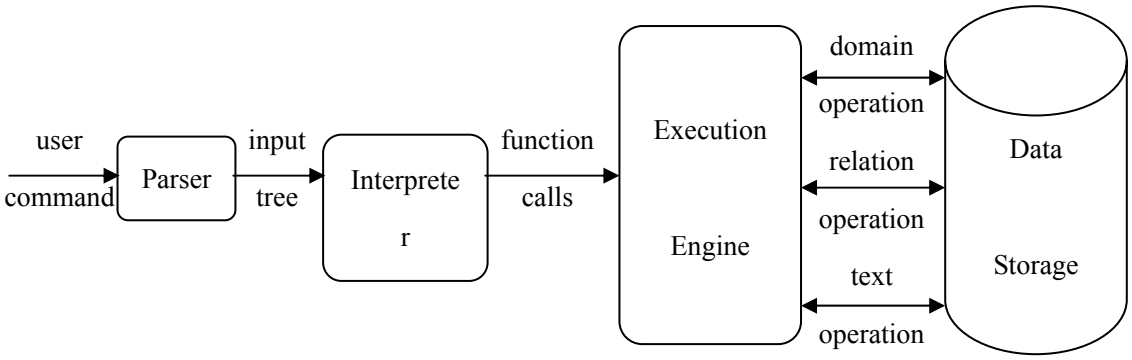

Figure 4.1 The JRelix system

The Parser reads the command-line input from the console, analyzes the command syntax and builds up an input tree for the command which can be interpreted by the interpreter. The parser is built with the help of JJTree and JavaCC [SDV96 & Kang03].

The **Interpreter** runs in an infinite loop. It calls the parser, receives the input tree,

traverses the tree, decomposes the user input into function calls and executes the calls by invoking the execution engine. It is implemented in Java and represented by the class *interpreter*.

The **Execution Engine** consists of function modules, i.e. Relation Processor [Hao98], Virtual Domain Actualizer[Yuan98] and Event Processor[He97], which can be invoked by the interpreter and fulfill the task of computations or data operations.

The **Data Storage** is the data which permanently resides on the disk and the run-time data in the RAM. There are two types of data in the JRelix system, the user-defined data which represents the elements that users defined in the system, and the system information data which describes the current status of the system. They are managed by both the interpreter and the execution engine.

4.3 Basic Text Operations

As a new element introduced into the JRelix system, text has its own set of operation commands. For the convenience of the programmer, most of the text commands are integrated into the commands we are already using to handle relations. This section talks in detail about the implementation of basic text operation commands. We will start with the text storage in the system, and then we will go to the basic operations such as definition and deletion, and at the end we will finish the section by introducing the integration of the text commands and the relation commands.

4.3.1 Text Storage

Similar to the relations in JRelix, the text elements have their own storage in the system. The storage of text consists of two parts: the text table and the text files.

The **Text table**, basically a hash table, is one type of system information data. It registers

the texts existing in the system with their names, data lengths and part of their contents. Every entry in the text table has the following structure.

Item	Type	Description
Name	String	Text name
Length	Integer	Text file length
Abstr	String	First 40 characters of text content

The text table resides in the RAM in run-time. It is written to the *.text* file on the disk when JRelix shuts down, and read from the same file when JRelix boots up. The text table is represented by the *TextTable* class in the current implementation.

The **Text file** is an ASCII file stored permanently on the disk with the suffix *.text* in the filename. It represents the text in the system. Unlike the relation files which are loaded into the RAM sometime in run-time, the text files are only opened and read during the operation and they never reside in the RAM. The text file is represented by the *Text* class in the implementation.

4.3.2 Text Definition and Initialization

- Define an Empty Text

The user is allowed to define a text without initialization. In this case, JRelix defines an empty text in the system by inserting an entry into the text table without creating a text file. The text length is set to 0 and the text abstract is set to null in the text table. An empty text is acceptable as input to the text operators, but no operation is actually executed on the text.

- Initialize with an ASCII File

JRelix adopts a similar syntax in text initialization as in relation initialization. It opens

the input ASCII file by the absolute pathname, and copies the content of the file into a **.text* file under the running directory. The format of the text files on the Windows platform is converted to the one on the UNIX platform during the initialization by replacing "\r\n" with "\n" in the input file. Meanwhile, an entry is appended to the text table in the memory with the name, file length and abstract of the new text.

- Initialize with a Text or a Relation

The process of initialization with another text is similar to that with an ASCII file. The difference is that the system reads the input from the running directory instead of the given file path. No format conversion is needed in the copy process.

Text initialization with relations is implemented to make it possible to apply the text operations on the relation data. The data in the relation is converted into a string format and written into the text file during the text initialization. Tuples are separated by line-feeds and the fields in the same tuple are separated by tab characters in the text file. The following example shows the content of text *BoMtext* which comes from a relation *BoM*.

```
>pr BoM;
+----------------------+-------------+----------------------+
| assembly             | qty         | subassembly          |
+----------------------+-------------+----------------------+
| cover                | 1           | plate                |
| cover                | 2           | screw                |
| fixture              | 2           | plug                 |
| fixture              | 2           | screw                |
| plug                 | 1           | mould                |
| plug                 | 2           | connector            |
| wallplug             | 1           | cover                |
| wallplug             | 1           | fixture              |
+----------------------+-------------+----------------------+
relation BoM has 8 tuples
>text BoMtext <- BoM;
```

```
>pr BoMtext;
------------------------------------------------------------------------------
                                Text BoMtext
------------------------------------------------------------------------------
cover   1       plate
cover   2       screw
fixture 2       plug
fixture 2       screw
plug    1       mould
plug    2       connector
wallplug        1       cover
wallplug        1       fixture
------------------------------------------------------------------------------
Text BoMtext has 132 characters
```

4.3.3 Integration of Text and Relation

To increase the usability of this new feature in JRelix, and to make it possible to handle text and relations in the same way, some basic relational operators are enhanced to handle the text as well as the relations. The following table gives the relational operators which are able to work on text and their equivalent textual operators.

Relational Operator on Text	Equivalent Textual Operator	Usage
pr *text_name*;	pt *text_name*;	print out text
sr *text_name*;	st *text_name*;	show text information
dr *text_name*;	dt *text_name*;	delete text

Texts are not allowed to share or reuse relation names, and vice-versa.

4.3.4 Text Listing, Printing and Deleting

- Listing

The show-text command lists all the entries in the text table or one specified entry if the text name is given as the parameter. To make the output look tidy, the tab characters in the content are already converted into space characters during the text initialization. A

typical text listing is shown below.

```
>sr;
---------------------------------- Text Entry ----------------------------------
Name                Length              Content
--------------------------------------------------------------------------------
person              203                 Ted married Alice in 1932. Their childre
newperson           203                 Ted married Alice in 1932. Their childre
BoMtext             132                 cover 1 plate cover 2 screw fixture 2 pl
--------------------------------------------------------------------------------
```

- Printing

The print-text command prints the content of the text with its name and length. No conversion is done on the content before printing.

```
>pr person;
--------------------------------------------------------------------------------
                                  Text person
--------------------------------------------------------------------------------
Ted married Alice in 1932. Their children, Mary (1934) married Alex in 1954 (Joe
was born to Mary and Alex in 1956) and James (1935) married Jane in 1960 (James
and Jane had Tom in 1961 and Sue in 1962).
--------------------------------------------------------------------------------
Text person has 203 characters
```

- Deleting

The delete-text command deletes the text file and removes the corresponding entry in the text table if given text exists. Otherwise, it does nothing except to return with a warning.

4.4 Binary Grep Operators

The binary grep family consists of six members, namely *igrep*, *ugrep*, *dgrep*, *sgrep* (*diff*), *lgrep* and *rgrep*. They differ from each other in the relational operation applied on the grepped matches in the texts. The strategies in their implementation are almost the same.

Generally speaking, the binary operators take the following sequential steps in the implementation.

- Generate attributes from the syntax tree passed by the parser.
- Apply the match pattern to two texts respectively, and create two intermediate relations containing the matches from the texts.
- Apply the corresponding binary-join operation on the intermediate relations.
- Set the values of unmatched fields to null, and arrange attributes to user-specified order.

We are now step by step going to explain the details of the implementation with a concrete example. Assume that we have two texts *Jshort* and *Sshort*.

```
>pr Jshort;
--------------------------------------------------------------------------------
                                  Text Jshort
--------------------------------------------------------------------------------
Joe invited Sue to a movie which he enjoyed a lot.
--------------------------------------------------------------------------------
Text Jshort has 50 characters
>pr Sshort;
--------------------------------------------------------------------------------
                                  Text Sshort
--------------------------------------------------------------------------------
Joe invited Sue to a movie which she only enjoyed a little.
--------------------------------------------------------------------------------
Text Sshort has 59 characters
```

We want to find the difference between these two texts. Here the tool we use is *sgrep*, also called *diff*.

```
>domain val1, val2 strg;
>domain pos1, pos2 intg;
>JSdiff <- Jshort diff("\w+", pos1, val1, val2, pos2) Sshort;
>pr JSdiff;
+-------------+---------------------+---------------------+-------------+
| pos1        | val1                | val2                | pos2        |
+-------------+---------------------+---------------------+-------------+
| dc          | dc                  | little              | 52          |
| dc          | dc                  | only                | 37          |
| dc          | dc                  | she                 | 33          |
| 33          | he                  | dc                  | dc          |
| 46          | lot                 | dc                  | dc          |
+-------------+---------------------+---------------------+-------------+
relation JSdiff has 5 tuples
```

The Symmetric difference grep extracts the different words in two texts and gives their occurrence positions. Four steps have been taken behind the screen.

Step 1. Generate attributes from given parameters.
Before running the pattern matching, we first validate the parameters provided by the user and pass the attributes to the pattern matcher. Basically, we need three attributes for each text. They are:

The **Value attribute**, which will contain the values of matches returned by the text matcher. In the example, predefined domain *val1* is accepted as the value attribute for the left-hand text and *val2* for the right-hand text. If any of the given domains is undefined, *bigrep* will abort and print out a domain-not-declared message.

The **Position attribute**, which will give the positions of the matches in the text. In the example, the integer type domains *pos1* and *pos2* are parsed as position attributes for the left-hand and right-hand text respectively. If any of the position attributes are omitted by the user, *bigrep* will pass a null value to the matcher.

The **Sequence attribute**, which will keep the sequence information for the matches. It is a system-information attribute which is automatically created by *bigrep* and passed to the matcher. The Sequence attribute is not contained in the result of binary grep operations.

The first step is executed in the interpreter. After the necessary attributes are prepared, they are passed with the matching pattern to the matchers in two *Text* instances.

Step 2. Grep in texts.

Each Text instance validates the attributes received from the interpreter or creates a default attribute if any of them is null. Afterwards, it applies the pattern to the text and puts the matches into a relation. The text matcher is built with the help of the Java Pattern

class and Matcher class.

In our example, the matcher generates the following intermediate relations for the matches in Jtext and Stext, and returns them to the interpreter. We can see that the domains given by the user become attributes which contain value and position information for two texts. The sequence information which is not useful to *bigrep* will be ignored in the interpreter.

Matches in Jtext:

val1	pos1	_seql
Joe	0	0
invited	4	1
Sue	12	2
to	16	3
a	19	4
movie	21	5
which	27	6
he	33	7
enjoyed	36	8
a	44	9
lot	46	10

Matches in Stext:

val2	pos2	_seqr
Joe	0	0
invited	4	1
Sue	12	2
to	16	3
a	19	4
movie	21	5
which	27	6
she	33	7
only	37	8
enjoyed	42	9
a	50	10
little	52	11

Step 3. Apply binary join to intermediate relations.

The difference between the six binary operations exists in the relational operations applied on the intermediate relations. The intersection operation is applied in *igrep*, and the symmetric difference operation is applied in *sgrep*, etc.

The relational operation on the two relations is shown below. The values of *val1* and *val2* are always the same, but we want the value attribute of the unmatched side to be null. Furthermore, the attributes are not in the desired order, thus we have one more step to go.

val1	pos1	_seq1	val2	pos2	_seqr
he	33	7	he	dc	dc
little	dc	dc	little	52	11
lot	46	10	lot	dc	dc
only	dc	dc	only	37	8
she	dc	dc	she	33	7

Step 4. Clarifying and presenting.

The last step is to set the values of the unmatched tuples to null value ("*dc*") to make the result more readable. Finally the relation is projected to the user-specified attributes and presented as the output of the binary grep operation.

pos1	val1	val2	pos2
dc	dc	little	52
dc	dc	only	37
dc	dc	she	33
33	he	dc	dc
46	lot	dc	dc

4.5 Text-To-Attribute Operator

4.5.1 Range Join

To address the needs of retrieving the hierarchical structure from the text, the feature of range joins, namely high-range join and low-range join, is added in JRelix for the system programmers. That is to say, the function of range joins is implemented in the *Relation* class, but the JRelix parse is not amended to accept the range-join operation from the JRelix programmer. If the JRelix enhancement is to subsequently open range joins to the JRelix programmer, only a thorough testing is needed on the operation before it becomes functional. Figure 4.2 illustrates the implementation strategy of range joins.

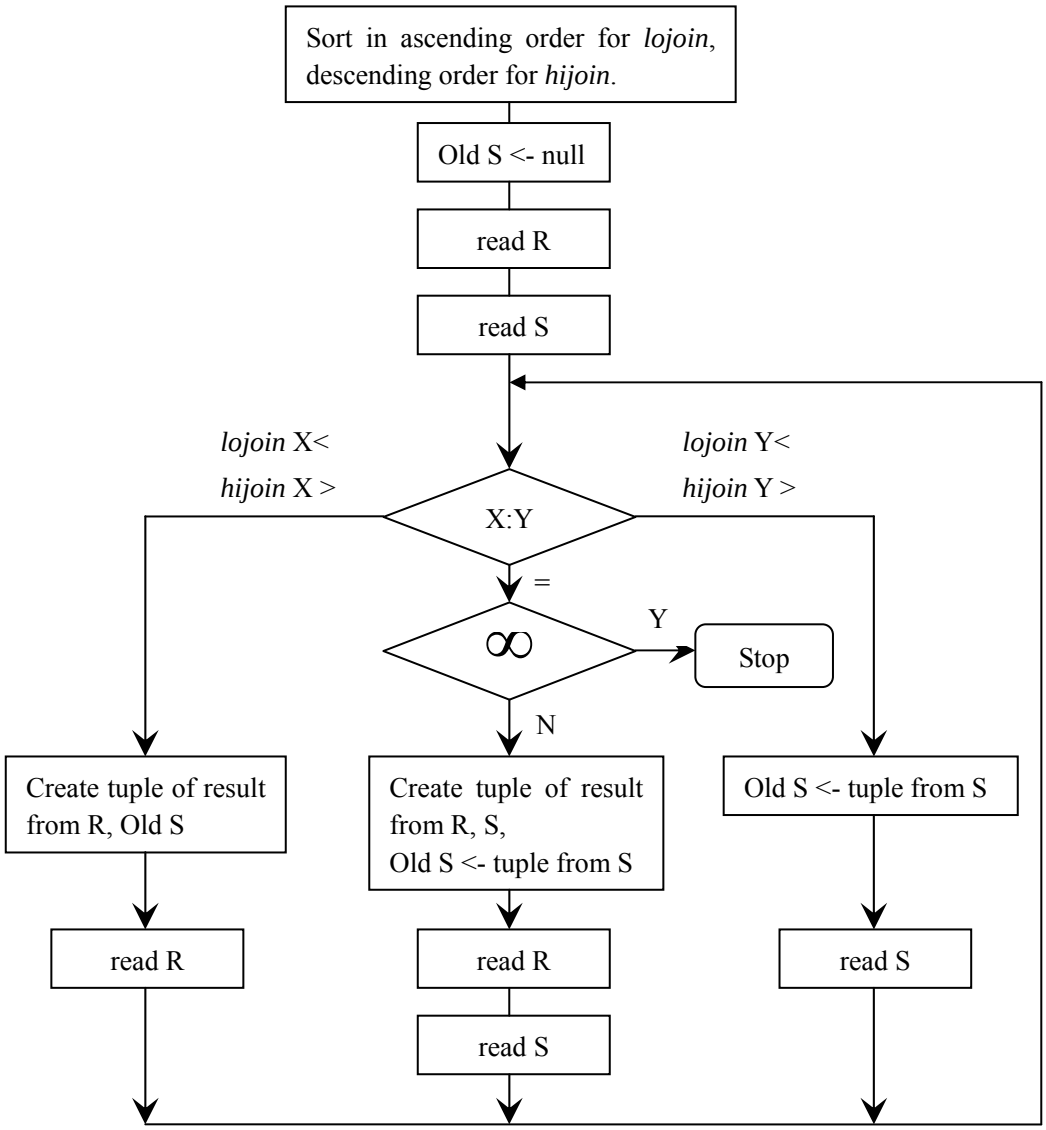

Figure 4.2 Implementing Range-join R[X join Y]S

4.5.1 text2attr Operator

The *text2attr* operator is implemented to extract different levels of elements in the text and join them together according to their natural structure. It takes parameters in groups, each of which contains a compulsory string as the matching pattern, an optional integer type domain as the match sequence attribute and an optional string type domain as the match value attribute. In this section, we will discuss how it works with the following

example. We apply *text2attr* to text *JmetS* and retrieve its textual structure.

```
>pr JmetS;
-------------------------------------------------------------------------------
                                  Text JmetS
-------------------------------------------------------------------------------
After work, Joe met Sue.
-------------------------------------------------------------------------------
Text JmetS has 24 characters
>domain charseq, wordseq intg;
>domain char, word strg;
>charword <- text2attr(".", charseq, char, "\w+|\s+|\p{Punct}", wordseq, word) JmetS;
>pr charword;
```

charseq	char	wordseq	word
0	A	0	After
1	f	0	After
2	t	0	After
3	e	0	After
4	r	0	After
5		1	
6	w	2	work
7	o	2	work
8	r	2	work
9	k	2	work
10	,	3	,
11		4	
12	J	5	Joe
13	o	5	Joe
14	e	5	Joe
15		6	
16	m	7	met
17	e	7	met
18	t	7	met
19		8	
20	S	9	Sue
21	u	9	Sue
22	e	9	Sue
23	.	10	.

```
relation charword has 24 tuples
```

The text is parsed into characters and words, and they are presented in a relation structured according to their natural hierarchy. In the interpreter, three steps are executed to produce the relation.

Step 1. Generate attributes.

The first step is to generate the attributes for the pattern matcher from the parameters given by the user. As introduced in section 4.4, there are three types of attributes needed by the matcher:

The **Value attribute** – comes from the string type domain provided by the user. If no string type domain is found in the parameters, a null value will be passed to the matcher.

The **Position attribute** – is automatically created by the interpreter for the reason that the user is not asked to provide this parameter. It will be left out when we present our final result.

The **Sequence attribute** – is an important attribute to the text2attr operator. It comes from the integer type domain given in the parameters. When it is not provided, A null value will be passed to the matcher.

The interpreter parses the parameters in groups, each of which starts with a pattern parameter and ends before the next group or the end of the parameters. Each group stands for one level of elements which will be grepped from the text and hierarchized into the relation. In the next step, the interpreter will grep different levels of elements in the text with the help of the parameter groups, and arrange the returned relations into a temporary vector.

Step 2. Grep in the text.

The pattern matcher takes a group of parameters and greps one type of elements defined by the pattern. It retrieves the value, position and sequence of the matches, puts them into corresponding attributes in the intermediate relation and returns it to the interpreter.

We are grepping two types of elements in our example, thus two intermediate relations are generated in this step:

Relation for *char* elements:

char	_temp_X9X_2	charseq
A	0	0
f	1	1
t	2	2
e	3	3
r	4	4
	5	5
w	6	6
o	7	7
r	8	8
k	9	9
,	10	10
	11	11
J	12	12
o	13	13
e	14	14
	15	15
m	16	16
e	17	17
t	18	18
	19	19
S	20	20
u	21	21
e	22	22
.	23	23

Relation for *word* elements:

word	_temp_X9X_3	wordseq
After	0	0
	5	1
work	6	2
,	10	3
	11	4
Joe	12	5
	15	6
met	16	7
	19	8
Sue	20	9
.	23	10

In the intermediate relations, value and sequence information resides in the user-defined

attributes, and position information resides in a temporary attribute. We are now going to take a further step to build the hierarchical structure.

Step 3. Low-range join and projection in relation.

To retrieve the hierarchical text structure, different levels of elements are related by applying a low-range join operation to the intermediate relations on the temporary position attributes. It produces a universal relation containing position, value and sequence information as follows.

_temp_X9X_2	char	charseq	_temp_X9X_3	word	wordseq
0	A	0	0	After	0
1	f	1	0	After	0
2	t	2	0	After	0
3	e	3	0	After	0
4	r	4	0	After	0
5		5	5		1
6	w	6	6	work	2
7	o	7	6	work	2
8	r	8	6	work	2
9	k	9	6	work	2
10	,	10	10	,	3
11		11	11		4
12	J	12	12	Joe	5
13	o	13	12	Joe	5
14	e	14	12	Joe	5
15		15	15		6
16	m	16	16	met	7
17	e	17	16	met	7
18	t	18	16	met	7
19		19	19		8
20	S	20	20	Sue	9
21	u	21	20	Sue	9
22	e	22	20	Sue	9
23	.	23	23	.	10

The Starting position of the elements is not desired in the result. We project the relation to the attributes provided by the parameter groups and present it as the final result.

charseq	char	wordseq	word
0	A	0	After

1	f	0	After
2	t	0	After
3	e	0	After
4	r	0	After
5		1	
6	w	2	work
7	o	2	work
8	r	2	work
9	k	2	work
10	,	3	,
11		4	
12	J	5	Joe
13	o	5	Joe
14	e	5	Joe
15		6	
16	m	7	met
17	e	7	met
18	t	7	met
19		8	
20	S	9	Sue
21	u	9	Sue
22	e	9	Sue
23	.	10	.

4.6 Markup-To-Nest Operator

The need to read and parse xML texts into JRelix requires an xML parser. It should be able to take different kinds of xML, i.e. XML and HTML, as the argument. To achieve the flexibility and to make our implementation independent of development environments, we decided to implement our specific xML parser.

4.6.1 xML Parser

A similar semi-structure data parser was implemented to create relations from specific semi-structure inputs with the help of the JRelix parser [Yu03]. Breaking away from the relation building parser, our new xML parser is more flexible and capable of handling different kinds of xML formats. It parses a markup text input and constructs a tree using the *SimpleNode* class in the JRelix system.

Basically, the parser reads through the text and builds one node for each element it encounters in the text. The structure of the xML input is preserved by the tree structure of the nodes. The following table shows how parser-tree nodes represent the elements in the markup text.

Element	*SimpleNode* instance
name	String *identifier*
content	Object *info*
attributes	Hashtable *attrs*
children	Vector *children*

In addition to parsing an xML into a tree, the xML parser calculates the starting position of each element, and records it by inserting an attribute called ".relativepos" into the attribute list of the node. The new attribute represents the relative position of the element to the beginning of the xML input with tags excluded. As we will see in the implementation of *mu2nest*, the relative position helps to compute the absolute position of the element in the whole text.

To demonstrate how the xML parser works on markup texts, we use the same text *personmu* as in section 3.5.2.

```
<Person>
  <Name>Ted</Name> married
  <Family><Spouse>Alice</Spouse> in <Married>1932</Married>. Their children,
    <Children><Name>Mary</Name> (<DoB>1934</DoB>) married
      <Family><Spouse>Alex</Spouse> in <Married>1954</Married>
        (<Children><Name>Joe</Name> was born to Mary and Alex in <DoB>1956</DoB>
        </Children>)
      </Family>
      and <Name>James</Name> (<DoB>1935</DoB>) married
      <Family><Spouse>Jane</Spouse> in <Married>1960</Married>
        (<Children>James and Jane had <Name>Tom</Name> in
         <DoB>1961</DoB> and <Name>Sue</Name> in <DoB>1962</DoB>
        </Children>).
      </Family>
    </Children>
  </Family>
</Person>
```

From the top level down, our parser produces a tree for the whole text.

```
Person - info=<Person><Name>Ted</Name> married <Family> ... </Family></Person>
       - attrs:  .relativepos=0
  Name - info=<Name>Ted</Name>
       - attrs:  .relativepos=0
  Family - info=<Family><Spouse>Alice</Spouse> in <Married>1932</Married> ... </Family>
         - attrs:  .relativepos=12
    Spouse - info=<Spouse>Alice</Spouse>
           - attrs:  .relativepos=12
    Married - info=<Married>1932</Married>
            - attrs:  .relativepos=21
    Children - info=<Children><Name>Mary</Name> (<DoB>1934</DoB>) married...</Children>
             - attrs:  .relativepos=43
      Name - info=<Name>Mary</Name>
           - attrs:  .relativepos=43
      DoB - info=<DoB>1934</DoB>
          - attrs:  .relativepos=49
      Family - info=<Family><Spouse>Alex</Spouse> in <Married>1954 ... </Family>
             - attrs:  .relativepos=63
        Spouse - info=<Spouse>Alex</Spouse>
               - attrs:  .relativepos=63
        Married - info=<Married>1954</Married>
                - attrs:  .relativepos=71
        Children - info=<Children><Name>Joe</Name> was born to Mary and ... </Children>
                 - attrs:  .relativepos=77
          Name - info=<Name>Joe</Name>
               - attrs:  .relativepos=77
          DoB - info=<DoB>1956</DoB>
              - attrs:  .relativepos=110
      Name - info=<Name>James</Name>
           - attrs:  .relativepos=120
      DoB - info=<DoB>1935</DoB>
          - attrs:  .relativepos=127
      Family - info=<Family><Spouse>Jane</Spouse> in <Married>1960 ... </Family>
             - attrs:  .relativepos=141
        Spouse - info=<Spouse>Jane</Spouse>
               - attrs:  .relativepos=141
        Married - info=<Married>1960</Married>
                - attrs:  .relativepos=149
        Children - info=<Children>James and Jane had <Name>Tom</Name> in ... </Children>
                 - attrs:  .relativepos=155
          Name - info=<Name>Tom</Name>
               - attrs:  .relativepos=174
          DoB - info=<DoB>1961</DoB>
              - attrs:  .relativepos=181
          Name - info=<Name>Sue</Name>
               - attrs:  .relativepos=190
          DoB - info=<DoB>1962</DoB>
              - attrs:  .relativepos=197
```

The xML parser builds up a tree of the exact structure of the markup text. If there is any attribute given in the tags of the markup text, it will also be extracted and processed in the same way as the system-generated attribute ".relativepos".

4.6.2 mu2nest Operator

The mu2nest operator is implemented to convert markup texts into nested relations. It extracts from the text the specified attributes at the top level and generates nested relations for the child elements. In section 4.6.1 we have seen how the xML parser parses a markup text *personmu* into an element tree. In this section, we will use the same example to explain the implementation of the *mu2nest* operator.

```
>domain content strg;
>domain start, length intg;
>personnest <- mu2nest("<.*?>", content, start, length) personmu;
>pr personnest;
+----------------------+---------+---------+------------------+------------------+
| content              | start   | length  | Name             | Family           |
+----------------------+---------+---------+------------------+------------------+
| Ted married Alice in | 0       | 203     | 20               | 21               |
+----------------------+---------+---------+------------------+------------------+
relation personnest has 1 tuple
>pr .Name;
+----------------------+----------------------+-------------+-------------+
| .id                  | content              | start       | length      |
+----------------------+----------------------+-------------+-------------+
| 20                   | Ted                  | 0           | 3           |
+----------------------+----------------------+-------------+-------------+
relation .Name has 1 tuple
>pr .Family;
+----------------------+-------------------------------------------------------------+
| .id                  | content                                                     |
+----------------------+-------------------------------------------------------------+
| 21                   | <Family .abspos=12><Spouse>Alice</Spouse> in <Married>1932 |
+----------------------+-------------------------------------------------------------+
relation .Family has 1 tuple
```

To create nested relations from the markup text, the *mu2nest* operator takes the following four steps:

Step 1. Generate domain lists for top level and one level down relations.

In this step, *mu2nest* interprets the domain parameters given by the user, and generates domain lists for three types of relations:

The **Parent relation**, or top-level relation, which comes from the root node in the element tree. In the first step, the domain list of the parent relation only contains user-provided domains, i.e. *content*, *start* and *length* in our example.

The **Leaf child relation**, which comes from a leaf node in the element tree. Its domain list contains exactly the same domains passed by the user, before an *.id* domain is added at the time it becomes a nested relation.

The **Non-leaf child relation**, which comes from a non-leaf node in the element tree. It contains a *content* domain, or an *.unanalyzed* domain if the *content* domain is omitted by the user. An *.id* domain will also be added into the relation when it becomes nested.

Step 2. Parse markup text into an element tree

With the help of our xMLParser, *mu2nest* reads the input text as a string and parses it into a *SimpleNode* tree with the element structure preserved. The generated tree of our example is showed in section 4.6.1. In the following steps, we will traverse the first two levels of the tree and convert them into nested relations.

Step 3. Build up the top level relation

The children of the root of the tree are traversed for the first time, and every child is inserted in the parent relation as a nested relation. In our example, child relations *Name* and *Family* are inserted into the top level relation in this step.

Now that the domain list of the top level relation is complete, we can create this relation and assign its values from the text. The assignment of the values is shown in the table with values in our example.

Attribute	Value	Example
content	Content of the top level element with tags excluded	Ted married Alice ... in 1962).
start	Absolute position of the top level element	0
length	Length of the top level element with tags excluded	203

The absolute position of an element is represented by its *.abspos* attribute. The attribute helps to calculate the absolute positions of itself and its descendents in the whole text. The *.abspos* attribute is added to the element when the element is a non-leaf node, and will be separated for further analysis, i.e. the *Family* element in our example.

So far we are unable to assign child relation surrogates in this step because the child relations are not generated. We will leave it for the next step.

Step 4. Build up the one level down relations

We traverse the child nodes for the second time, and assign values to leaf child relation and non-leaf child relation. The value assignment is shown as follows:

Attribute	Value	Example
Leaf child relation		Name
content	Element value without tags	Ted
start	Sum of the absolute position of the parent and the relative position of the child	0
length	Length of the element value	3
Non-leaf child relation		Family
content	Element value with tags, and with the *.abspos* attribute inserted to remember its absolute position	<Family .abspos=12><Spouse>Alice </Spouse> in ... </Family>

The relative position of the child is represented by the *.relativepos* attribute which is

added by the xML parser. The sum of the relative position of the child and the absolute position of the parent in the whole text gives the absolute position of the child element, i.e. the *start* attribute of the leaf child relation.

Rather than remembering the absolute position in the *start* attribute, a non-leaf child relation saves it as the value of the *.abspos* attribute for further analysis. That is the way *mu2nest* preserves the position information in domain algebra computation and recursions.

Chapter 5

Conclusions

The first part of this chapter provides a summary of the research and implementation that have been accomplished, and the new features of the extended JRelix database system. In the second part of the chapter, some suggestions are given for research and enhancement in the future.

5.1 Summary

We have proposed an Aldat extension within which text can be described and manipulated meaningfully as an equivalent of relation. The extension is implemented in the JRelix database, which is enhanced into a federated relational/text database. The extended operations are simple and intuitive. Yet supported by the inherent capability of semi-structured data manipulation in JRelix, they are able to perform simple text mining in plain text, discover schema in implicit structured text, convert between relations and texts, and execute search and extraction in structured/unstructured text.

The problem of text management is solved satisfactorily, and the following new features are introduced into the enhanced JRelix system.

- Text type is added. As a new user-defined element in JRelix, text is now able to be stored as a subfield in relations, or as an independent data type equivalent to relations. The former allows the existing relational and domain algebra to be applied on text as a string attribute, while the latter allows indices and views to be built for the text, and text retrieval/update to be executed without querying relations in a relational system.

- Simple text mining is provided. The implicit information and natural structure inside text can be extracted by the text-to-attribute operation which parses text fragments into attributes in a relation.

- Text structure retrieval is accomplished. Structured text marked up by SGML-style tags can be converted into a parse tree represented by nested relations. In a tree structure, subtexts of a node and its descendants are labelled on the node as pure text, which also leads to a simple solution to the subtree extraction problem.

- Semi-structured data is catered for. The structured text in JRelix is allowed to be semi-structured but well-formed text, with a fluid and finite schema. The problem of text schema discovery can be solved satisfactorily with the programming tools already implemented in JRelix [Merr03, Yu04].

- Full-text search is achieved, for both plain text and structured text. Regular expressions are adopted as the pattern matching standard in JRelix, which allows exact match and approximate match in a flexible way.

- Text comparison is implemented. The binary grep operations provide simple but powerful comparison between two texts, which include both independent text and text embedded in relations.

As an enhancement of a database programming language, the design and implementation of our text database model are modest. The enhancement does not provide many high-level operations on text such as automated grammatical inference tools and heterogeneous data sources reconciliation tools. Instead, it endows JRelix with basic but extensible text operations on the basis of which further programming can perform complex text manipulations. While the needs for text management are rapidly evolving, it gives a key to the database programmers to develop appropriate database systems for different applications.

5.2 Future Work

5.2.1 Text Update

We have not designed or implemented text update operations in the enhanced text/relational database. In a similar manner to the update operation in relational algebra, text update is an important tool for processing vast bodies of textual information. Unlike the data in conventional databases which is structured and divided into regions (e.g. attributes, tuples and relations), text has no intrinsic inner boundaries or regions, to which updates can be confined. This gives us the flexibility in defining text operations, and at the same time the challenge to make sure the change of the text is properly bounded and effective. We propose to extend the relational update operator to text and adopting regular expressions. Text update will be performed by replacing the subtexts matched by the update criteria with the new value given by the text expression.

The simplest update is to replace a certain string in the text with a new string. In the following example, we replace the name of *Ted* to *Tom* in the text *person*, which we have used in section 3.1.1.

```
>update person change "Ted" <- "Tom";
```

The updates using regular expression will change any strings in the text which satisfy the

update criteria. In the following example, we change all 20th century years to 19th century years in *person*, e.g. to change 1956 to 1856, etc.

```
>update person change "19/d/d"<- "18/d/d";
```

As another example of the usage of regular expression in text update, the following command deletes the markups embedded in the text,

```
>update personmu change "<.*?>" <- "";
```

A more complicated update is to compare two texts and change one of them according to the result of the comparison. In the following example, we find out and mark up the added string in *newtext* by comparing *newtext* with *oldtext* using the *dgrep* operation. The expected result of the update is shown in figure 5.3.

```
>update newtext change ".*" <- "<added>.*</added>" using newtext dgrep("\w+", pos1, val1, val2, pos2) oldtext;
```

Sue wondered why she had accepted his invitation.

Figure 5.1 The text *oldtext*

Sue wondered why she had not accepted his invitation.

Figure 5.2 The text *newtext* before update

Sue wondered why she had <added>not </added>accepted his invitation.

Figure 5.3 The text *newtext* after update

As we have seen above, the syntax of the text update can be similar to that of the relational update. The update region is specified by the regular expression in the *change* clause or by the tuples returned by the *using* clause, and the new value is given by the expression following the assignment arrow ("<-"). This also proves that text can be viewed as a special type of relation with implicit and flexible boundaries of attributes and tuples.

5.2.2 Auto-Markup

Following on from the last example in section 5.2.1, text update can mark up some valuable information, which gives convenience to the further processing of the text. In the current implementation, electronic markups have to be inserted into text manually. However, we are already able to discover the hidden information inside a text and extract it into a relation with an implemented text mining tool, as we can see in the fourth example in section 3.4. It leads to the idea that we can also mark up the discovered information and leave it in the text for further process, such as text schema discovery and text to relation conversion.

Thus an auto-markup tool is proposed for future enhancement, which looks for the hidden information in the text with the given pattern and marks it up automatically with given tags. For example, it is possible to mark up the text *CourDesc* in section 3.4 into a new text *CourDescMu* with the following command,

```
>text CourDescMu <- automu("COMP\s+\d+", "CourName",
                          "\w.*?\n", "CourTitl",
                          "\w.*?\n", "CourIntr",
                          "Prerequisite.*?\n", "CourPrer",
                          "Instructor.*?\n", "CourInst") CourDesc;
```

The proposed *automu* operator accepts parameters in pairs. The first parameter gives the pattern for matching, and the second parameter gives the name of the tag which will be used to mark up the matches. The operator parses through the text without backtracking and terminates at the end of the input string. It returns the following text as output.

```
<CourName>COMP 575</CourName> - <CourTitl>Fundamentals of Distributed
Algorithms</CourTitl>
<CourIntr>Study of a collection of algorithms that are basic to the world of concurrent
programming...</CourIntr>
<CourPrer>Prerequisite: COMP 310</CourPrer>
<CourInst>Instructor: Carl Tropper</CourInst>

<CourName>COMP 617</CourName> - <CourTitl>Information Systems</CourTitl>
<CourIntr>Seminar course. A major area of application of the techniques covered in
308-612 is discussed...</CourIntr>
<CourPrer>Prerequisite: COMP 612</CourPrer>
<CourInst>Instructor: Timothy Merrett</CourInst>

<CourName>COMP 642</CourName> - <CourTitl>Numerical Estimation</CourTitl>
<CourIntr>Efficient and reliable numerical algorithms in estimation and their
applications...</CourIntr>
<CourPrer>Prerequisites: MATH 323, MATH 324 and COMP 350</CourPrer>
<CourInst>Instructor: Xiao-Wen Chang</CourInst>
```

Figure 5.4 The result of the auto-markup operation

5.2.3 High-level Join

As a special type of relation, text should be able to be joined with relations. This operation is provided by the high-level join. It takes relations or texts as operands, ignoring the difference between relation and text, and gives a relation or a set of texts as output. Let's consider a secretary's work in sending a New Year's greeting to all the staff in the company by email. Suppose we have a marked-up text as a greeting email template, with the changeable fields such as *email*, *title* and *name* surrounded by tags. The text is shown in figure 5.2.

```
From: marie@company.com
To: <email></email>

Dear <title></title> <name></name>,

Wish you a Merry Christmas and Happy New Year!

Regards,

Marie (Secretary)
```

Figure 5.2 The *template* text

And we have a relation containing the email, title and name information for the letters. The relation is shown in figure 5.3.

```
Staff
(email                  title                  name            )
 andrew@company.com     Mr.                    Andrew Z.W Pang
 bob@company.com        Mr.                    Bob C. Smith
 sue@company.com        Ms.                    Sue K. White
 ...
```

Figure 5.3 The *staff* relation

The task now is to fill in the email template with the staff information and generate greeting messages. The work can be done by applying *ijoin* on the *template* text and the *staff* relation as follows,

```
>messages <- template ijoin staff;
```

The enhanced *ijoin* operation takes the marked up subfields in *template* as attributes and inner-joins the text and the relation on their common attributes. The result contains a generated message for each tuple in the *staff* relation as shown below,

```
messages
(email                title   name              template                              )
andrew@company.com    Mr.     Andrew Z.W Pang   From: marie@company.com
                                                To: <email>andrew@company.com</email>
                                                Dear <title>Mr.</title>
                                                     <name>Andrew Z.W Pang</name>,
                                                Wish you a Merry Christmas and Happy New
                                                Year!
                                                Regards,
                                                Marie (Secretary)
bob@company.com       Mr.     Bob C. Smith      From: marie@company.com
                                                To: <email>bob@company.com</email>
                                                Dear <title>Mr.</title>
                                                     <name>Bob C. Smith</name>,
                                                Wish you a Merry Christmas and Happy New
                                                Year!
                                                Regards,
                                                Marie (Secretary)
sue@company.com       Ms.     Sue K. White      From: marie@company.com
                                                To: <email>sue@company.com</email>
                                                Dear <title>Ms.</title>
                                                     <name>Sue K. White</name>,
                                                Wish you a Merry Christmas and Happy New
                                                Year!
                                                Regards,
                                                Marie (Secretary)
```

Figure 5.4 Generated *messages* relation

Complementing the text update and text-to-relation operations, hyper-join serves as a prototype of the relation-to-text operations and disregards the natural difference between relations and texts.

The implementation of text integration into JRelix is only the beginning of text research for database programming languages. The proposed future work indicates a vast area for text research and application in JRelix, which will exploit the outstanding data manipulation power of JRelix and endow JRelix with the characteristic of processing text, a huge body of evolving data on the Web.

Bibliography

[Abit97a] S. Abiteboul, "Querying Semistructured Data", *Proc. 6th Int. Conf. On Database Theory* (ICDT'97), Delphi, Greece (January 1997), Lecture Nodes in Computer Science 1186, Springer-Verlag, 1-18.

[Abit97b] S. Abiteboul, D. Quass, J. McHugh, J. Widon and J. Weiner, "The Lorel Query Language for Semistructure Data", *Journal of Digital Libraries 1*, 1 (April 1997) pp. 68-88.

[Abit99] S. Abiteboul, P. Buneman, and D. Suciu, "Data on the Web : From Relations to Semistructured Data and XML", Morgan Kaufmann, 1999.

[Agra99] R. Agrawal, R. Bayardo, and R. Srikant, "Athena: Mining-based interactive management of text databases", Research Report RJ 10153, IBM Almaden Research Center, San Jose, CA 95120, July 1999.

[Bake98] P. Baker, "Design and Implementation of Database Computations in Java", Master's thesis, School of Computer Science, McGill University, 1998.

[Blak94] G. E. Blake, M. P. Consens, P. Kilpelainen, P.-A. Larson, T. Snider and F. W. Tompa, "Text/relational Database Management Systems: Harmonizing SQL and SGML", Proc. Application of Databases (ADB94) Vadstena, Sweden (June 1994), Lecture Notes in Computer Science 819, Springer-Verlag, pp. 267-280.

[Brow98] L.J. Brown, M. Consens, I.J. Davis, C.R. Palmer, and F.W. Tompa. "A structured text ADT for object relational databases", *Theory and Practice of Object Systems* 4(4), 1998.

[Bune97] P. Buneman, "Semistructured Data", *Proc. of Symposium on Principles of Database Systems (PODS'97)*, p 117-121, 1997.

[Calv01] D. Calvanese, S. Castano, F. Guerra, et al, "Towards a Comprehensive Methodological Framework for Semantic Integration of Heterogeneous

Data Sources", In Eighth International Workshop on Knowledge Representation Meets Databases (KRDB), 2001.

[Chaf00] J. Chaffee, S. Gauch, "Personal ontologies for web navigation", *Proceedings of the ninth international conference on Information and knowledge management*, p.227-234, November 06-11, 2000, McLean, Virginia, United States

[Comp91] IEEE Computer, *Special Issue on Heterogeneous Distributed Database Systems*, 24(12), December, 1991.

[Coom87] J. H. Coombs, A. H. Renear, S. J. de Rose, "Markup systems and the future of scholarly text processing", *Comm. ACM 30*, 11 (November 1987), 933-947.

[Coop97] G. Cooper, "A simple constraint-based algorithm for efficiently mining observational databases for causal relationships", *Data Mining and Knowledge Discovery*, 2(1997).

[Fayy96] U. M. Fayyad, G. Piatetsky-Shapiro, and P. Smyth, "From data mining to knowledge discovery: An overview", in Advances in Knowledge Discovery and Data Mining, edited U. M Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, AAAI Press/MIT Press, pp. 1--34, 1996.

[Feld98] R. Feldman et al, "Text mining at the term level", *In Proc. of the 2nd European Symposium on Principles of Data Mining and Knowledge Discovery* (PKDD'98), Nantes, France, Sept 1998.

[Fuku98] K. Fukuda, T. Tsunoda, A. Tamura and T. Takagi, "Toward Information Extraction: Identifying Protein Names from Biological Papers", *Proceedings of the Pacific Symposium on Biocomputing* (PSB'98), page 705-716, Maui, Hawaii, January 1998.

[Gao99] X. Gao and L. Sterling, "Semi-Structured Data Extraction from Heterogeneous Sources", 2cd International Workshop on Innovative Internet Information Systems (IIIS'99), in conjunction with the European Conference on Information Systems (ECIS'99), Copenhagen, Denmark. 1999.

[Gonn87] G. H. Gonnet and F. W. Tompa, "Mind your grammar: a new approach to modeling text", *Very Large Data Bases* (VLDB), Vol. 13 (September 1987), 339-346.

[Hao98] B. Hao, "Implementation of the Nested Relational Algebra in Java", Master's thesis, School of Computer Science, McGill University, 1998. URL: http://www.cs.mcgill.ca/~tim/cv/theses/hao.ps.gz.

[He97] H. He, "Implementation of nested relations in a database programming language", Master's thesis, School of Computer Science, McGill University, 1997. URL: http://www.cs.mcgill.ca/~tim/cv/theses/he.ps.gz.

[Horn03] F. Horn, L. Lee, F.E. Cohen, "MuteXt: an automated method to extract mutation data from the literature", Pacific Symposium on Biocomputing 2003, January 3-7, 2003, Lihue, Hawaii (USA).

[ISO86] International Organization for Standardization, *Information processing – text and office systems – Standard Generalized Markup Language (SGML)*. ISO 8879: 1986

[Kazm86] Rick Kazman, "Structuring the text of the *Oxford English Dictionary* through finite state transduction", Technical Report CS86-20, University of Waterloo, Computer Science Department, 1986.

[Kori99] Noriko Kando, "Text Structure Analysis as a Tool to Make Retrieved Documents Usable", Research and Development Department, National Center for Scientific Information Systems (NACSIS), Japan, 1999.

[Lamp94] L. Lamport, "Latex User Guide and Reference Manual", 2 edition, Addison Wesley, 1994.

[Lent97] B. Lent, R. Agrawal and R. Srikant, "Discovering Trends in Text Databases", *Proc. 3 rd Int Conf. On Knowledge Discovery and Data Mining*, California, 1997.

[Lin01] D. Lin and P. Pantel, "DIRT - Discovery of inference rules from text", *In Proceedings of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2001.

[Litw90] W. Litwin, L. Mark and N. Roussopoulos, "Interoperability of multiple

autonomous databases", *ACM Computing Surveys*, 22(3):267-293, 1990.

[Lui96] R. Lui, "Implementation of Procedures in a Database Programming Language", Master's thesis, School of Computer Science, McGill University, 1996. URL: http://www.cs.mcgill.ca/~tim/cv/theses/lui.ps.gz.

[Matt00] Matthew Young-Lai and Frank Wm. Tompa, "Stochastic Grammatical Inference of Text Database Structure", *Machine Learning*, 40(2): 111-137, 2000

[McHu97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom, "Lore: A database management system for semistructured data", SIGMOD Record, 26(3): 54-66, September 1997.

[Merr84] T. H. Merrett, "Relational Information Systems", Reston Publishing Co., Reston, VA, 1984.

[Merr03] T. H. Merrett, "A Nested Relation Implementation for Semistructured Data", School of Computer Science, McGill University, December 2003.

[Midd01] S. E. Middleton, D. C. De Roure and N. R. Shadbolt, "Capturing Knowledge of User Preferences: ontologies on recommender systems", International Conference On Knowledge Capture, 2001.

[Raym96] D. R. Raymond, F. W. Tompa and D. Wood, "From Data Representation to Data Model: Meta-Semantic Issues in the Evolution of SGML", *Computer Standards and Interfaces 18* (1996) pp. 25-36.

[Salm94] A. Salminen and F. W. Tompa, "PAT expressions: an algebra for text search", *Acta Linguistica Hungarica 41*, 1-4(1992-1993) 1994, 277-306

[Salm96] A. Salminen and F. W. Tompa, "Grammars++ for Modelling Information in Text", Dept. of Computer Science Technical Report CS-96-40, University of Waterloo (November 1996) 46 pp.

[Seba02] F. Sebastiani, "Machine learning in automated text categorization", ACM Computing Surveys, 34(1):1--47, March 2002.

[Shan95] H. Shang. "Trie methods for text and spatial data on secondary storage", Ph.D. Dissertation, School of Computer Science, McGill University, January 1995. URL: http://www.cs.mcgill.ca/~tim/cv/theses/shang.ps.gz.

[Shet90] A. Sheth and J.A. Larson, “Federated database systems for managing distributed, heterogeneous, and autonomous databases”, ACM Computing Surveys, 22(3):183-236, 1990.

[Silv98] C. Silverstein, S. Brin, R. Motwani and J. D. Ullman, "Scalable Techniques for Mining Causal Structures", Proc. 1998 Int. Conf. Very Large Data Bases, pages 594--605, New York, NY, August 1998.

[Sun00] W. Sun, “Updates and Events in a Nested Relation Programming Language”, Master’s thesis, School of Computer Science, McGill University, 2000.
URL: http://www.cs.mcgill.ca/~tim/cv/theses/wSunThesis.html.

[Tana02] L. Tanabe, W. J. Wilbur, “Tagging gene and protein names in biomedical text”, Bioinformatics 18(8) (2002) 1124-1132.

[Thie92] J. Thierry-Mieg and R. Durbin, “Syntactic definitions for the ACeDB data base manager”, Technical report, MRC Laboratory for Molecular Biology, Cambridge, England, 1992.

[Tomp89] F. W. Tompa, “What is (tagged) text?”, *Dictionaries in the Electronic Age: Proc. 5th Conf. of University of Waterloo Centre for the New OED*, Oxford, UK (September 1989) pp. 81-93.

[Tomp92] F. W. Tompa, “Experiences with the OED”, Centre for the New OED and Text Research, University of Waterloo, 1992.
URL: http://db.uwaterloo.ca/~fwtompa/.papers/hist.dict.ps

[Tomp97] F.W. Tompa, “Views of text”, *Digital Media Information Base (DMIB'97)*, Nara, Japan, November 26-28, 1997

[Wein85] E. S. C. Weiner, “The New OED: Problems in the computerization of a Dictionary”, *University Computing*, Vol. 7 (1985) 66-71

[Yu04] Z. Yu, “Implementation of Recursively Nested Relation of JRelix”, School of Computer Science, McGill University, January 2004.
URL: http://www.cs.mcgill.ca/~tim/cv/theses/YuProject.pdf.gz

[Yuan98] Z. Yuan, “Implementation of the domain algebra in Java”, Master’s thesis, School of Computer Science, McGill University, 1998.

URL: http://www.cs.mcgill.ca/~tim/cv/theses/yuan.ps.gz

[Zhen02] Y. Zheng, "Abstract Data Types and Extended Domain Operations in a Nested Relational Algebra", Master's thesis, School of Computer Science, McGill University, 2002.

URL: http://www.cs.mcgill.ca/~tim/cv/theses/yzheng_thesis_ps.tar.gz.