Basic Operators for Semistructured Data in a Relational Programming Language

Yu Gu

School of Computer Science
McGill University, Montreal

December 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 a relational database implementation that supports not only traditional relational algebra and domain algebra but also complex data type and recursive nesting with powerful database programming language. This thesis documents some new features and operators of JRelix. Among them, type polymorphism (union type domain) and the relational pattern search (grep) operator are especially useful when dealing with semistructured data in a relational database.

We use union type domains to increase the flexibility of rigid type definition in relational databases. In addition, we implement the grep operator in the relational algebra to facilitate queries on semistructured data. Grep returns a relation which could contain the type and name of the attribute where it finds the match and the position in that attribute and value of the match. Moreover, we also implement top-level scalar and substring function which are also very useful in a relational database language.

Résumé

JRelix est une implémentation de base de données relationnelle qui supporte non seulement l'algèbre relationnelle traditionnelle et l'algèbre de domaine mais aussi les types de donnée complexes et la récursivité avec un langage de programmation puissant. Ce document de thèse liste quelques nouvelles fonctionnalités et opérateurs de JRelix. Parmi eux, le polymorphisme (« union type domain ») et l'opérateur grep (« relational pattern search ») sont très utiles afin de traiter des données semi structurées en base de données relationnelle.

On utilise le polymorphisme pour augmenter la flexibilité des définitions rigides de type. En addition, nous implémentons l'opérateur grep en algèbre relationnel pour faciliter les requêtes sur des données semi structurées. Grep retourne une relation qui peut contenir le type et le nom de l'attribut où a été trouve l'enregistrement et la position et la valeur de celui-ci. En outre, nous avons aussi implémenté la fonction « top level scalar » et la fonction « substring » qui sont aussi très utiles en base de données relationnelle.

Acknowledgements

My greatest gratitude belongs to my supervisor Professor Tim H. Merrett, for his enthusiastic encouragement, inspiration and patience during my entire study and research at McGill, especially when I was under stress struggling with the writing of this thesis and challenges at work. I cannot imagine how I could have finished this thesis without his guidance, advice, insight, and support.

My appreciation also goes to our Aldat lab coordinator, Zongyan Wang, who integrated our code to JRelix and provided great help during the coding phase. I would also like to thank Fan Guo, who provided her own implemented code so that I could test part of my code which is closely related.

I would like to take this opportunity to thank the School of Computer Science of McGill University for giving me the chance to study here and the faculty for providing the great courses and administrative and technical staff for providing all kinds of help.

I also want to thank Frederick Sauvanet who translated abstract of this thesis into French, and Tim Wai for proof reading my thesis.

Last but not the least, I would like to thank my family members for their support, especially my father Keguang who has always had faith in me, my sister Wei who cares about me, my husband Bin who has shown great patience and support during this entire journey and my son Ziyu who makes me happy and reminds me how beautiful life can be.

Chapter 1 Introduction

This thesis documents the enhancements of JRelix relational database language and several basic operations for supporting semistructured data. In section 1.1, we will present the motivation of having the union type domain and the grep command in JRelix. We will introduce relevant background and related work in section 1.2. In the last section, we will give the outline of this thesis.

1.1 Motivation

The relational data model introduced by E.F. Codd[Codd70] has become a core technique for many commercial database systems. Relational algebra (See section 2.4), which is fundamental to relational query languages, has been extensively investigated in the database research world. Generally, relational algebra defines a set of operators that work on relations. The operators in relational algebra are very similar to traditional set algebra. There are several basic operators that we will introduce in Chapter 2 including projection, selection, and join operators. The result of relational algebra operation is a relation. Combined with the domain algebra (see section 2.5), which works with the domain, relational algebra acts as a strong data manipulating and querying tool in the database world. Yet traditional relational algebra lacks the regular expression search functionality. Wouldn't it be nice to have an operator to search a certain pattern in the whole relation and return a relation as a result, which would contain not only the position where the match found and the type and name of the attribute, but also the context of the match? (See section 1.1.1) In addition, traditional relational algebra does not have polymorphism on attribute type, which is particular useful when dealing with semistructured data. (See section 1.1.2)

1.1.1 Grep Command and Substring Function

What we want to add to the relational algebra is a 'grep' operator which will return a relation as the search result of all the matches on a particular regular expression pattern. Pattern matching has been implemented in several commercial relational database languages. For example, in Oracle10g, we can search a specified column with regular expression. Regular expressions work with SQL REGEXP_LIKE operator and the

REGEXP_INSTR, REGEXP_SUBSTR, and REGEXP_REPLACE functions. These are similar to the existing operator and functions but now offer powerful pattern-matching capabilities [Rischert03]. In MySql, REGEXP operator works the same way as the LIKE operator except that it is followed by regular expression [MySql]. This regular expression search functionality has been added to SQL1999 standard [SQL99]. There are obviously limitations to this approach such as being incapable of searching the whole relation without specifying a particular attribute. SQL works on the individual domain not the whole relation, which is relatively simpler to implement. It is important to have a relational algebra operator 'grep' that searches the regular expression pattern in the entire relation and returns a relation as its result, which contains the attribute and type in which the matches are found, their position and the value of the matches. This approach is particular useful when dealing with semistructured data because of the nature of semistructured data, where the schema of the tuples are variable or mutable. It is difficult and ineffective to specify a search attribute in the "where" clause, as traditional SQL does. Thus, the attribute and type of where the matches are found become even more interesting than the match value itself. In addition, it is of particular interest to return the position and value along with the attribute and type information as well.

In addition to the 'grep' operator, we would like to add a substring function in JRelix to facilitate search functionality. The substring function is easier to implement than the 'grep' operator because it deals with only one attribute at a time. It is implemented in the SQL and other relational database languages.

1.1.2 Union Type Domain

Union types [Merrett03, ACC+97, Buneman97], a form of type polymorphism, are absent in relational algebra. In traditional relational algebra, attribute type is fixed. We want to introduce union types to have type polymorphism in relational algebra to accommodate semistructured data. "Generally, polymorphism refers to the ability to appear in many forms. In object-oriented programming, polymorphism refers to a programming language's ability to process objects differently depending on their data type or class." [Polymorphism05]. Polymorphism is one of the most important features of an object oriented programming language, because it allows many different types of an item to be treated with the same interface [Eckel00]. In the implementation part, the distinction of

different type will be picked up and dealt with respectively. For example, if we want to compute the area of a shape, which can be a circle, square or triangle, we can declare an object to have the type of shape, and initialize it with any value of possible type of circle, square or triangle. Polymorphism enables the programmer to define different area methods for these derived classes, such as circles, square and triangles. No matter what the shape of an object, the area method will return the correct result.

In relational algebra, we want to use union types for the similar purpose. This provides the same interface in the relational algebra command no matter the underlying type attribute value. For example, a phone number can be an integer or a string type. But with traditional relational algebra, the attribute type of phoneNum has to be fixed before the data can be populated. For the same reason, query on an attribute is type specified. The following query "where phoneNum=8884357" assumes that attribute phoneNum has integer type. What if phone number is expressed as '888-4357' or '888help' and we want also to allow string type of phone number to be stored in the relation. This problem will be solved if we define the type of attribute phoneNum to be either string or integer. In this case "where phoneNum=8884357" and "where phoneNum='888-4357'" are both valid queries.

We allow union type to be defined not only on primitive types but also on previously defined types. For example, we can declare zipcode as type of integer, and postcode as type of string. Then we declare another domain with name postalcode and type of either zipcode or postcode to cover the situation of where zipcode looks like 12345 and postcode looks like 'a1b2c3'. We use postalcode to search in the "where" clause. This way the query will return the correct result no matter what the integer type or string type is of the attribute value.

Since a union type domain can be defined on a previously defined domain, and the domain can be a nested relation, then it should be allowed to define on nested domains. For example, street address can be a bunch of strings or a bunch of combination of street number and street name. So it can be defined as a union type of two nested domains, namely street1 and street2. Domain street1 is defined as a relation that has an attribute streetInfo and type of string. Domain street2 is defined as a relation that has two string type attributes: streetNum and streetName (See [Merrett03] in detail).

We want to add union types to relational algebra to cover the situation like we discussed in the previous examples. No matter the integer or string type of phoneNum, and different complex type of street address, the operation on union type domains will return the correct result. Union type can be quite useful in relational algebra to manipulate semistructured data. Since relational data is strictly typed, using union type domain will loosen the constraints on attribute types, thereby accommodating semistructured data in the relational database. In the underlying implementation, we need to deal with different types of a union type domain to make sure the operation on union type will return the correct result. This poses a challenge of schema matching when one initializes the relation with semistructured data.

1.1.3 Top Level Scalar

In this thesis, we will also introduce the top level scalar which also works closely with nested relation. Top level scalar is a relation level constant. It can be used in any relational and domain algebra. For example, in relation R(x,y), x and y are two attributes of R. We can have domain level scalar, say "let z be x+y", then z is a virtual attribute which is a domain level attribute. Similarly, we can have a scalar at relation level. Since it is at the same level as a relation, it can be particular useful when dealing with nested relations. In JRelix, relational algebra is subsumed by domain algebra. We allow relational operations in domain algebra [Merrett84].

1.2 Background and Related Work

In this section, we will review the history of the relational data model and give an introduction to semistructured data research. In addition, we will survey regular expression search tools, namely the UNIX grep family, and other tools for regular expression searching.

1.2.1 Relational Database

After E.F. Codd first introduced relational data model [Codd70] 35 years ago, it has become the core technique of most database management systems. The earliest relational algebra language is PRTV (Peterlee Relational Test Vehicle [Todd76]). The earliest relational database system projects were IBM System R [ABC+76] and Ingres (INteractive Graphics REtrieval System[SHWK76]). The structured query language (SQL), which was first developed at IBM, won out Ingres's QUEL and finally became the industry standard with ANSI (American National Standards Institute) in 1986 and ISO (International Standards Organization) in 1987. Eventually System R evolved into SQL/DS, which later became DB2. Many other commercial database systems such as Informix, Sybase, and Microsoft SQL Server have been developed based on the source code of Ingres.

The relational data model is thought of as two-dimensional tables, known as a relation. Columns of the table are labeled and the names of the column are called attributes. The values of the attributes are called domains. A row in the table is called a tuple. The properties of a relation were first given by E.F.Codd in [Codd70]; where no restrictions were made on relation to be flat. He introduced normal forms later [Codd71] [Codd72], and this significantly affected the research direction and the development of commercial relational databases. The following are the basic properties of a flat relation or first normal form.

1) All rows are distinct.
2) The ordering of rows is immaterial.
3) Each column is labeled, making the ordering of columns insignificant.
4) The value in each row under a given column is "simple". [Merrett84]

"Simple" in the last property means that the type of the column can only be of a primitive type such as String, Integer, Boolean, etc. This excludes the possibility of nested relation in which a value of a column can be a relation.

1.2.2 Nested Relation

By removing the last property of a flat relation, a relation can become nested. This idea was first brought up by Codd with the relational data model. He wrote "Nonatomic values can be discussed within the relational framework. Thus, some domains may have relations as elements. These relations may, in turn, be defined on nonsimple domains, and so on." [Codd70]. In this paper, he refers to relations with nonsimple domains as unnormalized set and proposed an approach to eliminate nonsimple domains. He called this normalization.

The nested relation was formally proposed by Makinouchi [Mak77]. It provides a natural presentsation of hierarchical data. Fisher and Van Gucht first discussed the one level nested relation [FG85]. This was generalized to arbitrary depth by Thomas and Fisher later [TF86]. There are two operators which transform between flat relations and nested relations, namely, nest and unnest which were first introduced by Jaeschke and Schek [JS82]. These operators are superfluous if we include relational operator in domain algebra. The nested relation is implemented in JRelix (see section 2.6). In JRelix, relational algebra is subsumed by domain algebra. Domain algebra operators can be used on a nested domain, which is a relation.

1.2.3 Semistructured data

In between the strictly structured data that a relational database dealt with and the non structured data such as raw data (e.g., image and sound) there are a lot of data that have some extent of structure, either explicit or implicit in the data. We call these data semistructured data. Most of the data that reside on the World-Wide Web especially XML [BPM+04] fall into this category. The need to integrate wide a variety of data formats and data found on the Web has brought the topic of semistructured data to the forefront of research [Abiteboul97].

"Roughly speaking, semistructured data are data that are neither raw data, nor very strictly typed as in conventional database systems" [Abiteboul97]. Abiteboul discussed several main aspects of semistructured data in this paper. The following is a list of some basic features.

1) The structure is irregular

As in the previous phone number example, the type can be integer or string. The same information can be expressed either as a simple element or in a complex structure. For example, street information can be a string type element for one tuple, or a set of tuples of street number and street name, etc.

2) The structure is implicit

Some semistructured data have structure implicit to the data. For example, XML text has tags to indicate the structure, therefore we need tools like parser to extract it from the data.

3) The structure is indicative not constraining

The structure in semistructured data is usually for the purpose of describing the data, not to set type constraints on the data. So strict type is not necessary and proper for semistructured data.

4) Schema is not pre-defined

Unlike traditional relational database, the schema of semistructured data is not well defined before the data are populated. Rather, it is extracted or discovered from the data.

5) The schema could be very large and queried as data

Semistructured data may come from different sources and may have quite large schema and relatively small data. For example, one may get a list of local restaurants from the

Web, each restaurants may have quite different information from others. So the schema may get very large and querying on schema will be as important querying on data.

6) The schema is ignored

Sometimes it is very useful to ignore the schema. For example, we just want to browse the data or search a particular string in the entire data. Although this is not normally possible with traditional relational query languages, it is necessary to have string searching tools to be added to the query languages.

7) The schema evolves rapidly

Unlike traditional relational database, in which the schema is almost immutable, with semistructured data, the schema can change repidly. A good example is ACeDB (A C. elegans Database) which was originally developed as a database to store genetic data relating to the worm [TMD92]. Schema and data in ACeDB can be seen as edge-labeled trees. This format was preferable to a relational or object database system due to the fact that it is not easy to deal with the dynamic changing schema in traditional database systems.

8) The distinction between schema and data is blurred

In relational database, the schema captures the structure of the data. In semistructured data, this distinction may not make much sense. For example, the sex of a person can be used as schema in one source: Male or Female, and the possible value are 'Y' or 'N'; or it can be data in another source: Sex, with the values 'male' or 'female'.

Research on manipulating semistructured data

A lot of database research has been done on manipulating and querying semistructured data. Due to the above nature of semistructured data, it is often considered and constructed as a directed graph or tree structure with arbitrary depth. For example, ACeDB data, XML data, Lore data. There is no effective way or almost impossible to query this data structure by traditional relational or object oriented query languages. A new flavor query language is needed.

Briefly, there are two ways to build query languages for semistructured data. The first one is to start from SQL (or OQL[CD92, ODMG 3.0]) and add enough "features" to be able to query semistructured data. The second approach is to start from a formal language on

semistructured data then to massage that language into an acceptable syntax. These two approaches end up with very similar languages [Buneman97].

Lore and Lorel

Lore is a database management system designed particularly for semistructured data[MAG+97] . Data in Lore is thought of as a directed graph. Notes are objects and can be identified by a unique object identifier. Leaf nodes contain values of primitive types of database and other nodes contain pointers to other objects. The edges are labeled with the name of the object to which they point. Lorel is a SQL/OQL style language for querying semistructured data and can be viewed as an extension of OQL of ODMG data model [AQMWW97]. Lorel provides simple path expression in each part of its "select-from-where" form of language. Using Path expressions provides a way to the user to query the data without even knowing the schema of the database. This is a desired feature for querying semistructured data because the schema is not fixed for each record and may be dynamically changing. Lorel can bring information to the surface, but it is not capable of performing complex restructuring of the data, such as "deleting/collapsing edges with a certain property, relabeling edges, or performing local interchanges [Buneman97]".

UnQL

In contrast, UnQL (Unstructured Query Langauge) is capable of various forms of restructuring as mentioned above. For example, a traverse construct [BDHS96] allows one to transform a database graph while traversing it, for instance, to replace all labels A by the label B. "This powerful operation combines tree rewriting techniques with some control obtained by a guided traversal of the graph. For instance, one could specify that the replacement occurs only if particular edge, say B, is encountered on the way from the root" [Abiteboul97]. UnQL starts from structural recursion [BBW92, BLS+94, BNTW95]. Its data model consists of a rooted, labeled graph, or alternatively, it can be thought of as a tree. The leaves represent atomic values and the non-leaf nodes represent objects. Objects do not have identifiers as in Lore. Non-atomic values are represented as sets of label/value pairs. This model can be used to represent relational database. The fixed depth structure of UnQL has the expressive power of nested relational algebra. It can also

traverse arbitrary depth by using regular expression on the path [BDS95, BDHS96, BFS00].

XML

"XML is a new standard adopted by the World Wide Web Consortium (W3C) to complement HTML for data exchange on the Web" [ABS00]. Like HTML, XML also uses tags to enclose an element, but the tags of XML are user defined and represent the structure of the data. We can see them as attribute names in a relation. All the tags must appear in pairs, and the start tag and the end tag have identical names except that the end tag starts with a "/". The content in between the start tag and end tag can be a string and other elements that quoted with tags, thus we can view this structure as a nested relation.
Many query languages have been proposed for XML, for example, XML-QL [DFFS98] uses path expression and patterns to extract data from XML data. The query result is constructed as XML data as well. Another example is XQuery [BCF+04], it has been proposed as the W3C standard query language for XML data. Other XML query languages are: Quilt [RCF00], XQL[DFH+99].

Besides the work on building database and query language particular for semistructured data, considerable effort towards adding semistructured data manipulating features to existing relational databases have been made. For example, Oracle adds XML data and query language to its database by providing a new data type, XMLType, and XPath [CD99] query language [GSS04]. The users need to know both SQL and XPath in order to query XML in Oracle, but if we have a relational implementation on semistructured data, then we do not need to add separate things and the user can use a single language for everything. That is what we are trying to do in JRelix. Currently JRelix has some features to support semistructured data, like path expression [Yu04], metadata operators [Guo05], etc. We will take a quick look at JRelix in the following section.

1.2.4 Semistructured data in the Aldat Project

JRelix is a relational language that can work on nested relations [Hao98]. It is quite natural to take advantage of this ability and add some new features like path expressions [Yu04], metadata operators [Guo05], grep commands on test data [Xie05] and union

types to accommodate and query semistructured data, which itself is nested in nature. WE will briefly review what has already been done so far in JRelix to support semistructured data in this section and then will give an overview of JRelix in Chapter 2 and explain its implementation in Chapter 4.

JRelix has been extended to support semistructured data coherently due to its native ability to deal with nested structures of data. In Zhan's project [Yu04], he implemented several new features in JRelix to support semistructured data, including semi-structured data loading, improved queries to support recursive nesting, path expression and regular expression operators. "After allowing JRelix to accept recursive nesting, the regular expression operators ("*", "+", ".", "?") have been implemented to query the relations with a recursively nesting domain. In addition, a path operator, which may be frequently used in querying nested relations, has been implemented as a short-cut by using the / operator. "

In Fan's thesis [Guo05], she describes the implementation of several new features of attribute metadata in JRelix for supporting semistructured data. These operators help to manipulate the metadata of a relation. With wild card in path expression together, they provide ways to query in arbitrary levels of nested relations and facilitate schema discovery in a nested relation. The operators include *quote*, *eval*, *transpose*, *typeof* , *relation* and *self*.

In Jiantao's thesis [Xie05], he proposed an Aldat extension within which text can be described and manipulated meaningfully as an equivalent of a relation. The flat text or nested text can be loaded in and queried as a realtion. The extended operations are simple, intuitive and 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." In addition to the extension of existing operators to manipulate text, several new operators has been added, such as binary grep operators like igrep, ugrep, dgrep, sgrep (diff), lgrep and rgrep, which are used to provide searching in text and text2attr and mu2nest and to convert between texts and relations.

In this thesis, we extend JRelix operations to further support semistructured data. Union type domain which supports type polymorphism makes it easier to process semistructured data in relational database. The Grep operator, which searchs the entire relation and returns the attribute and type along with the position and value of where the pattern is found, becomes particular useful when dealing with semistructured data. We will show more of the details in the following chapters.

1.2.5 String searching and grep command

String searching has been a very important subject in text processing. There are many string searching algorithms implemented in all kinds of software under most operating systems. There are two catagories of searching algorithm, exact string matching and regular expression string matching. For example, the "grep" operator in UNIX is a regular expression searching tool, and the "find" command in Microsoft Office is an exact string matching tool. Many of the algorithms need preprocessing phase. Some create tables for the pattern, some build automata for the pattern, some use hash tables. Many of the regular expression algorithms use similar ideas as those for exact string matching plus a preprocessing phase to build automaton for the pattern. We will give a short overview on some well-known exact string searching algorithms and regular expression matching algorithms.

The simplest and earliest string searching algorithm is the brute force algorithm. Suppose the length of string that need to be found or the pattern has length of m, and the text to search against has length of n. The brute force algorithm checks at all positions in the text between 0 to n-m, whether the pattern starts from there or not. Then, after each attempt, it shifts the pattern one position to the right. The time complexity of this searching phase is O(mn). This algorithm has been studied carefully and has been enhanced in various ways by different researchers.

Knuth, Morris and Pratt discovered first linear time string-matching algorithm [KMP77]. This retains the information that the brute force approach wasted by gathering it during the scan of the text. It uses this information to reduce the number of times it compares each character in the text to a character in the pattern. It only looks at each character in the text. It achieves a running time of O(n + m), which is optimal in the worst case

scenario. In the worst case Knuth-Morris-Pratt algorithm has to examine all the characters in the text and pattern at least once. Unlike the brute force algorithm, Knuth-Morris-Pratt algorithm needs to preprocess the pattern. It creates a table based on the pattern and uses the table during the matching process.

Another efficient string searching algorithm was developed by Bob Boyer and J Strother Moore in 1977 [BM77]. The algorithm preprocesses the pattern that is being searched for. "It doesn't need to actually check every character of the pattern but rather skips over some of them. Its efficiency derives from the fact that, with each unsuccessful attempt to find a match between the pattern and the text it's searching in, it uses the information gained from that attempt to rule out as many positions of the text as possible where the string could not match" [Wikipedia05]. "This kind of skip is the goal of the Boyer-Moore algorithm. Unlike the Knuth-Morris-Pratt algorithm, which strives to look at each character in the text only once, the Boyer-Moore algorithm strives to completely ignore as many characters in the text as possible" [Ellard97]. The worst case behavior of the algorithm is linear in i + m, where i is the first position of a match found in search string.

There are many other algorithms derived from the above algorithms. To name a few, the Reverse Factor algorithm [Leeroq92] and the Turbo Reverse Factor algorithm [Lecroq95] were derived from Boyer-Moore algorithm and they both need to build an automaton in the preprocessing phase. Horspool-BM algorithm is a simplification of Boyer-Moore algorithm[Horspool80] and Turbo-BM is an amelioration of Boyer-Moore algorithm[CCGJ$^+$92].

One of the most broadly used regular expression search tool is the grep command in UNIX. Grep originates from the UNIX command **ed** (written by Kenneth Thompson [KD71]). UNIX users often use the **ed** command to search a string. For example, in the following command: s/*old*/*new*/**g**, **s** stands for substitution, **g** stand for globally, and / follows by the string or pattern that is searched. So this command will replace all "*old*" with "*new*" in the file. Another frequently used one is to globally search a pattern and print the result. For example **:g**/*car*/**p**, it prints all the line which contains string "*car*" . According to Douglas McIlroy [HH96], grep was invented by Kenneth Thompson, who provided regular expression search functionality as a independent command from UNIX text editor **ed**. Since in between **g** and **p**, there can be a regular expression, it was then

called grep. It represents "**g**lobal **r**egular **e**xpression **p**rint". Grep is listed in the Manual for Version 4 UNIX which is dated November, 1973 [HH96]. The first algorithm for seaching regular expression was published in 1968 by Kenneth Thompson [Thompson68]. In this paper, Thompson introduces a fast regular expression recognition technique and algorithm. "Each character in the text to be searched is examined in sequence against a list of all possible current characters. During this examination a new list of all possible next characters is built. When the end of the current list is reached, the new list becomes the current list, the next character is obtained, and the process continues. In the terms of Brzozowski [Brzozowski64], this algorithm continually takes the left derivative of the given regular expression with respect to the text to be searched. The parallel nature of this algorithm makes it extremely fast."

Thompson was the first to implement grep. Before that, Glantz did some of the earliest work on automated pattern matching [Glantz57]. Thompson's implementation of grep was based on building a nondeterministic finite automaton (NFA) for the regular expression. In 1975, Alfred Aho and Margaret Corasick made a new version of grep called egrep (enhanced grep) [AC75]. Hume later showed that egrep was faster than grep for simpler patterns because it used a deterministic finite automata (DFA) but was slower for longer patterns because of the setup time required to build a complete DFA [Hume88]. Egrep was later enhanced by using a lazy evaluation, which took zero setup time and just one additional test in the inner loop [AA04]. Currently, egrep stands for extended grep because it supports extended regular expressions, as opposed to the default grep patterns which are basic regular expressions [IEEE03a].

There are also other variants of regular expression search tools, such as awk [AWK88], GNU awk [Robbins98]. The following is a table of some of such tools.

Some Tools and Their Regex Engines (based on [Friedl97])

Program	**(Original) Author**	**Reference**	**Regex Engine**
awk	Aho, Weinberger, Kernighan	[Awk88]	DFA
new *awk*	Brian Kernighan	[Awk88]	DFA
GNU *awk*	Arnold Robbins	[Robbins98]	Mostly DFA, some NFA
MKS *awk*	Mortice Kern Systems	[MKSawk]	POSIX NFA
mawk	Mike Brennan	[Brennan91]	POSIX NFA
egrep	Alfred Aho	[Aho90]	DFA
MKS *egrep*	Mortice Kern Systems	[MKSegrep]	POSIX NFA

GNU Emacs	Richard Stallman	[CR91]	Trad. NFA (POSIX NFA available)
Expect	Don Libes	[Libes95]	Traditional NFA
expr	Dick Haight	[Friedl97]	Traditional NFA
grep	Ken Thompson	[Thompson68]	Traditional NFA
GNU *grep*	Mike Haertel	[GNUgrep05]	Mostly DFA, but some NFA
GNU *find*	GNU	[Findutils05]	Traditional NFA
flex	Vern Paxson	[Paxson00]	DFA
lex	Mortice Kern Systems	[MB90]	POSIX NFA
more	Eric Schienbrood	[Shienbrood92]	Traditional NFA
less	Mark Nudelman	[Nudelman05]	Variable (usually Trad. NFA)
Perl	Larry Wall	[Wall00]	Traditional NFA
Python	Guido van Rossum	[Rossum]	Traditional NFA
sed	Lee McMahon	[DR97]	Traditional NFA
Tcl	John Ousterhout	[Ousterhout94]	Traditional NFA
Vi	Bill Joy	[LR98]	Traditional NFA

1.3 Thesis outline

This thesis introduces several new features and operators that added to JRelix system including the grep command in relational algebra, substring functions, union type domains and top level scalar. In chapter 1, we describe our research motivation on union type domains and regular expression search algorithms and how they relate to traditional relational database and semistructured data. In chapter 2, we will give an overview of JRelix. Instead of a detailed review of the system;, we will provide a fundamental description of JRelix operations, in order for the reader to understand the discussions in the following chapters. In chapter 3, we will introduce the new features and operators that we proposed in JRelix in the form of a user's manual for database programmers. In chapter 4, we will present implementation details of those new features and operators and finally in last chapter we will summarize this thesis and discuss future directions. We will also include an appendix which contains all pertinent JRelix syntax (about 80% of all) as improved by this thesis.

Chapter 2 JRelix Overview

In this chapter, we will briefly introduce the JRelix database system. In section 2.1, we will introduce how to start the JRelix system, where section 2.2 outlines some of the most frequently used commands. In section 2.3 we will introduce how we declare a domain and a relation. In section 2.4 and 2.5 we will discuss relational algebra and domain algebra respectively. In section 2.6, we will introduce the nested relation. A review of other commands can be found on the web at CS612 course website [Merrett00]. We will not give expanded syntax for each command that we discuss. We will put detailed syntax in the Appendix. For many examples, we will reference the user's manual in Chapter 3, rather than give them here.

2.1 Getting Started

JRelix is implemented in Java. It can be started under the Java run time environment. To start, type the following command under the command line of either Windows or UNIX system.

```
java JRelix
```

The screen will show a prompt ">" after JRelix started. User can input JRelix command after ">".

2.2 Commands

Command	Function	Example
pr <relational expression>	Print the result of relational expression	Figure 3.1.1
sd (<Identifier>)?	Show domain definition	Figure 3.3.1
sr (<Identifier>)?	Show relation definition	Figure 3.3.1
quit	Quit JRelix system	

2.3 Declaration and Initialization

2.3.1 Domain Declaration

```
domain <IDList> <Type>
```

'domain' is the key word to indicate that this is a domain declaration command. IDList is a list of identifiers to represent the names of domains. Type can be simple type like string

or integer or complex type like nested relation. We have examples in Figure 3.2.1 for both simple type and nested domain declaration. We will expand this definition further to include union types in section 3.2.

2.3.2 Relation Declaration and Initialization

```
relation <IDList> "("<IDList> ")" (<Initialization>)?
```

The first IDList represents the names of relations that have been declared. The second IDList contains the domain list of the relation. Initialization is optional, we can initialize the relation now or do it later using assignment operator "<-". Basically, we can initialize a relation with another relation or from a file in addition to assigning values to each attribute in each tuple. The following is the syntax for initialization:

```
<Initialization> :
        <ASSIGN>
        ("{" <TupleList> "}" | <Identifier> | <FilePath> )
<TupleList> :   (  <Tuple> ( "," <Tuple> )* )?
```

We have examples in Figure 3.1.1. We can easily apply this syntax to relation declaration and initialization to relations which have union type domains. We will introduce examples in section 3.2. It is important to note that in the syntax, we assign a TupleList to the relation. TupleList is a set of Tuples. A Tuple can be a set of literals or TupleList. In case of TupleList, it represents a nested relation. We will introduce it in section 2.6.

2.4 Relational Algebra

Relational algebra is one of the most essential parts of relational databases. It provides operators to manipulate relations. The operators are functional, i.e., they take relations as inputs and produce a relation as a result. We can categorize them into unary and binary operators.

2.4.1 Unary Operators

The unary operator takes one operand of relation. Here we have the three most commonly used unary operators:

Projection

Projection produces a new relation on a subset of the attributes of the operand. Duplicates are eliminated during projection. We will use projection in an example shown in Figure 3.4.1.

Syntax:

```
<Projection>: "["(<ExpressionList>)?"]" in <Projection>|<Selection>)
              |
              <Selection>
```

Selection

Selection produces a new relation with tuples that satisfy a specific condition. We will use selection in an example shown in Figure 3.2.6.

Syntax:

```
<Selection>: where <Expression> in <Projection>
             |...| <Term>
```

(Term can be expanded to be Identifier, see Appendix for detail)

T-Selection

Projection and selection can be combined in a single operation, called T-selection. We will use T-selection in an example shown in Figure 3.4.5.

Syntax:

```
"["(<ExpressionList>)?"]"where < Expression > in
(< Projection >|<Selection>)
```

2.4.2 Binary Operators

Binary operators take two relations as input, and join them together to produce a result relation. There are two types of join operator, i.e. μ-joins and σ-joins. They both have the following syntax.

Syntax:

```
<JoinExpression>: <Projection>
(<JoinOperator> <Projection>
|
"[" <ExpressionList> ":"  <JoinOperator> ":" < ExpressionList > "]"
<Projection>
)*
```

μ-joins

The μ-join operators are extended from set operators, which include intersection, union and difference. We will use ijoin (intersection) in examples in section 3.2.3. This

produces a set of tuples which join together the tuples from two relations that have same values on the join attributes. Union and difference operators are similar to their respective set operators. To illustrate the definitions of these joins, suppose we want to join two relations R(A,B) and S(C,D) on attribute set B and C. B and C can be a set of attributes to which two relations are joined. They must have the same number of attributes, but the name and type need not be the same. In the case that they are the same, the attribute set of the result relation will be (A,B,D). We first define left, center and right as the following. Then we will define these μ-joins in the below table.

$center(R, S) \equiv \{(a,b,c,d) | (a, b) \in R \wedge (c, d) \in S \wedge b = c\}$

$left(R, S) \equiv \{(a, b, c, DC) | (a, b) \in R \wedge b = c \wedge \forall d, (c, d) \notin S\}$

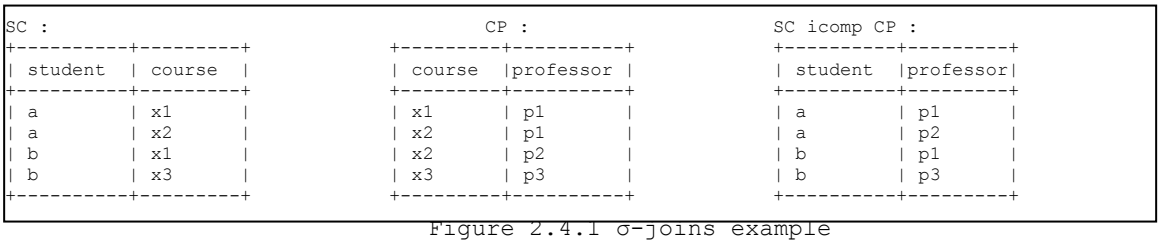
$right(R, S) \equiv \{(DC, b, c, d) | (c, d) \in S \wedge b = c \wedge \forall a, (a, b) \notin R\}$

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

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

σ-joins

The σ-join extends the truth-valued comparison operations on sets to relation. The comparison happened between the sets of values of the join attributes grouped on each other attributes. For example, we have two tables: SC(students take courses) and CP(courses instructed by professor) and we want to join these tables on attributes course. We will get a result relation that has two attributes: student and professor. We will group SC on student and CP on professor and compare for each group the set of courses. For operation **icomp**, we will get the following result for the given SC and CP.



```
SC :                                CP :                      SC icomp CP :
+----------+---------+              +---------+----------+    +----------+---------+
| student  | course  |              | course  |professor |    | student  |professor|
+----------+---------+              +---------+----------+    +----------+---------+
| a        | x1      |              | x1      | p1       |    | a        | p1      |
| a        | x2      |              | x2      | p1       |    | a        | p2      |
| b        | x1      |              | x2      | p2       |    | b        | p1      |
| b        | x3      |              | x3      | p3       |    | b        | p3      |
+----------+---------+              +---------+----------+    +----------+---------+
```

Figure 2.4.1 σ-joins example

We can define the σ-joins using the following notation. In relations R(A, B) and S(C, D), $R_a$is the set of values B associated by R with a given value, a, of A, and S_z is the set of values of C associated by S with a given value, d, of D. If A and B are disjoint sets of the attributes of R, and C and D are disjoint sets of the attributes of S, we can give the definitions as below.

σ-joins	**Operator**	**Name**	**Definition**
overlap	R **icomp** S	natural composition	$\{(a, d)\|R_a \cap S_d \neq \emptyset\}$
not overlap	R **sep** S	empty intersection join	$\{(a, d)\| R_a \cap S_d = \emptyset\}$
superset	R **sup** S	greater than or equal join	$\{(a, d)\| R_a \supseteq S_d \}$
proper superset	R **gtjoin** S	greater than join	$\{(a, d)\| R_a \supset S_d \}$
subset	R **lejoin** S	less than or equal join	$\{(a, d)\| R_a \subseteq S_d \}$
proper subset	R **ltjoin** S	less than join	$\{(a, d)\| R_a \subset S_d \}$
equal	R **eqjoin** S	equal join	$\{(a, d)\| R_a = S_d \}$

2.5 Domain Algebra

Unlike relational algebra, domain algebra works on attributes, not on relations. There are two types of operations: one is the scalar operation, which works within a tuple, while the other is the aggregate operation, which works across tuples. They both use virtual domains to define the operations. The syntax is the follwing. (See Appendix for the full description on Expression).

```
<let> <identifier> be <Expression>
<Expression>: ...<substr>   (for example: Expression can be substr)
```

We will have several opportunities to use virtual domains to define scalar operations in section 3.4 when we introduce substring function. Aggregation operators work with a group of tuples. For example we can get the min, max or sum of an attribute for a group of tuples or for all tuples in the table. We will have an example of “red +” in Figure 2.6.3.

2.6 Nested relation

Another very important feature is the concept and implementation of nested relations and recursive nesting. We will have examples when we introduce union type domains in section 3.2. The main idea of a nested relation is to treat a relation as a domain, that is, a domain can be a relation. So we extend the type of a domain from a primitive type to a complex type, namely a nested relation.

The syntax of declaring a nested domain is to simply change the "`relation`" to the "`domain`" as the following:

```
domain <IDList> "(" <IDList> ")"
```

The initialization of nested relation use the same syntax as a flat relation except here in a Tuple we can have a TupleList as its attribute. Let's take a look at the expanded syntax on Tuple.

```
<Tuple> : "(" <Constant> ( "," <Constant> )* ")"
<Constant> : <Literal> | "{" <TupleList> "}"
```

This example is quite similar to the example we will give in Figure 3.2.4 except that a union type domain is not used.

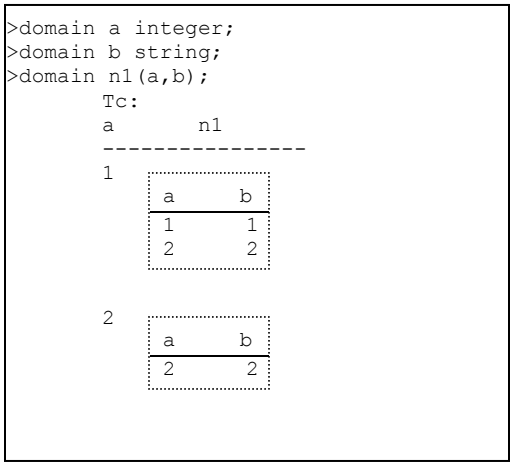


Figure 2.6.1 nested relation example

The way the nested relation is actually stored in the belonging table is by using a surrogate key. This surrogate key will be stored together with other attributes of a tuple. The actual content of the nested relation is stored in a relation with a name such as "." plus the name of the nested domain. In our example, the name will be ".n1". In relation ".n1" there is an .id attribute which will be used to store surrogate key so that the relative

tuples can be linked to the relation that contain this nested relation as its attribute. The following figure shows the way nested relation is stored.

```
relation Tc(a,n1)<-{(1,{(1,"1"),(2,"2")}),(2,{(2,"2")})};
pr Tc;
+-------------+----------------------+
| a           | n1                   |
+-------------+----------------------+
| 1           | 1                    |
| 2           | 2                    |
+-------------+----------------------+
relation Tc has 2 tuples
>pr .n1;
+---------------------+-------------+----------------------+
| .id                 | a           | b                    |
+---------------------+-------------+----------------------+
| 1                   | 1           | 1                    |
| 1                   | 2           | 2                    |
| 2                   | 2           | 2                    |
+---------------------+-------------+----------------------+
relation .n1 has 3 tuples
```

Figure 2.6.2 nested relation storage

Unnesting can be done by two ways: by raising the level through anonymity or by uniting tuples through reduction. The following figure gives two examples of level raising. Please find more examples from course notes of cs612 on nested relations [Merrett00].

```
>let redA be [red + of a] in n1;
>pr [a,redA] in Tc;
+-------------+-------------+
| a           | redA        |
+-------------+-------------+
| 1           | 3           |
| 2           | 2           |
+-------------+-------------+
expression has 2 tuples
```

Figure 2.6.3 level raising in nested relation

Chapter 3 Users' Manual

3.1 Grep

As described in [Merrett03], the "grep" is a new operator in JRelix for textual pattern matching. The purpose of it is to find a substring or a match with a particular pattern in databases. This pattern can be any regular expression. In this thesis, we will use the regular expression that is compatible with the unix egrep convention.

3.1.1 An Example

The following example shows what grep can do to find string "hello" in relation R.

```
>domain x intg;
>domain y,z strg;
>relation R(x,y,z)<-{(10,"x1","y1"),(20,"x2","y2"),(30,"x3 hello", "y3 hi,hello
there"),(40,".*+?\[]^$|","hello sam")};
>pr R;
+-------------+----------------------+----------------------+
| x           | y                    | z                    |
+-------------+----------------------+----------------------+
| 10          | x1                   | y1                   |
| 20          | x2                   | y2                   |
| 30          | x3 hello             | y3 hi,hello there    |
| 40          | .*+?\[]^$|           | hello sam            |
+-------------+----------------------+----------------------+
expression has 4 tuples
>pr grep "hello" in R;
+-------------+----------------------+----------------------+
| x           | y                    | z                    |
+-------------+----------------------+----------------------+
| 30          | x3 hello             | y3 hi,hello there    |
| 40          | .*+?\[]^$|           | hello sam            |
+-------------+----------------------+----------------------+
expression has 2 tuples
```

Figure 3.1.1 A simple example of grep

3.1.2 Parameter List 1 of Grep

The grep command can have two parameter lists: in parameter List 1, one can specify up to 4 parameters. They can have the types of integer, type, attribute or universal. The purpose of using parameter List 1 is to determine attribute, type, position, the grep finds the matching value. Since the grep searches all the attributes of a relation, the value can come from all possible types, so the value that contains the pattern is of universal type.

For example, the following grep operation will return the attribute where the grep finds the string "hello", the type of the attribute, the position at which it finds the string and the value that matches the pattern. In general, the value that matches a string is exactly the same as the string.

```
>domain Type type;
>domain Attr attribute;
>domain Pos integer;
>domain Val universal;
>pr grep(Attr,Type,Pos,Val) "hello" in R;
+--------------------+----------------------+------+--------+-------------+--------------+
| x   | y            | z                    | Attr | Type   | Pos         | Val          |
+-----+--------------+----------------------+------+--------+-------------+--------------+
| 30  | x3 hello     | y3 hi,hello there    | y    | string | 3           | string:hello |
| 30  | x3 hello     | y3 hi,hello there    | z    | string | 6           | string:hello |
| 40  | .*+?\[]^$|   | hello sam            | z    | string | 0           | string:hello |
+-----+--------------+----------------------+------+--------+-------------+--------------+
expression has 3 tuples
```

Figure 3.1.2 An example of grep with parameter List 1

3.1.3 Regular expression of Grep

In our implementation, we use the same regular expression as egrep. The following table is a summary of what we can use in our grep.

.	Represent any single character	grep "h." in R;
*	Zero or more times	grep "h.*" in R;
|	Or	grep "0|h" in R;
[]	One of a set of characters	grep "[xh]" in R;
()	Characters as a whole	grep "(xh)" in R;
^	The beginning of an attribute	grep "^h" in R;
$	The end of an attribute	grep "h.*o$" in R;
?	Zero or once	grep "he?o" in R;
+	One or more times	grep "h.+" in R;
\	Escape character for .*+?\[]()^$|	grep "\." in R;

Figure 3.1.3 Regular expression in pattern

Here are some examples:

Example 1: Search anything that has an "h" and is followed by a character in relation R.

```
>pr grep(Attr,Type,Pos,Val) "h." in R;
+----+----------------------+----------------------+------+--------+-----+----------------------+
| x  | y                    | z                    | Attr | Type   | Pos | Val                  |
+----+----------------------+----------------------+------+--------+-----+----------------------+
| 30 | x3 hello             | y3 hi,hello there    | y    | string | 3   | string:he            |
| 30 | x3 hello             | y3 hi,hello there    | z    | string | 3   | string:hi            |
| 30 | x3 hello             | y3 hi,hello there    | z    | string | 6   | string:he            |
| 30 | x3 hello             | y3 hi,hello there    | z    | string | 13  | string:he            |
| 40 | .*+?\[]^$|           | hello sam            | z    | string | 0   | string:he            |
+----+----------------------+----------------------+------+--------+-----+----------------------+
expression has 5 tuples
```

Figure 3.1.4 Example 1

Example 2: Search anything that has an "h" and is followed by zero or more characters in R.

```
>pr grep(Attr,Type,Pos,Val) "h.*" in R;
+-----+---------------------+---------------------+------+--------+-----+----------------------+
| x   | y                   | z                   | Attr | Type   | Pos | Val                  |
+-----+---------------------+---------------------+------+--------+-----+----------------------+
| 30  | x3 hello            | y3 hi,hello there   | y    | string | 3   | string:hello         |
| 30  | x3 hello            | y3 hi,hello there   | z    | string | 3   | string:hi,hello there|
| 30  | x3 hello            | y3 hi,hello there   | z    | string | 6   | string:hello there   |
| 30  | x3 hello            | y3 hi,hello there   | z    | string | 13  | string:here          |
| 40  | .*+?\[]^$|          | hello sam           | z    | string | 0   | string:hello         |
+-----+---------------------+---------------------+------+--------+-----+----------------------+
expression has 5 tuples
```

Figure 3.1.5 Example 2

Example 3: Search anything that has an "h" or an "x" and is followed by zero or more characters in R.

```
>pr grep(Attr,Type,Val,Pos) "[hx].*" in R;
+-----+----------------+---------------------+------+---------+----------------------+-------+
| x   | y              | z                   | Attr | Type    | Val                  | Pos   |
+-----+----------------+---------------------+------+---------+----------------------+-------+
| 10  | x1             | y1                  | y    | string  | string:x1            | 0     |
| 20  | x2             | y2                  | y    | string  | string:x2            | 0     |
| 30  | x3 hello       | y3 hi,hello there   | y    | string  | string:hello         | 3     |
| 30  | x3 hello       | y3 hi,hello there   | y    | string  | string:x3 hello      | 0     |
| 30  | x3 hello       | y3 hi,hello there   | z    | string  | string:hello there   | 6     |
| 30  | x3 hello       | y3 hi,hello there   | z    | string  | string:here          | 13    |
| 30  | x3 hello       | y3 hi,hello there   | z    | string  | string:hi,hello there| 3     |
| 40  | .*+?\[]^$|     | hello sam           | z    | string  | string:hello sam     | 0     |
+-----+----------------+---------------------+------+---------+----------------------+-------+
expression has 8 tuples
```

Figure 3.1.6 Example 3

Example 4: Search anything that has an "h", followed by zero or more characters and ends with "o" in R.

```
>pr grep(Attr,Type,Val,Pos) "h.*o$" in R;
+-----+---------------------+---------------------+------+---------+---------------+-------+
| x   | y                   | z                   | Attr | Type    | Val           | Pos   |
+-----+---------------------+---------------------+------+---------+---------------+-------+
| 30  | x3 hello            | y3 hi,hello there   | y    | string  | string:hello  | 3     |
+-----+---------------------+---------------------+------+---------+---------------+-------+
expression has 2 tuple
```

Figure 3.1.7 Example 4

Example 5: Search anything that starts with an "x", followed by zero or more characters and ends with a "2" in R.

```
>pr grep(Attr,Type,Pos,Val) "^x.*2$" in R;
+-----+---------------------+---------------------+------+---------+-------+---------------+
| x   | y                   | z                   | Attr | Type    | Pos   | Val           |
+-----+---------------------+---------------------+------+---------+-------+---------------+
| 20  | x2                  | y2                  | y    | string  | 0     | string:x2     |
+-----+---------------------+---------------------+------+---------+-------+---------------+
expression has 1 tuple
```

Figure 3.1.8 Example 5

Example 6: Search anything that has an "h" and is followed by zero or one "e" in R.

```
>pr grep(Attr,Type,Pos,Val) "he?" in R;
+--------+----------------------+----------------------+------+--------+-----+----------------+
| x      | y                    | z                    | Attr | Type   | Pos | Val            |
+--------+----------------------+----------------------+------+--------+-----+----------------+
| 30     | x3 hello             | y3 hi,hello there    | y    | string | 3   | string:he      |
| 30     | x3 hello             | y3 hi,hello there    | z    | string | 3   | string:h       |
| 30     | x3 hello             | y3 hi,hello there    | z    | string | 6   | string:he      |
| 30     | x3 hello             | y3 hi,hello there    | z    | string | 13  | string:he      |
| 40     | .*+?\[]^$|           | hello sam            | z    | string | 0   | string:he      |
+--------+----------------------+----------------------+------+--------+-----+----------------+
expression has 5 tuples
```

Figure 3.1.9 Example 6

Example 7: Search anything that has an "e" or "r" and is followed by zero or one character in R.

```
pr grep(Attr,Type,Pos,Val) "[er].?" in R;
+------+----------------------+----------------------+-----+--------+-----+----------------------+
| x    | y                    | z                    | Attr| Type   | Pos | Val                  |
+------+----------------------+----------------------+-----+--------+-----+----------------------+
| 30   | x3 hello             | y3 hi,hello there    | y   | string | 4   | string:el            |
| 30   | x3 hello             | y3 hi,hello there    | z   | string | 7   | string:el            |
| 30   | x3 hello             | y3 hi,hello there    | z   | string | 14  | string:er            |
| 30   | x3 hello             | y3 hi,hello there    | z   | string | 15  | string:re            |
| 30   | x3 hello             | y3 hi,hello there    | z   | string | 16  | string:e             |
| 40   | .*+?\[]^$|           | hello sam            | z   | string | 1   | string:el            |
+------+----------------------+----------------------+-----+--------+-----+----------------------+
expression has 6 tuples
```

Figure 3.1.10 Example 7

Example 8: Search anything that has an "e" or "r" and is followed by zero or one character and ends with an "e" in R.

```
pr grep(Attr,Type,Pos,Val) "[er].?e$" in R;
+------+----------------------+----------------------+-----+--------+-----+----------------------+
| x    | y                    | z                    | Attr| Type   | Pos | Val                  |
+------+----------------------+----------------------+-----+--------+-----+----------------------+
| 30   | x3 hello             | y3 hi,hello there    | z   | string | 14  | string:ere           |
| 30   | x3 hello             | y3 hi,hello there    | z   | string | 15  | string:re            |
+------+----------------------+----------------------+-----+--------+-----+----------------------+
expression has 2 tuples
```

Figure 3.1.11 Example 8

Example 9: Search anything that has an "e" or "r" and is followed by zero or one character and it reaches the end of the attribute in R.

```
pr grep(Attr,Type,Pos,Val) "[er].?$" in R;
+------+----------------------+----------------------+-----+--------+-----+----------------------+
| x    | y                    | z                    | Attr| Type   | Pos | Val                  |
+------+----------------------+----------------------+-----+--------+-----+----------------------+
| 30   | x3 hello             | y3 hi,hello there    | z   | string | 15  | string:re            |
| 30   | x3 hello             | y3 hi,hello there    | z   | string | 16  | string:e             |
+------+----------------------+----------------------+-----+--------+-----+----------------------+
expression has 2 tuples
```

Figure 3.1.12 Example 9

Example 10: Search anything that has an "e" or "h" and has zero or one character before it in R.

```
pr grep(Attr,Type,Pos,Val) ".?[eh]" in R;
+-------------+----------------------+-----------+------+--------+-----+----------------------+
| x           | y                    | z         | Attr | Type   | Pos | Val                  |
+-------------+----------------------+-----------+------+--------+-----+----------------------+
| 40          | .*+?\[]^$|           | hello sam | z    | string | 0   | string:h             |
| 40          | .*+?\[]^$|           | hello sam | z    | string | 0   | string:he            |
+-------------+----------------------+-----------+------+--------+-----+----------------------+
expression has 2 tuples
```

Figure 3.1.13 Example 10

Example 11: Search anything that has an "h" and is followed by one or more characters in R.

```
>pr grep(Attr,Type,Pos,Val) "h.+" in R;
+-------------+------------+----------------------+------+--------+-----+----------------------+
| x           | y          | z                    | Attr | Type   | Pos | Val                  |
+-------------+------------+----------------------+------+--------+-----+----------------------+
| 30          | x3 hello   | y3 hi,hello there    | y    | string | 3   | string:hello         |
| 30          | x3 hello   | y3 hi,hello there    | z    | string | 3   | string:hi,hello there|
| 30          | x3 hello   | y3 hi,hello there    | z    | string | 6   | string:hello there   |
| 30          | x3 hello   | y3 hi,hello there    | z    | string | 13  | string:here          |
| 40          | .*+?\[]^$| | hello sam            | z    | string | 0   | string:hello sam     |
+-------------+------------+----------------------+------+--------+-----+----------------------+
expression has 5 tuples
```

Figure 3.1.14 Example 11

Example 12: Search anything that has an "h" and has at least one character before "h" in R.

```
pr grep(Attr,Type,Pos,Val) ".+h" in R;
+-------------+----------+----------------------+------+--------+-----+----------------------+
| x           | y        | z                    | Attr | Type   | Pos | Val                  |
+-------------+----------+----------------------+------+--------+-----+----------------------+
| 30          | x3 hello | y3 hi,hello there    | y    | string | 0   | string:x3 h          |
| 30          | x3 hello | y3 hi,hello there    | z    | string | 0   | string:y3 h          |
| 30          | x3 hello | y3 hi,hello there    | z    | string | 0   | string:y3 hi,h       |
| 30          | x3 hello | y3 hi,hello there    | z    | string | 0   | string:y3 hi,hello th|
+-------------+----------+----------------------+------+--------+-----+----------------------+
expression has 4 tuples
```

Figure 3.1.15 Example 12

Example 13: Search anything that has an "h", followed by zero or more characters and then an "o" and has at least one character after this "o" in R.

```
>pr grep(Attr,Type,Pos,Val) "h.*o.+" in R;
+-------------+------------+----------------------+------+-------+----+----------------------+
| x           | y          | z                    | Attr | Type  | Pos| Val                  |
+-------------+------------+----------------------+------+-------+----+----------------------+
| 30          | x3 hello   | y3 hi,hello there    | z    | string| 3  | string:hi,hello there|
| 30          | x3 hello   | y3 hi,hello there    | z    | string| 6  | string:hello there   |
| 40          | .*+?\[]^$| | hello sam            | z    | string| 0  | string:hello sam     |
+-------------+------------+----------------------+------+-------+----+----------------------+
expression has 3 tuples
```

Figure 3.1.16 Example 13

Example 14: Search anything that has an "h" and is followed by zero or more "e"s and then

followed by an "i" or a "l" in R.

```
>pr grep(Attr,Type,Pos,Val) "he*[il]" in R;
+-------------+-------------+----------------------+-----+--------+----+----------------------+
| x           | y           | z                    | Attr| Type   | Pos| Val                  |
+-------------+-------------+----------------------+-----+--------+----+----------------------+
| 30          | x3 hello    | y3 hi,hello there    | y   | string | 3  | string:hel           |
| 30          | x3 hello    | y3 hi,hello there    | z   | string | 3  | string:hi            |
| 30          | x3 hello    | y3 hi,hello there    | z   | string | 6  | string:hel           |
| 40          | .*+?\[]^$|  | hello sam            | z   | string | 0  | string:hel           |
+-------------+-------------+----------------------+-----+--------+----+----------------------+
expression has 4 tuples
```

Figure 3.1.17 Example 14

Example 15: Search anything that has an "h" and is followed by one or more "e"s and then followed by a "r" or a "l" in R.

```
>pr grep(Attr,Type,Pos,Val) "he+[rl]" in R;
+-------------+------------+----------------------+------+-------+----+----------------------+
| x           | y          | z                    | Attr | Type  | Pos| Val                  |
+-------------+------------+----------------------+------+-------+----+----------------------+
| 30          | x3 hello   | y3 hi,hello there    | y    | string| 3  | string:hel           |
| 30          | x3 hello   | y3 hi,hello there    | z    | string| 6  | string:hel           |
| 30          | x3 hello   | y3 hi,hello there    | z    | string| 13 | string:her           |
| 40          | .*+?\[]^$| | hello sam            | z    | string| 0  | string:hel           |
+-------------+------------+----------------------+------+-------+----+----------------------+
expression has 4 tuples
```

Figure 3.1.18 Example 15

Example 16: Search anything that has an "he" and is followed by zero or more "l"s and then followed by an "o" and then zero of more any characters and an "e" or a "m" in R.

```
>pr grep(Attr,Type,Pos,Val) "hel*o.*[em]" in R;
+-------------+-------------+----------------------+------+-------+----+----------------------+
| x           | y           | z                    | Attr | Type  | Pos| Val                  |
+-------------+-------------+----------------------+------+-------+----+----------------------+
| 30          | x3 hello    | y3 hi,hello there    | z    | string| 6  | string:hello the     |
| 30          | x3 hello    | y3 hi,hello there    | z    | string| 6  | string:hello there   |
| 40          | .*+?\[]^$|  | hello sam            | z    | string| 0  | string:hello sam     |
+-------------+-------------+----------------------+------+-------+----+----------------------+
expression has 3 tuples
```

Figure 3.1.19 Example 16

Example 17: Search anything that has an "*" and is followed by zero or more characters and then followed by a "^" in R. Please note that some characters such as * and ^ in the following example need a "\" to escape their original meaning. Other characters which need a "\" to escape its meaning include ., +, ?, \, [,], $, and |.

```
>pr grep(Attr,Type,Val,Pos) "\*.*\^" in R;
+-------------+-------------+----------+------+--------+----------------------+-------------+
| x           | y           | z        | Attr | Type   | Val                  | Pos         |
+-------------+-------------+----------+------+--------+----------------------+-------------+
| 40          | .*+?\[]^$|  | hello    | y    | string | string:*+?\[]^       | 1           |
+-------------+-------------+----------+------+--------+----------------------+-------------+
expression has 1 tuple
```

Figure 3.1.20 Example 17

Example 18: Search anything that has "he" and is followed by zero or one "l" and then followed by an "o" in R. So this will find heo, helo, since we don't have these in R, it will return an empty relation.

```
>pr grep(Attr,Type,Pos,Val) "hel?o" in R;
+-------------+---------------+----------------------+------+-----+----+----------------------+
| x           | y             | z                    | Attr | Type| Pos| Val                  |
+-------------+---------------+----------------------+------+-----+----+----------------------+
+-------------+---------------+----------------------+------+-----+----+----------------------+
expression has 0 tuple
```

Figure 3.1.21 Example 18

Example 19: Search anything that has an "h" and is followed by zero or more characters and then followed by an "e" in R.

```
>pr grep(Attr,Type,Pos,Val) "h.*e" in R;
+-------------+-----------+----------------------+-----+-------+----+----------------------+
| x           | y         | z                    | Attr| Type  | Pos| Val                  |
+-------------+-----------+----------------------+-----+-------+----+----------------------+
| 30          | x3 hello  | y3 hi,hello there    | y   | string| 3  | string:he            |
| 30          | x3 hello  | y3 hi,hello there    | z   | string| 3  | string:hi,he         |
| 30          | x3 hello  | y3 hi,hello there    | z   | string| 3  | string:hi,hello the  |
| 30          | x3 hello  | y3 hi,hello there    | z   | string| 3  | string:hi,hello there|
| 30          | x3 hello  | y3 hi,hello there    | z   | string| 6  | string:he            |
| 30          | x3 hello  | y3 hi,hello there    | z   | string| 6  | string:hello the     |
| 30          | x3 hello  | y3 hi,hello there    | z   | string| 6  | string:hello there   |
| 30          | x3 hello  | y3 hi,hello there    | z   | string| 13 | string:he            |
| 30          | x3 hello  | y3 hi,hello there    | z   | string| 13 | string:here          |
| 40          | .*+?\[]^$|| hello sam            | z   | string| 0  | string:he            |
+-------------+-----------+----------------------+-----+-------+----+----------------------+
expression has 10 tuples
```

Figure 3.1.22 Example 19

Example 20: Since *****, + and ? specify how many times a pattern occurs, it makes no sense to put them at the very beginning of a pattern. So we do not allow this kind of situation, for example:

```
>pr grep(Attr,Type,Pos,Val) "*m" in R;
invalid syntax of pattern:* cannot be at the beginning of a pattern
```

Figure 3.1.23 Example 20

3.1.4 Parameter List 2 of Grep

In parameter List 2, we can specify several parameters of type string. They can be used to output parts of the string that matches the pattern. Given following example, in the "before" attribute of the first tuple, there is a blank after x3. For this tuple, the "end" is an empty string. For the second tuple, in the "end" attribute, there is a blank before "there". For the third tuple, there is a empty string in the "before" attribute, and a blank before "sam" in the "end" attribute. In order to show this clearly, we attach a "B" and an "E" at the beginning of the string and at the end of the string, respectively.

```
>pr grep(;before,end) "\beforehello\end" in R;
+-------------+----------------------+----------------------+----------------------+----------+
| x           | y                    | z                    | before               | end      |
+-------------+----------------------+----------------------+----------------------+----------+
| 30          | x3 hello             | y3 hi,hello there    | x3                   |          |
| 30          | x3 hello             | y3 hi,hello there    | y3 hi,               |  there   |
| 40          | .*+?\[]^$|           | hello sam            |                      |  sam     |
+-------------+----------------------+----------------------+----------------------+----------+
expression has 3 tuples
let b be "B";
let e be "E";
let bb be b cat before cat e;
let eb be b cat end cat e;
>pr [x,y,z,bb,eb] in grep(;before,end) "\beforehello\end" in R;
+-------+----------------------+----------------------+----------------------+----------------------+
| x     | y                    | z                    | bb                   | eb                   |
+-------+----------------------+----------------------+----------------------+----------------------+
| 30    | x3 hello             | y3 hi,hello there    | Bx3 E                | BE                   |
| 30    | x3 hello             | y3 hi,hello there    | By3 hi,E             | B thereE             |
| 40    | .*+?\[]^$|           | hello sam            | BE                   | B samE               |
+-------+----------------------+----------------------+----------------------+----------------------+
```

Figure 3.1.24 An example of grep with only parameter List 2

We can also use both the parameter List 1 and parameter List 2 as the following example. This example has the same as the former one with blank and empty string.

```
>domain before string;
>domain end string;
>pr grep(Attr,Type,Pos,Val;before,end) "\beforehello\end" in R;
+------+-----------+----------------------+-----+--------+-----+--------------------------+---------+--------+
| x    | y         | z                    | Attr| Type   | Pos | Val                      | before  | end    |
+------+-----------+----------------------+-----+--------+-----+--------------------------+---------+--------+
| 30   | x3 hello  | y3 hi,hello there    | y   | string | 0   | string:x3 hello          | x3      |        |
| 30   | x3 hello  | y3 hi,hello there    | z   | string | 0   | string:y3 hi,hello there | y3 hi,  |  there |
| 40   | .*+?\[]^$|| hello sam            | z   | string | 0   | string:hello             |         |  sam   |
+------+-----------+----------------------+-----+--------+-----+--------------------------+---------+--------+
expression has 3 tuples
>pr [x,y,z,Attr,Type,Pos,Val,bb,eb] in grep(Attr,Type,Pos,Val;before,end) "\beforehello\end" in R;
+----+-------------+----------------------+-----+--------+----+----------------------+-----------+------------+
| x  | y           | z                    | Attr| Type   | Pos| Val                  | bb        | eb         |
+----+-------------+----------------------+-----+--------+----+----------------------+-----------+------------+
| 30 | x3 hello    | y3 hi,hello there    | y   | string | 0  | string:x3 hello      | Bx3 E     | BE         |
| 30 | x3 hello    | y3 hi,hello there    | z   | string | 0  | string:y3 hi,hello t | By3 hi,E  | B thereE   |
| 40 | .*+?\[]^$|  | hello sam            | z   | string | 0  | string:hello sam     | BE        | B samE     |
+----+-------------+----------------------+-----+--------+----+----------------------+-----------+------------+
expression has 3 tuples
```

Figure 3.1.25 An example of grep with both parameter lists

Furthermore, we can specify the parameters in between the pattern as in the following example. Here p2 represents anything in between "he" and "l". For example, in the first tuple, we have Val as "hel", hence there is nothing in between "he" and "l", and p2 is an empty string. We have the same situation with third and fifth tuple. For the second, fourth and sixth tuple, we have Val as "hell", there is an "l" in between "he" and "l", so p2 is l for these tuples.

```
>domain p2 string;
>pr grep(Attr,Type,Pos,Val;p2) "he\p2l" in R;
+-------------+-----------+----------------------+---------+--------+------+----------------------+-----+
| x           | y         | z                    | Attr    | Type   | Pos  | Val                  | p2  |
+-------------+-----------+----------------------+---------+--------+------+----------------------+-----+
| 30          | x3 hello  | y3 hi,hello there    | y       | string | 3    | string:hel           |     |
| 30          | x3 hello  | y3 hi,hello there    | y       | string | 3    | string:hell          | l   |
| 30          | x3 hello  | y3 hi,hello there    | z       | string | 6    | string:hel           |     |
| 30          | x3 hello  | y3 hi,hello there    | z       | string | 6    | string:hell          | l   |
| 40          | .*+?\[]^$|| hello sam          | z       | string | 0    | string:hel           |     |
| 40          | .*+?\[]^$|| hello sam          | z       | string | 0    | string:hell          | l   |
+-------------+-----------+----------------------+---------+--------+------+----------------------+-----+
expression has 6 tuples
```

Figure 3.1.26 An example of grep with parameter in between the pattern

Examples with more parameters:

```
>pr grep(Attr,Type,Pos,Val;before,after,end) "\beforeh.\aftero\end" in R;
+---+-----------+----------------------+-----+--------+---+-------------------------+--------+-------+---------+
| x | y         | z                    | Attr| Type   | Pos| Val                    | before | after | end     |
+---+-----------+----------------------+-----+--------+---+-------------------------+--------+-------+---------+
| 30| x3 hello  | y3 hi,hello there    | y   | string | 0  | string:x3 hello         | x3     | ll    |         |
| 30| x3 hello  | y3 hi,hello there    | z   | string | 0  | string:y3 hi,hello there | y3     | ,hell |  there  |
| 30| x3 hello  | y3 hi,hello there    | z   | string | 0  | string:y3 hi,hello there | y3 hi, | ll    |  there  |
| 40| .*+?\[]^$|| hello sam            | z   | string | 0  | string:hello sam        |        | ll    |  sam    |
+---+-----------+----------------------+-----+--------+---+-------------------------+--------+-------+---------+
expression has 4 tuples


>pr grep(Attr,Type,Pos,Val;before,after,end) "\beforeh.\afterl\end" in R;
+---+-----------+----------------------+------+-------+-----+-------------------------+--------+-------+---------+
| x | y         | z                    | Attr | Type  | Pos | Val                     | before | after | end     |
+---+-----------+----------------------+------+-------+-----+-------------------------+--------+-------+---------+
| 30| x3 hello  | y3 hi,hello there    | y    | string| 0   | string:x3 hello         | x3     |       | lo      |
| 30| x3 hello  | y3 hi,hello there    | y    | string| 0   | string:x3 hello         | x3     | l     | o       |
| 30| x3 hello  | y3 hi,hello there    | z    | string| 0   | string:y3 hi,hello there | y3     | ,he   | lo there|
| 30| x3 hello  | y3 hi,hello there    | z    | string| 0   | string:y3 hi,hello there | y3     | ,hel  | o there |
| 30| x3 hello  | y3 hi,hello there    | z    | string| 0   | string:y3 hi,hello there | y3 hi, |       | lo there|
| 30| x3 hello  | y3 hi,hello there    | z    | string| 0   | string:y3 hi,hello there | y3 hi, | l     | o there |
| 40| .*+?\[]^$|| hello sam            | z    | string| 0   | string:hello sam        |        |       | lo sam  |
| 40| .*+?\[]^$|| hello sam            | z    | string| 0   | string:hello sam        |        | l     | o sam   |
+---+-----------+----------------------+------+-------+-----+-------------------------+--------+-------+---------+
expression has 6 tuples

>domain p1,p2 strg;
>pr grep(Attr,Type,Pos,Val;before,p1,p2,end) "\before3\p1h.\p2l\end" in R;
+---+-----------+----------------------+------+-------+-----+-------------------------+--------+-----+-----+---------+
| x | y         | z                    | Attr | Type  | Pos | Val                     | before | p1  | p2  | end     |
+---+-----------+----------------------+------+-------+-----+-------------------------+--------+-----+-----+---------+
| 30| x3 hello  | y3 hi,hello there    | y    | string| 0   | string:x3 hello         | x      |     |     | lo      |
| 30| x3 hello  | y3 hi,hello there    | y    | string| 0   | string:x3 hello         | x      |     | l   | o       |
| 30| x3 hello  | y3 hi,hello there    | z    | string| 0   | string:y3 hi,hello there | y      |     | ,he | lo there|
| 30| x3 hello  | y3 hi,hello there    | z    | string| 0   | string:y3 hi,hello there | y      |     | ,hel| o there |
| 30| x3 hello  | y3 hi,hello there    | z    | string| 0   | string:y3 hi,hello there | y      |  hi,|     | lo there|
| 30| x3 hello  | y3 hi,hello there    | z    | string| 0   | string:y3 hi,hello there | y      |  hi,| l   | o there |
+---+-----------+----------------------+------+-------+-----+-------------------------+--------+-----+-----+---------+
expression has 6 tuples
```

Figure 3.1.27 An example of grep with more parameters

3.1.5 Using relation or top level scalar as a pattern

The grep command can take a relation or a top level scalar as the pattern. Using a top level scalar as a pattern is straightforward, as it can represent a search string. The meaning of using a relation as the search pattern is to take the advantage of the set property of a relation to represent multiple search strings.

For example, relation g9 has two tuples, so the pattern is basically the union of "l" and "h".

```
>relation g9(z)<-{("1"),("h")};
>pr g9;
+----------------------+
| z                    |
+----------------------+
| 1                    |
| h                    |
+----------------------+
relation g9 has 2 tuples

>pr grep(Attr,Type,Pos,Val) g9 in R;
+-------------+----------------------+----------------------+------+---------+-------+---------------+
| x           | y                    | z                    | Attr | Type    | Pos   | Val           |
+-------------+----------------------+----------------------+------+---------+-------+---------------+
| 10          | x1                   | y1                   | x    | integer | 0     | integer:1     |
| 10          | x1                   | y1                   | y    | string  | 1     | string:1      |
| 10          | x1                   | y1                   | z    | string  | 1     | string:1      |
| 30          | x3 hello             | y3 hi,hello there    | y    | string  | 3     | string:h      |
| 30          | x3 hello             | y3 hi,hello there    | z    | string  | 3     | string:h      |
| 30          | x3 hello             | y3 hi,hello there    | z    | string  | 6     | string:h      |
| 30          | x3 hello             | y3 hi,hello there    | z    | string  | 13    | string:h      |
| 40          | .*+?\[]^$|           | hello sam            | z    | string  | 0     | string:h      |
+-------------+----------------------+----------------------+------+---------+-------+---------------+
expression has 8 tuples

>declare s1 string<-"hello";
>pr grep(Attr,Type,Pos,Val) s1 in R;
+-------------+----------------------+----------------------+------+-------+---+----------------------+
| x           | y                    | z                    | Attr | Type  |Pos| Val                  |
+-------------+----------------------+----------------------+------+-------+---+----------------------+
| 30          | x3 hello             | y3 hi,hello there    | y    | string| 3 | string:hello         |
| 30          | x3 hello             | y3 hi,hello there    | z    | string| 6 | string:hello         |
| 40          | .*+?\[]^$|           | hello sam            | z    | string| 0 | string:hello         |
+-------------+----------------------+----------------------+------+-------+---+----------------------+
expression has 3 tuples
```

Figure 3.1.28 An example of grep using top level scalar as pattern

3.2 Union Types

In traditional relational database approach, types are always fixed prior to populating the database. Once the data is populated, its binary storage cannot be interpreted without having knowledge of the schema. With semi-structured data we may specify the type after the database has been populated. This type may only partially describe the structure and often does so imprecisely. An important consequence is that the data instance may have more than one type [ABS00]. In JRelix, we will use union types to handle this situation.

3.2.1 Definition of Union Types

For example, we define c2 to have type of integer or string; we use "|" to separate each type. We can also define an attribute to have a nested relation or other primitive types. In the example shown below, we define n1 to be a nested relation on a and b, then we define c1 to have type n1 or string. We can define relation R2 on a and c2.

```
>domain c2 integer|string;
>domain a integer;
>domain b string;
>domain n1(a,b);
>domain c1 n1|string;
>relation R2(a,c2);
>sd;
------------------------------ Domain Entry -------------------------------
Name         Type            NumRef     IsState   Dom_List
---------------------------------------------------------------------------
n1           idlist          1          false     .id, a, b,
c2           union           1          false     integer|string
c1           union           0          false     n1|string
b            string          1          false
a            integer         2          false
---------------------------------------------------------------------------
```

Figure 3.2.1 Examples of definition of union types

3.2.2 Initialization of Relations with Union Types

When initializing a relation with union types, you do not need to specify which type you are using. The system will automatically check the definition of the union type to find a matched type according to the data input. For example, we want to initialize R2, since c2 can be a integer or a string, we can do the following:

```
>relation R2(a,c2)<-{(1,1),(2,"1")};
>pr R2;
+-------------+----------------------+
| a           | c2                   |
+-------------+----------------------+
| 1           | integer:1            |
| 2           | string:1             |
+-------------+----------------------+
relation R2 has 2 tuples
```

If we define c3 as of type a or b, then when initializing, the system will automatically find its base type and store it. Only scalar types will be resolved in this way. For union types which have nested relation attributes, we need to store the nested attributes information.

```
domain c3 a|b;
relation R3(a,c3)<-{(1,1),(3,"b")};
>pr R3;
+-------------+----------------------+
| a           | c3                   |
+-------------+----------------------+
| 1           | integer:1            |
| 3           | string:b             |
+-------------+----------------------+
relation R3 has 2 tuples
```

Figure 3.2.3 Example 2 of initialization of union types

The way you initialize the relation with nested attributes as one of its union types is similar to the way you initialize a nested relation. For example:

```
relation Tc(a,c1)<-{(1,{(1,"1"),(2,"2")}),(2,{(2,"2")})};
pr Tc;
+-------------+----------------------+
| a           | c1                   |
+-------------+----------------------+
| 1           | n1:1                 |
| 2           | n1:2                 |
+-------------+----------------------+
relation Tc has 2 tuples
>pr .n1;
+----------------------+-------------+----------------------+
| .id                  | a           | b                    |
+----------------------+-------------+----------------------+
| 1                    | 1           | 1                    |
| 1                    | 2           | 2                    |
| 2                    | 2           | 2                    |
+----------------------+-------------+----------------------+
relation .n1 has 3 tuples
```

Figure 3.2.4 Example 1 of initialization of union types with nested relation

In this example, c1 can be a string or a nested relation n1(a,b). Since in the initialization statement the system sees "{" at the position for c1, it determines that this must be a nested relation. If the data are a type of n1, the system will store this n1 information in each cell along with the surrogate key (see 2.6 for detail). For example: n1:1, and 1 is the surrogate key for the nested relation. We can use the command "pr .n1;" to see the data of n1;

The type of an attribute is not fixed for all tuples. It may vary according to the input. The following example shows a mixing type on c1. The first tuple has a nested relation n1 as its type, and the second tuple has a string as its type on c1. Please note that the surrogate is monotonely increasing for each nested relation.

```
relation Tc10(a,c1)<-{(1,{(1,"1"),(2,"2")}),(2,"2")};
pr Tc10;
+-------------+----------------------+
| a           | c1                   |
+-------------+----------------------+
| 1           | n1:3                 |
| 2           | string:2             |
+-------------+----------------------+
relation Tc10 has 2 tuples
>pr .n1;
+---------------------+-------------+----------------------+
| .id                 | a           | b                    |
+---------------------+-------------+----------------------+
| 1                   | 1           | 1                    |
| 1                   | 2           | 2                    |
| 2                   | 2           | 2                    |
| 3                   | 1           | 1                    |
| 3                   | 2           | 2                    |
+---------------------+-------------+----------------------+
relation .n1 has 5 tuples
```

Figure 3.2.5 Example 2 of initialization of union types with nested relation

3.2.3 Querying Relations with Union Types

It is not necessary to specify the type when querying, the system will figure out what type your input query is and return the right result. For example, when you write c2=1, it means that this query is looking for the tuples which has c2 as type of integer and value of 1. However, if the query condition statement is c2="1", then the query will look for the tuples which has c2 as type of string and value of "1".

See the following result:

```
>pr where c2=1 in R2;
+-------------+----------------------+
| a           | c2                   |
+-------------+----------------------+
| 1           | integer:1            |
+-------------+----------------------+
expression has 1 tuple

>pr where c2="1" in R2;
+-------------+----------------------+
| a           | c2                   |
+-------------+----------------------+
| 2           | string:1             |
+-------------+----------------------+
expression has 1 tuple
```

Figure 3.2.6 Query of relations with union types

The joining of relations with union types is the same as the joining of relations without union types. In the following example, R2 and R3 will join the common attribute a and c2 from R2 and it will join c3 from R3. Since c2 and c3 are union types, when joined, the system will check on both types and values.

```
>pr R2 [a,c2 :ijoin: a,c3] R3;
+-------------+----------------------+----------------------+
| a           | c2                   | c3                   |
+-------------+----------------------+----------------------+
| 1           | integer:1            | integer:1            |
+-------------+----------------------+----------------------+
expression has 1 tuple
```

Figure 3.2.7 Join of relations with union types

A primitive type can join with union type also. In the following example, a from R2 will join with c3 from R31. c3 can have a type of integer or string, but only the tuple with type integer of attribute c3 and having the same value as a can be joined with R2.

```
R31<-[c3] in R3;
pr R2 [a:ijoin:c3] R31;
+-------------+----------------------+----------------------+
| a           | c2                   | c3                   |
+-------------+----------------------+----------------------+
| 1           | integer:1            | integer:1            |
+-------------+----------------------+----------------------+
expression has 1 tuple
```

Figure 3.2.8 Join of relations with union types

3.2.4 More examples with Union Types

Union types can be defined on other union types. In the following example, c4 is defined on union types c1 or b.

```
domain c4 c1|b;
relation Tc21(a,c4)<-{(1,{(1,"1"),(2,"2")})};
pr Tc21;
+-------------+----------------------+
| a           | c4                   |
+-------------+----------------------+
| 1           | c1:n1:5              |
+-------------+----------------------+
relation Tc21 has 1 tuple
>pr .n1;
+----------------------+-------------+----------------------+
| .id                  | a           | b                    |
+----------------------+-------------+----------------------+
| 5                    | 1           | 1                    |
| 5                    | 2           | 2                    |
//others omited
```

Figure 3.2.9 Union types defined on other union types

In the following example, the third tuple has an entry{} for the second attribute c4, the system will figure out that this is a nested relation, and since it is empty, we will see a 0 as its surrogate. It uses the same mechanism as when we initialize a relation with an empty set for a nested attribute.

```
relation Tc31(a,c4)<-{(1,{(1,"1"),(2,"2")}),(2,"hello"),(3,{})};
pr Tc31;
+-------------+----------------------+
| a           | c4                   |
+-------------+----------------------+
| 1           | c1:n1:7              |
| 2           | string:hello         |
| 3           | c1:n1:0              |
+-------------+----------------------+
relation Tc31 has 3 tuples
>pr .n1;
+----------------------+-------------+----------------------+
| .id                  | a           | b                    |
+----------------------+-------------+----------------------+
| 7                    | 1           | 1                    |
| 7                    | 2           | 2                    |
//others omitted
```

Figure 3.2.10 An example with empty nested relation

The following example shows duplicates when initializing a relation with union types. The duplicate will be eliminated since they have the same types and values.

```
>relation Tc31(a,c4)<-{(1,{(1,"1"),(2,"2")}),(1,{(1,"1"),(2,"2")})};//duplicate
eliminated
+-------------+----------------------+
| a           | c4                   |
+-------------+----------------------+
| 1           | c1:n1:9              |
+-------------+----------------------+
relation Tc31 has 1 tuple
>pr .n1;
+----------------------+-------------+----------------------+
| .id                  | a           | b                    |
+----------------------+-------------+----------------------+
| 9                    | 1           | 1                    |
| 9                    | 2           | 2                    |
//others omitted
```

Figure 3.2.11 Duplicate elimination with union types

Nested domain types can be defined on the union types, in the following example, n5 is defined on two union types c6 and b.

```
domain c6 integer|string|n1;
domain n5(c6,b);
relation Tc5(a,n5)<-
{(1,{(1,"b1"),("2","b2"),({(1,"first"),(2,"second")},"b3")})};
>pr Tc5;
+-------------+----------------------+
| a           | n5                   |
+-------------+----------------------+
| 1           | 10                   |
+-------------+----------------------+
relation Tc5 has 1 tuple
pr .n5;
+----------------------+----------------------+----------------------+
| .id                  | c6                   | b                    |
+----------------------+----------------------+----------------------+
| 10                   | integer:1            | b1                   |
| 10                   | n1:11                | b3                   |
| 10                   | string:2             | b2                   |
+----------------------+----------------------+----------------------+
relation .n5 has 3 tuples
pr .n1;
+----------------------+-------------+----------------------+
| .id                  | a           | b                    |
+----------------------+-------------+----------------------+
| 11                   | 1           | first                |
| 11                   | 2           | second               |
//others omitted
```

Figure 3.2.12 Nested domain defined onunion types

In the following example, c7 is defined as n1 or n5 or c1. The system will figure out which nested attribute to use according to the input data.

```
domain c7 n1|n5|c1;

relation Tc6(a,c7)<-{(1,{(1,"a"),("2","b"),({(1,"1")},"c")}),
(2,{(10,"a"),("20","b"),({(100,"hello")},"c")}),(3,{(1,"first")})};
pr Tc6;
+-------------+----------------------+
| a           | c7                   |
+-------------+----------------------+
| 1           | n5:1                 |
| 2           | n5:3                 |
| 3           | n1:5                 |
+-------------+----------------------+
relation Tc6 has 3 tuples
pr .n5;
+----------------------+----------------------+----------------------+
| .id                  | c6                   | b                    |
+----------------------+----------------------+----------------------+
| 1                    | integer:1            | a                    |
| 1                    | n1:2                 | c                    |
| 1                    | string:2             | b                    |
| 3                    | integer:10           | a                    |
| 3                    | n1:4                 | c                    |
| 3                    | string:20            | b                    |
+----------------------+----------------------+----------------------+
relation .n5 has 6 tuples

pr .n1;
+----------------------+-------------+----------------------+
| .id                  | a           | b                    |
+----------------------+-------------+----------------------+
| 2                    | 1           | 1                    |
| 4                    | 100         | hello                |
| 5                    | 1           | first                |
//others omitted
```

Figure 3.2.13 Union type defined on more than one nested domain

3.3 Top Level Scalar

3.3.1 Declaration and initialization of top level scalar

A top level scalar has the same scope as a relation. A top level scalar is defined as any primitive type and can be used anywhere like a virtual attribute.

The following example shows how we define a top level scalar. s1 is defined to be an integer. We can use the "sd" (show domain) and the "sr" (show relation) commands to see its definition. Since it has the same scope as a relation, we put it in a relation table with other relations and in the meanwhile, we store it in the domain table so that we can remember its definition by storing it as a tree.

```
>declare s1 integer;
>sd s1;
------------------------------ Domain Entry -------------------------------
Name        Type            NumRef     IsState   Dom_List
---------------------------------------------------------------------------
s1          integer         0          false

Literal:470:453:null:0

---------------------------------------------------------------------------
>sr s1;
------------------------------ Relation Entry -----------------------------
Name        Type          Arity       NTuples     Sort    Active
-------------------------------------------------------------------------------------
s1          scalar              0          0          0          0
-------------------------------------------------------------------------------------
```

Figure 3.3.1 top level scalar declaration

To initialize a top level scalar, we use it in a similar way as we initialize a relation. For example:

```
>s1<-1;
>pr s1;
+-------------+
| s1          |
+-------------+
| 1           |
+-------------+
relation s1 has 1 tuple
```

Figure 3.3.2 initialization of top level scalar

We can also initialize it when we define a top level scalar, like we do with a relation.

```
>declare s2 string<-"hello";
>pr [s1,s2] in R;
+-------------+----------------------+
| s1          | s2                   |
+-------------+----------------------+
| 1           | hello                |
+-------------+----------------------+
expression has 1 tuple
```

Figure 3.3.3 initialization and declaration of top level scalar

We can also define a top level scalar on other top level scalars. For example:

```
>declare s3 integer<-3;
>declare s4 integer;
>s4<-s1+s3;

>pr [s1,s2,s3,s4] in R;
+-------------+----------------------+-------------+-------------+
| s1          | s2                   | s3          | s4          |
+-------------+----------------------+-------------+-------------+
| 1           | hello                | 3           | 4           |
+-------------+----------------------+-------------+-------------+
expression has 1 tuple
```

Figure 3.3.4 declare a top level scalar on other top level scalars

3.3.2 Querying with top level scalar

- Examples with a flat relation.

```
>pr where x=s1*10 in R;
+-------------+----------------------+----------------------+
| x           | y                    | z                    |
+-------------+----------------------+----------------------+
| 10          | x1                   | y1                   |
+-------------+----------------------+----------------------+
expression has 1 tuple

>pr [s2,Attr,Pos,y,z] in grep (Attr,Pos) s2 in R;
+----------------------+------+-----+----------------------+----------------------+
| s2                   | Attr | Pos | y                    | z                    |
+----------------------+------+-----+----------------------+----------------------+
| hello                | y    | 3   | x3 hello             | y3 hi,hello there    |
| hello                | z    | 0   | .*+?\[]^$|           | hello sam            |
| hello                | z    | 6   | x3 hello             | y3 hi,hello there    |
+----------------------+------+-----+----------------------+----------------------+
expression has 3 tuples

>let sub1 be substr(z,s1);
>pr [z,sub1] in R;
+----------------------+----------------------+
| z                    | sub1                 |
+----------------------+----------------------+
| hello sam            | ello sam             |
| y1                   | 1                    |
| y2                   | 2                    |
| y3 hi,hello there    | 3 hi,hello there     |
+----------------------+----------------------+
expression has 4 tuples
```

Figure 3.3.5 using top level scalars in flat relations

- Examples with a nested relation.

We can use top level scalar to query the nested relation shown in Figure 2.6.1

```
>declare x integer;
>x<-red + of a in n1;
>pr [a,x] in Tc;
+-------------+-------------+
| a           | x           |
+-------------+-------------+
| 1           | 3           |
| 2           | 2           |
+-------------+-------------+
expression has 2 tuples
```

Figure 3.3.6 using top level scalars in nested relations

3.4 Substring function

3.4.1 Define a substring

We can use the substring function with a string type domain. The substring function can take 2 or 3 parameters. The first parameter is the domain identifier that the substring will work on; the second and third parameters have types of integers and are similar to parameters of the Java substr function. That is, the first integer indicates the start position of the substring and the second integer indicates the position after the end of the substring. So the length of the substring equals to the second integer minus the first integer. The second integer can be omitted just like Java does with substr function, which means that the substring will start at the position given by the first parameter and continue to the end of the string.

The syntax is the following:

```
<SUBSTRING> "(" <Identifier> "," <ILITERAL> ("," <ILITERAL>)? ")"
<SUBSTRING> : "substr" | "substring"
<ILITERAL>:
<INTEGER_LITERAL>    |   <IDENTIFIER>
```

Let's see some examples.

Case 1: both integer parameters are provided

```
>let sub1 be substr(z,0,1);
>pr [z,sub1] in R;
+----------------------+----------------------+
| z                    | sub1                 |
+----------------------+----------------------+
| hello sam            | h                    |
| y1                   | y                    |
| y2                   | y                    |
| y3 hi,hello there    | y                    |
+----------------------+----------------------+
expression has 4 tuples
```

Figure 3.4.1 substr with both integer parameters provided

Case 2: the second integer parameter is omitted. In this case, the second "," can also be omitted. The function will interpret this as get sub string start from the position specified by the first integer and get the rest of the string.

```
>let sub4 be substr(z,1,);    //or let sub4 be substr(z,1);
>pr [z,sub4] in R;
+----------------------+----------------------+
| z                    | sub4                 |
+----------------------+----------------------+
| hello sam            | ello sam             |
| y1                   | 1                    |
| y2                   | 2                    |
| y3 hi,hello there    | 3 hi,hello there     |
+----------------------+----------------------+
expression has 4 tuples
```

Figure 3.4.2 substr without the second integer parameter

Case 3: the first integer parameter is omitted. This case will be interpreted as start from position 0 of the string, get the sub string till the position before the one specified by the second integer parameter.

```
>let sub3 be substr(z,,1);
>pr [z,sub3] in R;
+----------------------+----------------------+
| z                    | sub1                 |
+----------------------+----------------------+
| hello sam            | h                    |
| y1                   | y                    |
| y2                   | y                    |
| y3 hi,hello there    | y                    |
+----------------------+----------------------+
expression has 4 tuples
```

Figure 3.4.3 substr without the first integer parameter

We can also use other integer type identifiers to define the integer parameter. For example we can use an integer top level scalar or an integer domain to specify the two integer parameters in a substring function.

```
>let start be 1;
>let end be start+1;
>let subs be substr(z,start,end);
>pr [z,subs] in R;
+----------------------+----------------------+
| z                    | subs                 |
+----------------------+----------------------+
| hello sam            | e                    |
| y1                   | 1                    |
| y2                   | 2                    |
| y3 hi,hello there    | 3                    |
+----------------------+----------------------+
expression has 4 tuples
```

Figure 3.4.4 substr using identifier to specify the two integer parameters

The substring function is useful when we perform an update on a relation. In the following example, we first find those tuples that have "hello" in them and store the result in g1. Then we select those where the attribute is "z" in g1 and store the result in g2. Now

we want to change "hello" to "aloha" in g2. We need Pos, it is the start position of "hello" in attribute z of g2. We define sub1 as a substring of z starting from 0 and ending just before "hello". We then define sub2 as a substring of z starting from Pos and ending at the end of z. We know that the substring in between sub1 and sub2 is "hello", now we only need to concatenate sub1 with "aloha" and then with sub2 so that we can change "hello" to "aloha" in z.

```
>pr R;
+-------------+----------------------+----------------------+
| x           | y                    | z                    |
+-------------+----------------------+----------------------+
| 10          | x1                   | y1                   |
| 20          | x2                   | y2                   |
| 30          | x3 hello             | y3 hi,hello there    |
| 40          | .*+?\[]^$|           | hello sam            |
+-------------+----------------------+----------------------+
expression has 4 tuples
>g1<-grep(Attr,Type,Pos,Val) "hello" in R;
>g2<-[x,y,z,Pos] where Attr=quote z in g1;
>pr g2;
+-------------+----------------------+----------------------+-------------+
| x           | y                    | z                    | Pos         |
+-------------+----------------------+----------------------+-------------+
| 30          | x3 hello             | y3 hi,hello there    | 6           |
| 40          | .*+?\[]^$|           | hello sam            | 0           |
+-------------+----------------------+----------------------+-------------+
relation g2 has 2 tuples

>let sub1 be substr(z,0,Pos);
>let Pos1 be Pos+5;
>let sub2 be substr(z,Pos1);
>update g2 change z<-sub1 cat "aloha" cat sub2;
>pr g2;
+-------------+----------------------+----------------------+-------------+
| x           | y                    | z                    | Pos         |
+-------------+----------------------+----------------------+-------------+
| 30          | x3 hello             | y3 hi,aloha there    | 6           |
| 40          | .*+?\[]^$|           | aloha sam            | 0           |
+-------------+----------------------+----------------------+-------------+
relation g2 has 2 tuples
```

Figure 3.4.5 perform update on relation using substring function

Chapter 4 Implementation

In this chapter, we will discuss the implementations of those new features that we introduced in Chapter 3. In section 4.1, we will give an overview of JRelix system implementation. Then we will discuss implementation of grep in section 4.2. In section 4.3 we will discuss how we implement union types in JRelix. In section 4.4 we will discuss the implementation of top level scalar and in section 4.5 we will show how we implement the substring function in JRelix.

4.1 JRelix system implementation overview

In this section, we will give an overview of JRelix system, including its system structure, how JRelix commands are recognized by the parser and how they are interpreted by the interpreter.

4.1.1 System structure

The end user types in a JRelix command from the user interface. The command will be parsed by the parser and the generated parse tree will be interpreted by the interpreter. The interpreter acts like the brain of a human. It performs the essential functionality of analyzing and interpreting the command and achieves the functionality by cooperating with other functions. The following figure 4.1.1 gives the system structure of JRelix. Function calls from interpreter are performed in Database Engine. It has all the functions of relational and domain algebra and extended functionality, such as nested relations, computation, distributed computing and text operations. Both system data and user defined data will be stored in the disk storage in the form of files, which can be obtained by the user as a printout from the user interface.

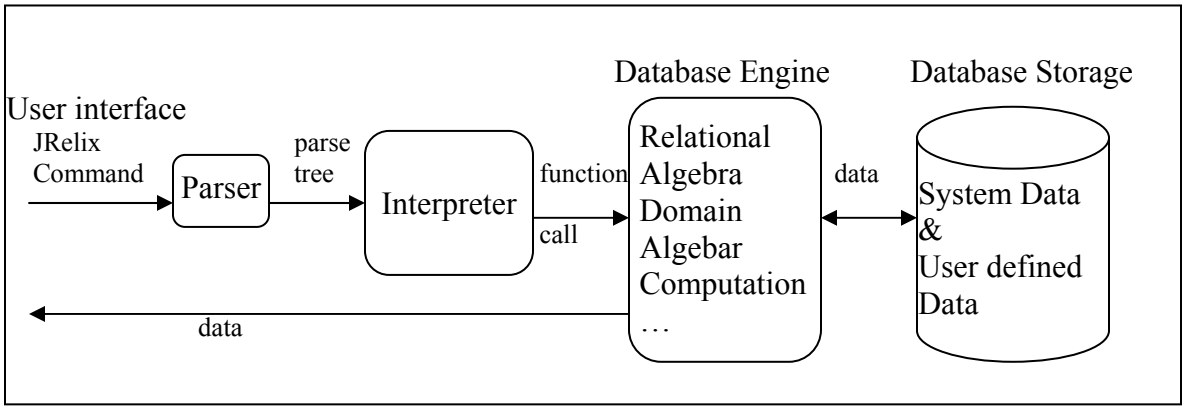


Figure 4.1.1 JRelix system structure

4.1.2 How the parser works (JJTree, JavaCC)

User input is parsed by the parser and it is important to define the syntax of the command in the parser properly so that the command can be recognized by the parser and parsed to the interpreter. Any incorrect syntax will be rejected by the parser. To build the parser in JRelix implementation, we use jjtree and javaCC. JavaCC is a parser generator, and jjtree is a preprocess for javaCC that is useful for building the parser tree. (Please find complete JavaCC document at https://javacc.dev.java.net/). The following is a simple example of how the parser parses the user input and builds a parse tree for the interpreter. We will explain this command in section 4.4. There are also other examples of syntax definition in section 4.2.1, 4.3.1, 4.4.1 and 4.5.1.

```
>declare s1 integer<-1;
```

In the parser, we define the syntax in the following way:

```
declare <IDList> <DType> [<TInitialization>]
<TInitialization> :<ASSIGN> <Literal>|<Expression>
```

Declare is a key word which will be recognized by the parser and it expects a list of identifiers after it. DType is a set of primitive types; it is also defined in the parser. This declaration can be followed by an initialization (<TInitialization>), which is also defined in the parser. Please find the detailed syntax in the Appendix. While the parser parses the input, each part of the command will be put into a parse tree in the predefined way that the interpreter expects. Figure 4.4.1 shows the parse tree of the above example. Any unexpected input will be thrown out as a syntax error by the parser.

4.1.3 How the interpreter and the actualizer works

The interpreter will take the root of the parse tree and read the information from the node to decide which action to take. The following figure shows the basic fields of a node. This information is entered at the parsing stage.

SimpleNode:

```
identifer:category name
opcode: operation code
type: operation type
name: node name(can be identifier name)
info: node info(can be literal value)
```

Take category Declaration as an example:

Operaton type of OP_DECLARATION has following kinds of operation code:

OP_RELATION	for relation declaration
OP_DECLARE	for top level scalar declaration
OP_VIEW	for view declaration
OP_DOMAIN	for domain declaration
OP_LET	for virtual domain declaration
OP_COMPUTATION	for computation declaration

The interpreter will call different functions based on the operation code. Field name and info are where the name of the identifier and the value of the literal are stored.

We have the Interpreter class in JRelix implementation, a major class of the interpreter. We have another very important class, Actualizer, which takes care of all the domain algebra, for example virtual domain actualization. For detailed document, please see [Yuan98] and [Hao98] for relational and domain algebra in JRelix.

4.2 Implementation of Grep

In our implementation, the syntax of the grep command will be different from UNIX egrep. We will add two parameter lists in the grep command and place parameters that are in parameter list 2 in the search string of grep. The major difference of our grep from Unix grep is to return the position and value of a match of a pattern. The major complication to achieve this is caused by the uncertainty of the length of the matching value of a pattern which has either '?', '*', or '+' in it. We will explain how we get the value in parameter lists without these wildcard characters in section 4.2.2 and we will deal with these wildcard in section 4.2.3. In section 4.2.4, we will explain how we deal with the second parameter list by using the method introduced in section 4.2.2 and 4.2.3.

We will start by introducing the syntax of grep and conclude in section 4.2.5 by giving some special cases of grep implementation.

4.2.1 syntax

In grep command, we can specify up to 2 parameter lists: search pattern and a relation expression. Figure 3.1.6 shows an example with both parameter lists. The following is the syntax of grep. Please refer to appendix for syntax of Selection and Identifier.

<GREP> ("("<GIDList>")")? (<Literal>|<Identifier>) <IN> <Selection>
<GIDList> : <LIDList>(";" <IDList>)?
<LIDList> :(<Identifier> ("," <Identifier>)*)?
<IDList> : <Identifier> ("," <Identifier>)*

4.2.2 Implementation for parameter list 1 without a wildcard in the pattern

Once the command is parsed by the parser, the parse tree is generated and parsed to the function evaluateGrep. This function will call other functions based on the number of parameter lists in the grep command. As shown in figure 3.1.1, the simplest case of grep, there is no parameter list in the grep command. We just need to call the UNIX egrep to see whether or not a particular pattern match exists in a string. We will give our algorithm in section 4.2.5.

Now, let's take a look at the parameter list 1 and to see how we can implement it to get the result. In figure 3.1.2, it shows an example with parameter list 1. Looking at the result table, we can think of a straightforward way to get the value of these parameters. We just

need to take each attribute in each tuple to see whether or not there are matches of a pattern and to output for each match the attribute name and the type where it is found and the position and the value of the match.

There could be multiple matches of the pattern in one string. We create the Grep class to save the position and value for each match. Figure 4.2.1 is the Grep class. It has three fields, the first two are straightforward, they are the value and position of the match found. The third one is string arrry paraValue, it stores all the user defined parameter values in the match string. We will need to use this array in the implementation for parameter list 2. In addition to these fields, The Grep also has two constructors. The first constructor will be used in the case where there is no parameter list 2; the second one will be used for the parameter list 2. We will discuss the parameter list 2 in section 4.2.4.

Class Grep:

```
public class Grep
{
        public String value;          //val
        public int position;          //pos
        public String [] paraValue;   //store values for parameter list 2

        public Grep(String v, int p)
        {
         value = v;
         position = p;
        }

        public Grep(String v,int p, String [] pv)
        {
         value = v;
         position = p;
         paraValue = new String[pv.length];
         for (int i= 0; i<pv.length; i++)
         {
          if (pv[i]!=null)
          paraValue[i] = new String(pv[i]);
         }
        }
}
```

	position Value	position ValueS	position ValueP
value	√	√	√
position	√	√	√
paraValue			√

Class fields are used in functions with "√"

Figure 4.2.1 class Grep

Since the essential part of the implementation is to find the position and value of a match of a pattern in a string, we will focus on the function **positionValue** in this section. Before we introduce the implementation of this function, let's first take a look at the main function of the grep commnad: **evaluateGrep(**in class Interpreter that calls **positionValue)** of finding matches of a pattern in a relation. We will introduce two other related functions **positionValueS** for pattern that has wild card "*", "+" or "?" and **positionValueP** for parameter list 2 repectively in section 4.2.3 and section 4.2.4.

The function **positionValue** will return an array of Grep object which will contain the position and value pairs that find in a string. The function **evaluateGrep** will call it for each matching cell in the relation. For the whole relation, we deal with each column one by one. For each column, we will call the UNIX egrep to get the cells that contain the match. For each such cell, we call the function **positionValue** to get the position and value pairs. Then we can construct a relation that has the value of that cell in that column and multiple pairs of position and value. Finally, we unite all these relations together to get the result relation. Let's illustrate the idea from the same example that we will use for the function **positionValue**.

```
grep(Attr,Type,Pos,Val) "h." in R;
```

Figure 4.2.2 shows the result relation that we constructed for column z and cell "y3 hi,hello there". We do this also for cell "hello" of z and cell "x3 hello" of x. The final result relation is the one that unites all of these relations. Please find the whole result relation of this example in Figure 3.1.4

```
>pr R;
+-------------+----------------------+----------------------+
| x           | y                    | z                    |
+-------------+----------------------+----------------------+
| 10          | x1                   | y1                   |
| 20          | x2                   | y2                   |
| 30          | x3 hello             | y3 hi,hello there    |
| 40          | .*+?\[]^$|           | hello                |
+-------------+----------------------+----------------------+


build result relation for cell 'y3 hi,hello there'
+----+----------------------+----------------------+------+--------+-----+----------------------+
| x  | y                    | z                    | Attr | Type   | Pos | Val                  |
+----+----------------------+----------------------+------+--------+-----+----------------------+
| 30 | x3 hello             | y3 hi,hello there    | z    | string | 3   | string:hi            |
| 30 | x3 hello             | y3 hi,hello there    | z    | string | 6   | string:he            |
| 30 | x3 hello             | y3 hi,hello there    | z    | string | 13  | string:he            |
+----+----------------------+----------------------+------+--------+-----+----------------------+
```

Figure 4.2.2 example of building the result relaiton

The following is the pseudo code of the above implementation of **evaluateGrep**:

```
Construct an empty relation result with column of search relation R and parameter list 1
For each column col in search relation R
        Call Unix egrep with the pattern and get the returning String array A[]
        For each string line in A
                posVal<- an array of Grep objects from function positionValue(line, pattern)
                Build a relation r that satisfy the condition: value of col=value of line
                For each tuple in r, duplicate it to the same number as length of posVal
                For tuple k of them,
                   put Attr as name of col, Type as type of col
                   and Pos as posVal[k].pos, Val as posVal[k].val
                Union r with result
Return result
```

If grep has parameter list 2 call **positionValueP**(Discuss in section 4.2.4) instead and build the result relation with attributes from parameter list 2

Figure 4.2.3 grep implementation

Now let's find out how the function **positionValue** works to get position and value of a match of a pattern in a string.

The Function **positionValue**:

```
public Grep[] positionValue(String line, String pattern)
```

It returns an array of Grep objects that contain position and value pairs of the matches of `pattern` in string `line`.

Idea:

The function builds up the array of Grep objects in a loop by calling the UNIX egrep with a piece of string starting from the every beginning of the string line with the initial length of 1 and the pattern. The UNIX egrep returns the string that it searches if it finds a match in it. Otherwise it will return null. The function **positionValue** will increase the length of search string by 1 and call the UNIX egrep until it finds a match or reaches the end of the string. When it finds a match, it calls the function **backwards** with this piece of string s and the pattern to get the starting position of the match value. The function **backwards** will call the UNIX egrep with a piece of s starting from the end, with initial length of 1 and the pattern to see if it finds a match. It will increase the length of search string by moving backwards from the starting position of the piece of s and calling the egrep until it finds a match. That piece of string will be the value of the match and the starting position of that piece of string will be the start position of that match. This position and value pair will be stored in a Grep object and added to the result array. The fuction **positionValue** then will set starting position to the next position and length to 1 and will start over again

to find the next match in line. The following two figures are flowcharts of these two functions.

The following is the flowchart of the function **positionValue:**

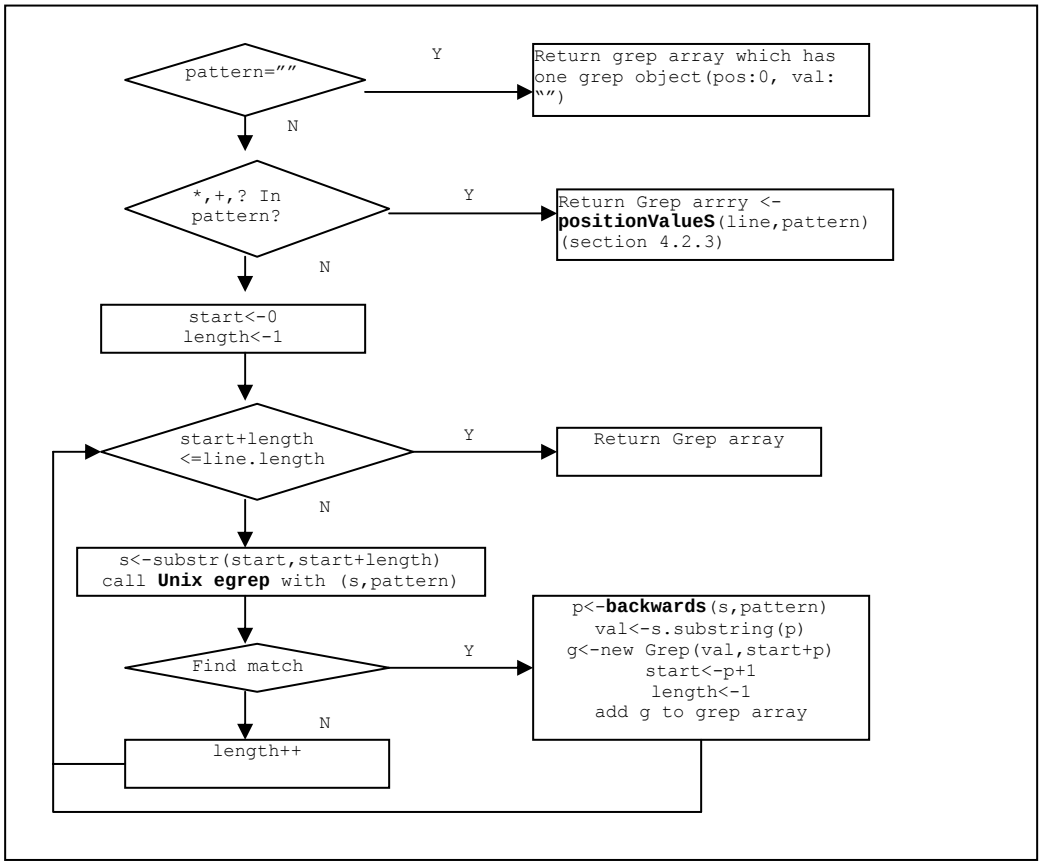


Figure 4.2.4 flowchart of function positionValue

The function **backwards**:

```
public int backwards(String s, String pattern)
```

It returns the start position where a pattern matche is found in a string s.

The following is the flowchart of the **backwards** function**:**

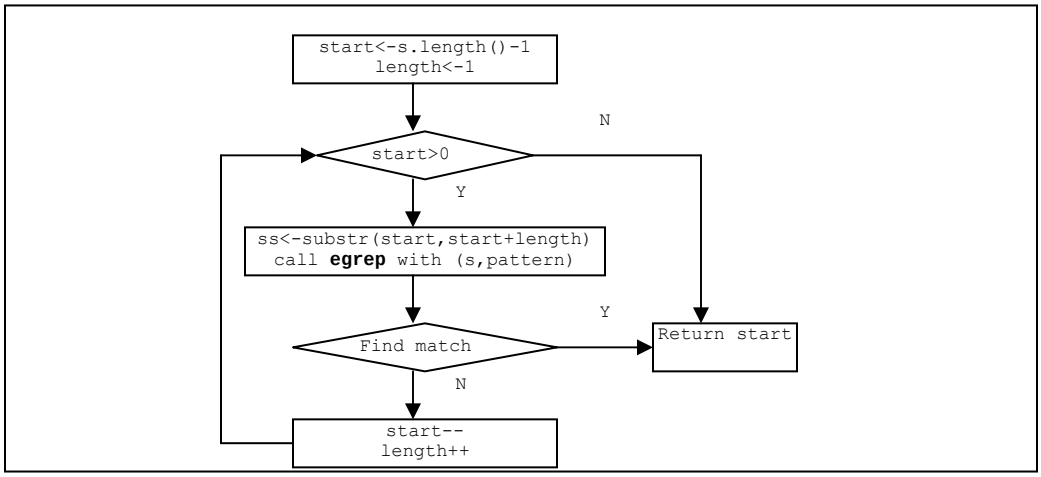


Figure 4.2.5 flowchart of function backwards

Let's use the same relation R in section 3.1.1 and use the following grep command:

```
grep(Attr,Type,Pos,Val) "h." in R;
```

Now we pass "y3 hi, hello there" as the value of `line`. And "h." as the value of `pattern` to the function **positionValue**. The following figure shows how the function works on this example.

Round 1

1. Start=0, length=1

y	3		h	i	,	h	e	l	l	o		t	h	e	r	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

2. Start=0, length=2

y	3		h	i	,	h	e	l	l	o		t	h	e	r	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

3. Start=0, length=3

y	3		h	i	,	h	e	l	l	o		t	h	e	r	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

4. Start=0, length=4

y	3		h	i	,	h	e	l	l	o		t	h	e	r	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

5. Start=0, length=5

y	3		h	i	,	h	e	l	l	o		t	h	e	r	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Find a match, call **backwards** to determine the start position of that match. In this case **backwards** returns 3
Construct a grep object(3,'hi') and store it to output array.
Set start to start+1, in this case, 4, and set length to 1

Round 2

1. Start=4, length=1

y	3		h	i	,	h	e	l	l	o		t	h	e	r	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

2. Start=4, length=2

y	3		h	i	,	h	e	l	l	o		t	h	e	r	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

3. Start=4, length=3

y	3		h	i	,	h	e	l	l	o		t	h	e	r	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

4. Start=4, length=4

y	3		h	i	,	h	e	l	l	o		t	h	e	r	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Find a match, call **backwards** to determine the start position of that match. In this case **backwards** return 6
Construct a grep object(6,'he') and store it to output array.
Set start to start+1, in this case, 7, and set length to 1

Round 3

1. Start=7, length=1

y	3		h	i	,	h	e	l	l	o		t	h	e	r	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

2. Start=7, length=2

y	3		h	i	,	h	e	l	l	o		t	h	e	r	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

…

8. Start=7, length=8

y	3		h	i	,	h	e	l	l	o		t	h	e	r	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Find a match, call **backwards** to determine the start position of that match. In this case **backwards** return 13
Construct a grep object(13,'he') and store it to output array.
Set start to start+1, in this case, 14. set length to 1

Round 4
Haven't find any match before reach the end of string

Figure 4.2.6 example of executing the function **positionValue** with the pattern ".h" and a search string "y3 hi, hello there"

The function **positionValue** has limitations, because it can not return a correct value wherever there a "?", "+", or "*" in it. The reason is that the uncertainty of the length of the match string. The following section will deal with the case when there is wild card.

4.2.3 Dealing with the wildcard

Before we discuss the implementation, let's see why the function **positionValue** can not get the correct result. We use the same search string and the pattern "h.*e" to illustrate the problem.

Round 1

1. Start=0, length=1

y	3		h	i	,	h	e	l	l	o		t	h	e	r	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

…

8. Start=0, length=8

y	3		h	i	,	h	e	l	l	o		t	h	e	r	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Find a match, call **backwards** to determine the start position of that match. In this case **backwards** return 6
The match is (6,'he') But here what we want is (3,'hi,he').

Round 2

1. Start=7, length=1

y	3		h	i	,	h	e	l	l	o		t	h	e	r	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

…

8. Start=7, length=8

y	3		h	i	,	h	e	l	l	o		t	h	e	r	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Find a match, call **backwards** to determine the start position of that match. In this case **backwards** return 13
The match is (13,'he') But here what we want is (3,'hi,hello the').

Round 3

1. Start=15,length=1

Haven't find any match before reach the end of string
But want we want also include the following: (3,'hi,he') ,(3,'hi,hello the'), (3,'hi,hello there'), (6,'hello the') and (6,'hello there')

Figure 4.2.7 problem of **positionValue** when a pattern has *, + or ?

So, we make another function **positionValueS** when there is a *, + or ? in the pattern.

The function **positionValueS** takes the same parameters and return the same type as the function **positionValue** does.

The function **positionValueS**:

public Grep[] **positionValueS**(String line,String pattern)
It takes two parameters:
line: search string
pattern: search pattern

It returns an array of Grep objects that contain the position and value pairs of the matches of pattern in line

Idea:

The idea is to break the pattern before and after the aforementioned wild card to two pattern and call the **positionValue** respectively, then merge the result.

A typical pattern example is: `patternBefore.*patternAfter`

The idea of merging is simple, since the match of pattern before the wild card always has the smaller position value than the match of pattern after the wild card. We just need to combine those that satisfy this condition.

Let's go through the same example to show how this function works when dealing with ".*"

pattern = 'h.*e'
patternBefore='h'
patternAfter='e'
call positionValue with search string and patternBefore and get array of grep object. They are (3,'h'), (6,'h'), (13,'h').
call positionValue with search string and patternAfter and get array of grep object. They are (7,'e'), (14,'e'), (16,'e').

Round 1 for the first grep object (3,'h')

1.

y	3		h	i	,	h	e	l	l	o		t	h	e	r	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Create (3,'hi,he')

2.

y	3		h	i	,	h	e	l	l	o		t	h	e	r	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Create (3,'hi,hello the')

3.

y	3		h	i	,	h	e	l	l	o		t	h	e	r	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Create (3,'hi,hello there')

Round 2 for the second grep object (6,'h')

1.

y	3		h	i	,	h	e	l	l	o		t	h	e	r	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Create (6,'he')

2.

y	3		h	i	,	h	e	l	l	o		t	h	e	r	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Create (6,'hello the')

3.

y	3		h	i	,	h	e	l	l	o		t	h	e	r	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Create (6,'hello there')

Round 3 for the second grep object (13,'h')

1.

y	3		h	i	,	h	e	l	l	o		t	h	e	r	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Create (13,'he')

2.

Y	3		h	i	,	h	e	l	l	o		t	h	e	r	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Create (13,'here')

Figure 4.2.8 example of positionValueS dealing with ".*"

We can see from Round 3, (7, 'e') can not combine with (13, 'h'), since 13 is greater than 7.

We need a slightly different code to deal with ".+" and ".?", because they have a different meaning from ".*".For example, if we use the same search string as the previous example by just changing the pattern to "h.+e" , we do not get "he" at round 2 and round 3. We need to make sure there is at least one character in between the values of patternBefore

and patternAfter. So in the code, we just need to change one line: “if (begin<=end)” to “if (begin<end)”.

Similarly for “.?”, we need to make sure there will be no more than one character between the values of patternBefore and patternAfter. In the previous example, if we change pattern to “h.?e”, we will get only two results: “he” in round 2 and round 3. In the code, we need to change the condition to begin=end or begin=end-1.

The following is the pseudo code of the **positionValueS** when dealing with wild card “.*”.We need first to check if they appear at the beginning or at the end of the pattern, then we deal with other situation. The code is changed slightly for “.+” and “.?”. For “.+”, we need to get rid of the result where there is no character in between the value of pattern before and pattern after. For “.?” , we need to get rid of the result where there is more than one character in between the two value.

```
dotStar<-position of ".*"
if (dotStar==0) //at the beginning of the pattern, for example: .*l
    patternAfter = pattern.substring(2);
    Grep[] posV = positionValue(line,patternAfter);
    for (int i = 0;i<posV.length;i++)
            int pos = posV[i].position;
            int valL = posV[i].value.length();
            posV[i].value = line.substring(0,pos+valL);
            posV[i].position = 0;
    return posV;
if (lastDotStar==pattern.length()-2) //at the end of the pattern, like l.*
    patternBefore = pattern.substring(0,pattern.length()-2);
    Grep[] posV = positionValue(line,patternBefore);
    for (int i = 0;i<posV.length;i++)
            int pos = posV[i].position;
            int valL = posV[i].value.length();
            posV[i].value = line.substring(pos);
    return posV;

patternBefore <- pattern.substring(0,dotStar);
patternAfter <- pattern.substring(dotStar+2);
Grep[] posVB <- positionValue(line,patternBefore);
Grep[] posVA <- positionValue(line,patternAfter);
for (int i = 0; i<posVB.length; i++)
   for (int j = 0; j<posVA.length; j++)
      int begin = posVB[i].position+posVB[i].value.length();
      int end = posVA[j].position;
      if (begin<=end)
          create a new Grep object
          set its value to line.substring(posVB[i].position,posVA[j].position +
                  (posVA[j].value).length()
          set its position to posVB[i].position;
          add this Grep object to output array posV;
return posV;
```

Checking the repetition in between when there is no "." before *, + or ? (figure 4.2.10)

Figure 4.2.9 example of the function **positionValueS** dealing with ".*"

Things get a bit more complicated when there is no "." before "*", "+" or "?". Now we need to get the character or set of character that can be repeated in the match. There are three cases: Suppose beforeChar is a character or characters in the second and third case. * could be + or ?.

patternBeforebeforeChar*patternAfter

patternBefore[beforeChar]*patternAfter

patternBefore(beforeChar)*patternAfter

The first case, there is a character before the wild card. In this case, we need to ensure that in between the matches of patternBefore and patternAfter are repetitions of that

character. In the second case, there is a set of characters in []. In this case, we must ensure that the characters that are in between the matches are repetitions of any one from this set. In the third case, () is before the wild card, which means that, we treat everything inside () as a single atomic item. So in between the matches there may be repetitions of this item. The number of repetitions depends on which one of "*", "+" or "?" is used. In the implementation, we need to keep the value of a repeating item for each of the three cases, and we need to add this verrification after the "`if (begin<=end)`" statement in code in figure 4.2.9. The following is the pseudo code for this checking.

```
beforeChar<-value of the repetition item
…
        in between the begin and end
        check if they are the repetitions of beforeChar
        if not skip else create a new Grep object
```

Figure 4.2.10 **positionValueS** dealing with "*","+","?" without preceding "."

4.2.4 Implementation of parameter list 2

Once we understand the implementation of grep with parameter list 1, it is not hard for us to understand how we implement grep for the parameter list 2. Since the parameter list 2 contains user defined parameters, and they are also in the pattern, which means they are part of the value of the match. So we still can use most of our code when dealing with the parameter list 1. We will make some changes to the code, and we need to add a parameter array for each pair of (position, value), since now each match value can contain several user defined parameters in between, and we need to store their value. We use a paraValue in the Grep class to do that as shown in Figure 4.2.1.

As shown in Figure 4.2.6, function evaluteGrep will call positionValueP when grep has parameter list 2. Before doing that, there is some preliminary work that is required. Let's use an example from figure 3.1.27 to show how we implement grep with parameter list 2,

grep(Attr,Type,Pos,Val;before,after,end) "\beforeh.\aftero\end" in R;

We still use the search string:

"y3 hi,hello there"

We will get two grep objects:

(position:0, value: "y3 hi,hello there", before: "y3 ", after: ",hell", end: " there")

(position:0, value: “y3 hi,hello there”, before: “y3 hi,”, after: “ll”, end: “ there”)

From the above example we can see that the user defined parameter represents part of the value. It can be treated as “.*”, since it can be anything at that position in between the pattern. Remember we have the function **positionValueS** that deals with “.*”. We just need to make slight modifications such that it can work for the user defined parameters. What we need here is an array of domain from the parameter list 2. Before we call the function **positionValueP**, we need to check the pattern to make sure all these user defined parameters are in the parameter list 2 and store all these parameters that appeared in the pattern in Domain array domN. The following is the pseudo code of the function **positionValueP**, Which finds the first parameter in domN, and takes everything before it as patternBefore and everything after it as patternAfter, get the Grep array for each of them and merge the result.

The function **positionValueP**:

```
public Grep[] positionValueP(String line,String pattern,String[] domN,
Domain[] param)
```

It takes four parameters:

`line`: search string

`pattern`: search pattern

`domN`: domain name array, which includes all the user defined parameters in `pattern`

`Param`: domain array of the parameter list 2

It returns array of the Grep objects which contains position and value pairs, and array of values of all user defined parameters of the matches of pattern in line

```
if (domN.length==0) return positionValue(line,pattern);
String paV[] = new String[param.length];
ind<-position of domN[0] in array param
para<-"\"+ domN[0]
dotStar <- position of para in pattern
patternBefore = pattern.substring(0,dotStar);
patternAfter = pattern.substring(dotStar+para.length());
Grep[] posVB = positionValue(line,patternBefore);
String paraN[] = new String[domN.length-1];
for (int i = 1; i<domN.length; i++)
   paraN[i-1]=domN[i];
if patternAfter is empty
      Grep g<- a new grep object with
        position: line.length()
        value:''
        paraValue[ind]<-substr(line.position of para in line)
      put it in posVA
else
      Grep[] posVA = positionValueP(line,patternAfter,paraN,param);
for (int i = 0; i<posVB.length; i++)
   for (int j = 0; j<posVA.length; j++)
   {
    int begin = posVB[i].position+posVB[i].value.length();
    int end = posVA[j].position;
    if (begin<=end)
    {
     String paraV = line.substring(begin,end);
     if (domN.length==1)
      for(int k=0;k<posVA[j].paraValue.length;k++)
        if (posVA[j].paraValue[k]!=null)
          paV[k]=posVA[j].paraValue[k];
     paV[ind]= new String(paraV);
     Grep g <-create a Grep object with
        value : line.substring(posVB[i].position,end+posVA[j].value.length())
        position : posVB[i].position
        and user defined parameter array paV
     add g to Grep array posV
    }
}
return posV;
```

Figure 4.2.11 the function **positionValueP** dealing with parameter list 2

The following is a walk through example of calling **`positionValueP`** with

```
line: 'y3 hi,hello there'
pattern: '"\before h.\after o\end'
domN: ['before','after','end']
param:[before,after,end]
```

Call 1: ind=0; para='\before'; dotStar=0; patternBefore=''; patternAfter='**h.**\after**o**\end'
posVB: [(0,'')]; paraN: ['after','end']

Call 2 posVA<-call **positionValueP** with line; patternAfter: '**h.**\after**o**\end'; paraN: ['after','end']; param
ind=1; para='\after'; dotstart=2; patternBefore='h.'; patternAfter='**o**\end'
posVB: [(3,'hi'),(6,'he'),(13,'he')]; paraN: ['end']

Call 3 posVA<-call **positionValueP** with line; patternAfter: '**o**\end'; paraN: ['end']; param
ind=2; para='\end'; dotstart=1; patternBefore='o'; patternAfter=''
posVB: [(10,'o')]; posVA:[(17,'')]
inside the for loop:

1. begin=11; end=17; paraV=' there'; g<-('o there', 10, paV[2]=' there'); put g in posV, return posV

y	3		h	i	,	h	e	l	l	o		t	h	e	r	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

1. begin=5; end=10; paraV=',hell'; g<-('hi, hello there', 3, paV[2]=' there'; paV[1]=',hell'); put g in posV

y	3		h	i	,	h	e	l	l	o		t	h	e	r	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

2. begin=8; end=10; paraV='ll'; g<-('hello there', 6, paV[2]=' there'; paV[1]='ll'); put g in posV

y	3		h	i	,	h	e	l	l	o		t	h	e	r	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

3. begin=13; end=10 not satisfy the condition: begin<=end
return posV with 2 Grep object

1. begin=0; end=3; paraV='y3 '; g<-('y3 hi,hello there',0, paV[2]=' there'; paV[1]=',hell'; paV[0]='y3 '); put g in posV

y	3		h	i	,	h	e	l	l	o		t	h	e	r	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

2. begin=0; end=6; paraV='y3 hi,'; g<-('y3 hi,hello there',0, paV[2]=' there'; paV[1]=',hell'; paV[0]='y3 hi,'); put g in posV

y	3		h	i	,	h	e	l	l	o		t	h	e	r	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Figure 4.2.12 example of calling **`positionValueP`**

4.2.5 Other cases (no parameter list, use relation or top level scalar as pattern)

There are some special cases of the grep command. The simplest case where there is no parameter list in the grep command is illustrated in example in Figure 3.1.1. The following is the pseudo code. It is similar to the code in Figure 4.2.6, except here we don't need to worry about the parameter list 1 in the result relation.

```
Construct an empty relation result with column of search relation R
For each column col in search relation R
        Call Unix egrep with the pattern and get the returning String array A[]
        For each string line in A
                Build a relation r that satisfy the condition: value of col=value of line
                Union r with result
Return result
```

Figure 4.2.13 grep implementation without parameter list

Besides regular expression pattern, in our grep command, we can use top level scalar and relation as pattern. In the case of top level scalar, we treat it as a string that we want to find in the search relation. It is unnecessary to call the UNIX egrep to get the match; this can be performed by a Java function used to find the position if there is parameter list 1 in the grep command. And the value is just the same as the pattern. It is useful when we want to search a large number of strings in the search relation. We can use relation as pattern to achieve that where we treat relation as a set of search string; we will get a match when either one value in the pattern relation is found in the search relation. What we need here in the code is to add a loop. No pseudo code is provided as this operation is rather straightforward. Please find the code in the function **evalGrepScalarPV** and **evalGrepRelPV** when there is parameter list 1 and **evalGrepRel**, **evalGrepScalar** for the case without parameter list 1.

4.3 Implementation of Union Type

Hitherto, types have been primitive such as integer, string, nested relation, but now we wish to implement types with previously declared domains and unions of types. To parse these extended types we must support after the list of identifiers whose types are being declared, not just one of the primitive type names but any expression made of primitive type names and identifiers(previously declared domains), by one or more union operator('|'). To determine which particular domain in the declaration of a union type is referred to by the string initializing the value on the relation we must first ensure there is a mechenism to expand the declaration so that we can determince the primitive types or nested domains that these union types are declared upon. In addition, we must find the match for the initialization string by comparing the expanded types with the parse tree of the initialization string. The first, union type domain declaration will be discussed in section 4.3.2. The second, relation initialization will be discussed in section 4.3.3(which gives as backgroud, the way we implement relation initialization before we introduce union types) and section 4.3.4. But first, let's look at the parsing issue in section 4.3.1.

4.3.1 Syntax of union type domain declaration

First, let's take a look at the old parser for domain declaration.

```
<DOMAIN> <IDList> <Type>
```

For example, you can declare x and y to be string in the following way:

```
Domain x,y string;
```

Now, in order to accommodate union type, we need to add "|" to express the meaning of "or" and we need to use * to say a union type can have more then one type. We also need to expand type so that it can be an identifier in addition to all the primitive types and nested relation. The following is the new parser for domain declaration:

```
<DOMAIN> <IDList> <Type> (| <Type>)*
```

Please see appendix for syntax detail.

4.3.2 Implementation of union type domain declaration

We need to have the definition of union type in Domain class and store it in a disk file ".union", so that each time when JRelix is loaded, the definition of union type is also

loaded into memory, and each time when JRelix is quit, the definition of union type is dumped into this file.

In Domain class, we add the following field to represent union type. This is simply the type array copied from the domain declaration parse tree.

```
public SimpleNode[] union;
```

And we also need a new constructor for union type domain.

Now, let's take a look at a declaration parse tree example. We use the same example as shown in Figure 3.2.1.

In figure 4.3.1, the root node indicates that this is a domain declaration parse tree. It has at least two type of nodes. The first one is always an IDList node, which will have all the identifiers of domains that being declared. Second type of node is Type node. For a normal domain, it has only one Type node, and the type of node can not be identifier, which is indicated in opcode field of the node(shown in bold in the figure). For a union type domain, it can have one or more Type node, and the Type node can be an identifier. That's how we distinguish a normal domain from a union type domain. Once a union type domain is declared, the Type nodes will be assigned to union array of that domain, and saved in .union file.

```
>domain c1 n1|string;
```

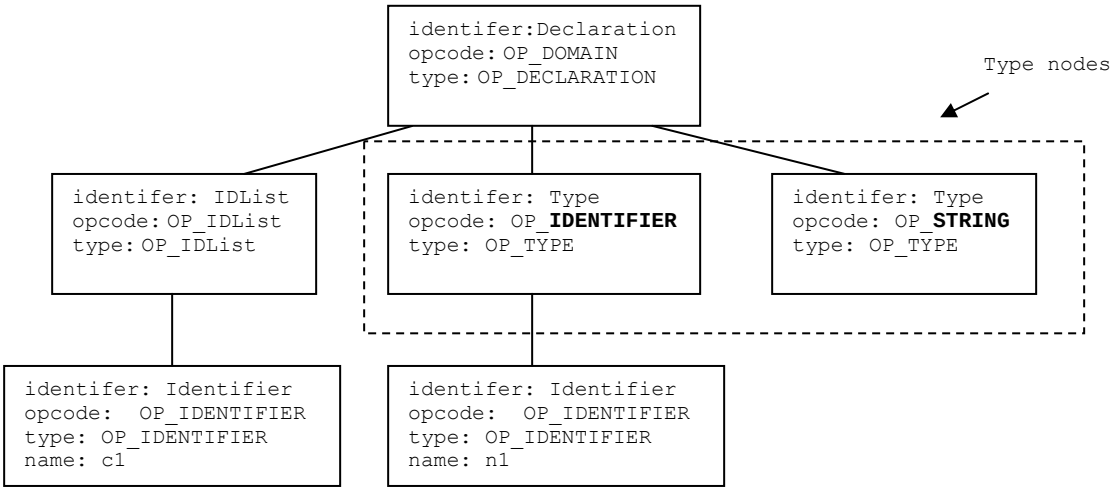


Figure 4.3.1 An example of union type domain definition tree

4.3.3 Initialization of relation without union type domains

In order to minimize the impact on the existing system, we do not change the syntax of relation initialization. So the syntax of initializing a relation with union type domains is the same as initializing a relation without a union type domain. We don't need to specify which particular type we use when we provide the initialization sting. This causes some difficulties when we implement. First of all, the system needs to decide which particular type is actually used by comparing the union type domain declaration with the initialization string. Second, once the real type is determined, it needs to be stored with the input data value. Otherwise, when we retrieve the data later on, we can not tell which type it really is.

Before implementing relation initialization for union type domains, we need to understand the existing function **RelationalInitialization** so that we will understand the modifications that need to be done to this function. Let's take a look at a simple example.

```
>domain a integer;
>domain b string;
>domain n1(a,b);
>relation R1(a,n1)<-{(1,{(1,"1"),(2,"2")}),(2,{(2,"2")})};
```

The values will be stored in relation R1 and .n1 in the following way:

```
pr R1;
+-------------+----------------------+
| a           | n1                   |
+-------------+----------------------+
| 1           | 1                    |
| 2           | 2                    |
+-------------+----------------------+
relation R1 has 2 tuples
```

In relation R1, n1 stores surrogate of the nested relation.

```
>pr .n1;
+----------------------+-------------+----------------------+
| .id                  | a           | b                    |
+----------------------+-------------+----------------------+
| 1                    | 1           | 1                    |
| 1                    | 2           | 2                    |
| 2                    | 2           | 2                    |
+----------------------+-------------+----------------------+
relation .n1 has 3 tuples
```

In relation .n, .id stores surrogate of the nested relation.

Now let's take a closer look at the initialization string. In figure 4.3.2, it shows the parse tree of the initialization string. To implement the initialization, the function traverses this tree and compares the node with the domain declaration of that relation. Please ignore the part in dotted circle until section 4.3.4.

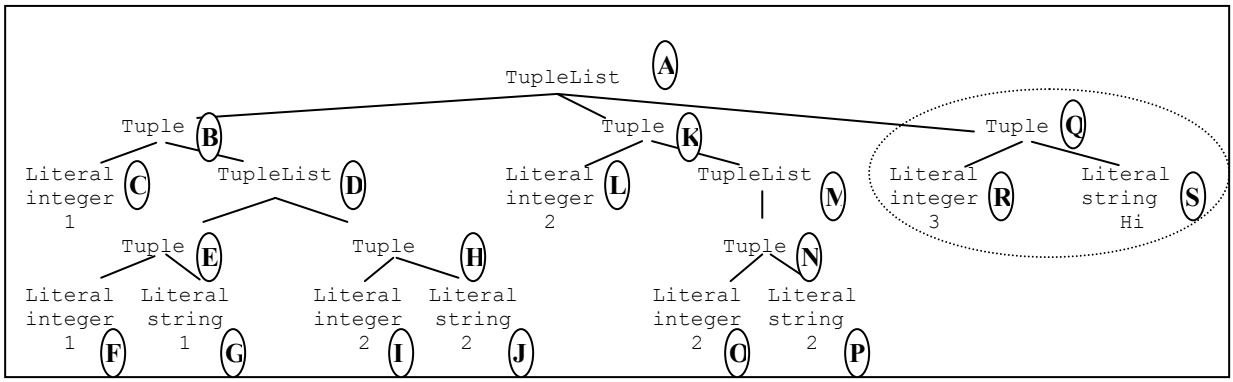


Figure 4.3.2 An example of initialization string parse tree

From the above tree we can see that the root node is always a TupleList, which makes sense, since a relation is a set of tuples. So the type of children of TupleList can only be Tuple. A tuple consists domains, thus the type of childern of Tuple can be a primitive type literal or nested relation that is represented as a TupleList. Please note that this parse tree has nothing to do with the relation declaration or domain declaration, which is the purpose of the function **RelationalInitialization**. It takes the parse tree and the relation as its parameters and checks for each tuple if its node types match with the domain declarations. Upon a positive match, it assigns a value to the data array of the relation. This function will be called recursively once it meets the TupleList type node of a Tuple.

The following is the signature of **RelationalInitialization**.

Long **RelationalInitialization**(SimpleNode node,Hashtable relsHT, String name, long surr, int tupleNum)

Parameter node is the root node of the parse tree. Parameter relsHT is the hashtable that contains all the definition of relations. Parameter name is the name of relation. Parameter surr is the surrogate of the nested relation. Parameter tupleNum is current tuple number of the relation.

The first call is in RelationDeclaration: **RelationalInitialization**(node, relsHT, name, 0, 0)

The main flowchart of this function is shown in figure 4.3.3. Henceforth in the flowchart, anything related to the node in the initialization string parse tree will be illustrated using dashed lines in order to distinguish them from domain declaration.

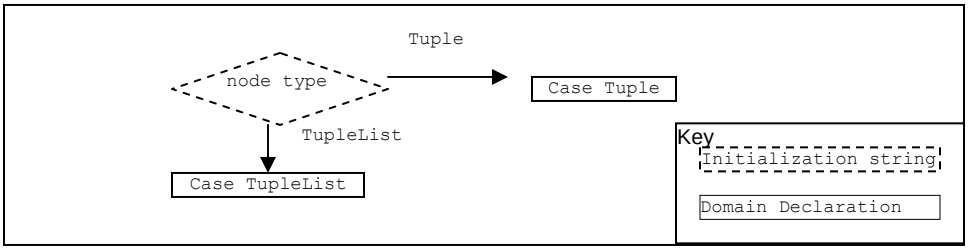


Figure 4.3.3 Function **RelationalInitialization**

The following flowchart is the expanding of case TupleList. It calls the function **RelationalInitialization** again with each tuple node. When implementing the union type domain, there is nothing changed in this part of the code. Please note, here the type of node is TupleList, all its children are Tuple type nodes.

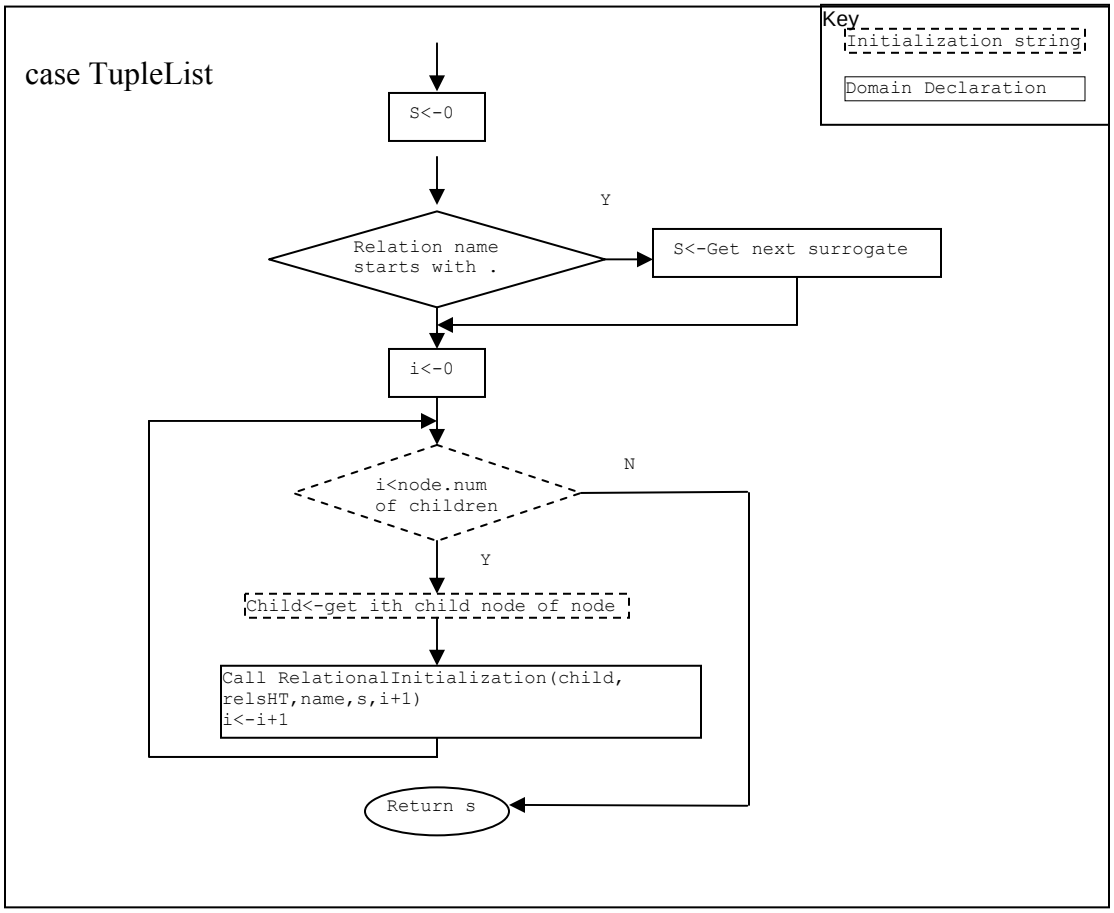
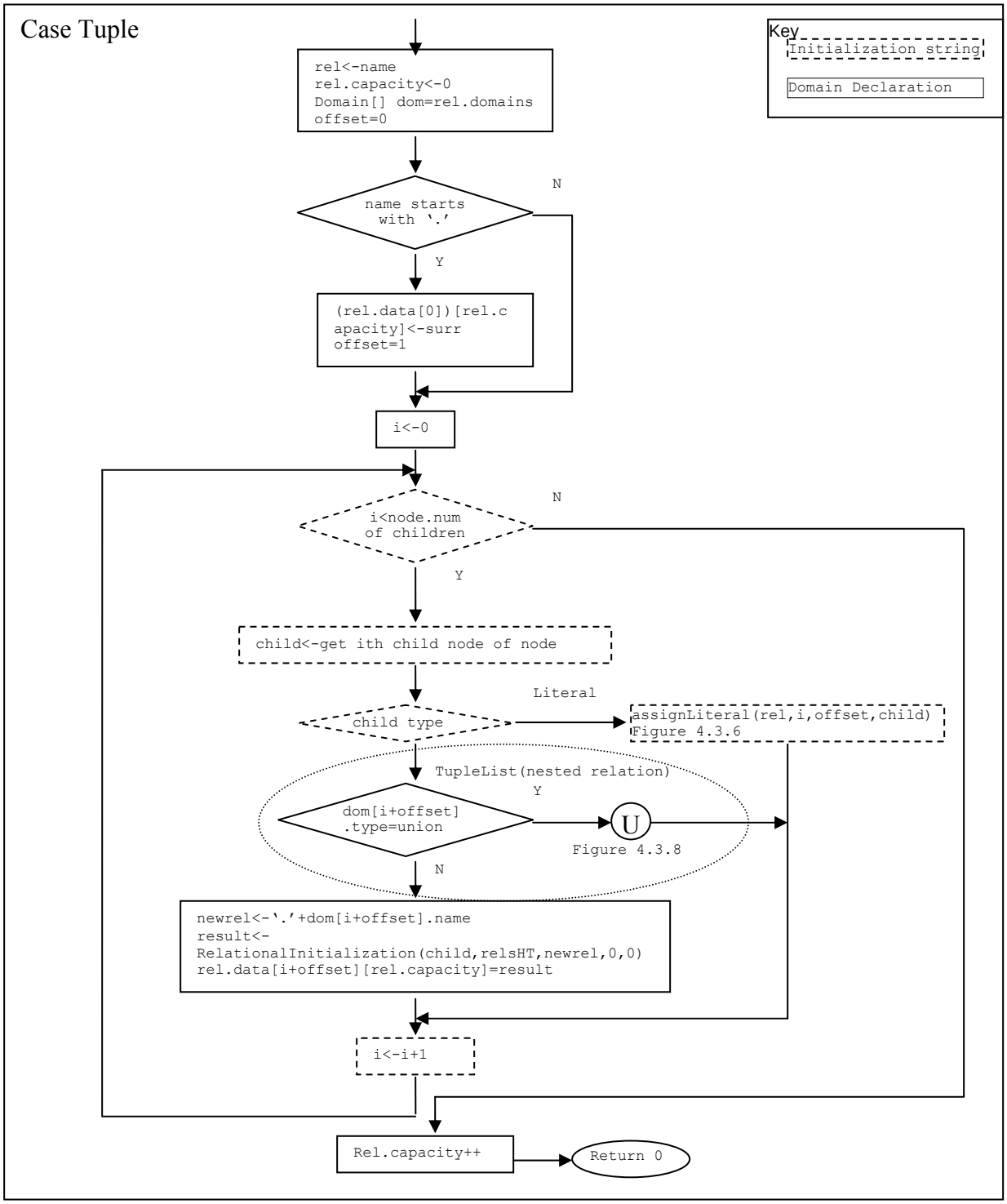


Figure 4.3.4 Case TupleList

Now, let's take a look at case Tuple. In figure 4.3.5, the dotted circle is where we need to deal with union type domain. We can ignore it for now. Also, we need to deal with union type in the function **assignLiteral**, we will explain it later in this section. Please note, here, node type is Tuple, its child node type can be Literal or TupleList. When the child node type is TupleList, it means this domain input is a nested relation.



The function **assignLiteral** is very simple. The following is the flowchart of **assignLiteral**, please ignore for now the part in dotted circle until section 4.3.4, where the union type is dealt with.

assignLiteral `(Relation rel, int i, int offset, SimpleNode node)`

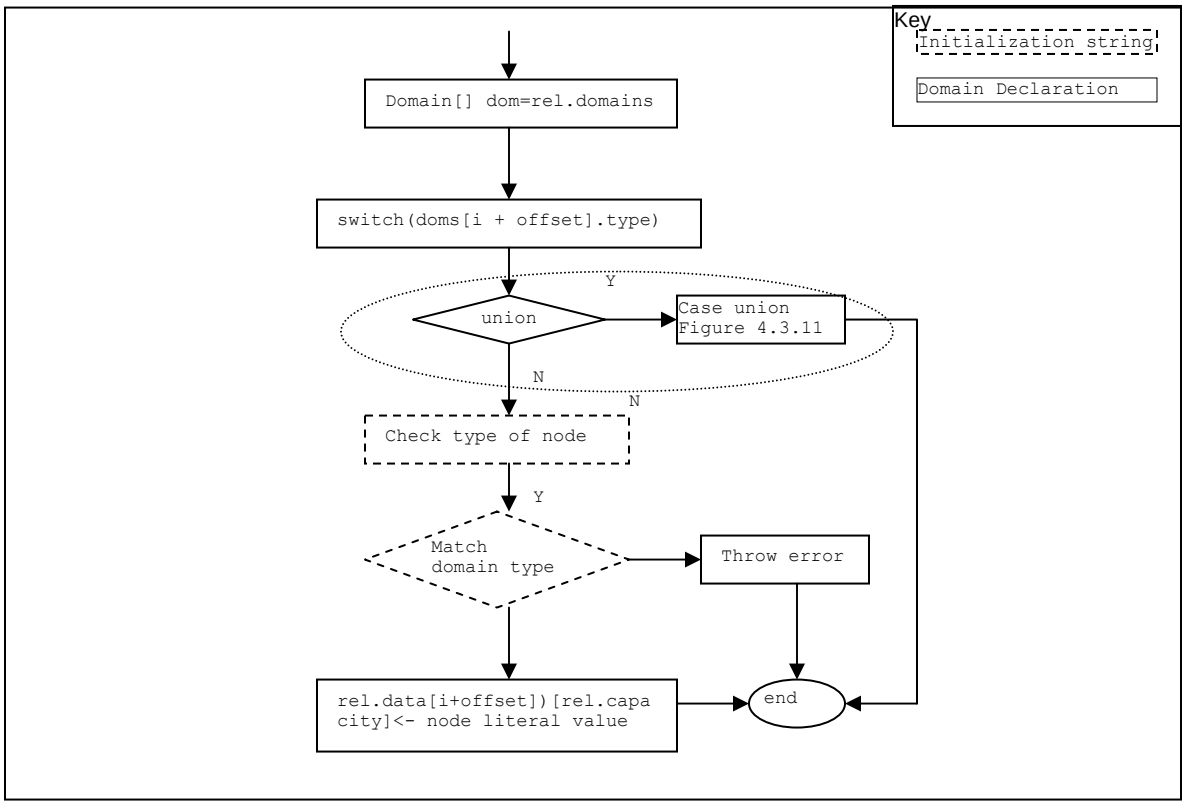


Figure 4.3.6 the function **assignLiteral**

Now, let take the example initialization string parse tree shown in figure 4.3.2, we can walk through the function flowchart to see how the initialization of relation R1 is performed. We can ignore the parts in dotted square until section 4.3.4 after we finishing the discussion with union type domain initialization.

```
call 1: RelationalInitialization(A, relsHT, R1, 0, 0)
          Type of A: TupleList
          First child of A: B
          call 2: RelationalInitialization(B, relsHT, R1, 0, 1)
                    Type of B: Tuple
                    First child of B: C
                    Type of C: Literal
                    Call assignLiteral(R1,0,0,C)
                              (R1.data[0])[0]<-1
                    Second child of B: D
                    Type of D: TupleList
                    c1.type is union; first domain of c1 is n1; n1 is
                    idList; match(.n1,D) return true.
                    newrel<-.n1
                    call 3 result<- RelationalInitialization(D,relsHT,.n1,0,0)
                              Type of D: TupleList
                              S<-1
                              First child of D: E
                              Call 4 RelationalInitialization(E, relsHT,.n1,1,1)
                                        Type of E: Tuple
                                        (.n1.data[0])[0]<-1
                                        offset<-1
                                        Call assignLiteral(.n1,0,1,F)
                                                  (.n1.data[1])[0]<-1
                                        Call assignLiteral(.n1,1,1,G)
                                                  (.n1.data[2])[0]<-'1'
                                        .n1.capacity<-1
                              Second child of D: H
                              Call 5 RelationalInitialization(H, relsHT, .n1, 1, 2)
                                        Type of H: Tuple
                                        (.n1.data[0])[1]<-1
                                        offset<-1
                                        Call assignLiteral(.n1,0,1,I)
                                                  (.n1.data[1])[1]<-2
                                        Call assignLiteral(.n1,1,1,J)
                                                  (.n1.data[2])[1]<-'2'
                                        .n1.capacity<-2
                              return 1
                    (R1.data[1])[0]<-1
                    R1.capacity<-1
          Second child of A: K
          call 6: RelationalInitialization(K, relsHT, R1, 0, 2)
                    Type of K: Tuple
                    First child of K: L
                    Type of L: Literal
                    Call assignLiteral(R1,0,0,L)
                              (R1.data[0])[1]<-2
                    Second child of K: M
                    Type of M: TupleList
                    c1.type is union; first domain of c1 is n1; n1 is
                    idList; match(.n1,M) return true.

                    Newrel<-.n1
                    Call 7 result<- RelationalInitialization(M,relsHT,.n1,0,0)
                              Type of M: TupleList
                              S<-2
                              First child of M: N
                              Call 8 RelationalInitialization(N, relsHT,.n1, 2, 1)
                                        Type of N: Tuple
                                        (.n1.data[0])[2]<-2
                                        offset<-1
                                        Call assignLiteral(.n1,0,1,O)
                                                  (.n1.data[1])[2]<-2
                                        Call assignLiteral(.n1,1,1,P)
                                                  (.n1.data[2])[2]<-'2'
                                        .n1.capacity<-3
                              return 2
                    (R1.data[1])[1]<-2
                    R1.capacity<-2      (R1.data[1])[1]<-n1:2

          Call 8: RelationalInitialization(Q, relsHT, R1, 0, 3)
                    Type of Q: Tuple
                    First child of Q: R
                    Type of R: Literal
                    Call assignLiteral(R1,0,0,R)
                              (R1.data[0])[2]<-3
                    Second child of Q: S
                    Type of S: Literal
                    Call assignLiteral(R1,0,0,S)
                              (R1.data[1])[2]<-'string: Hi'
                    R1.capacity<-3

          Return 0
```

Figure 4.3.7 an example call of the function **RelationalInitializtion**

4.3.4 Initialization of relation with union type domains

We now have a complete understanding of how the function **RelationalInitialization** works without union type domains. In the following discussion, we will focus on how to deal with union type domains when implement initialization of the relation. Since there is no change to the right hand side of the assignment operator, namely the initialization string, what we need to do is to make some changes to the function so that it can determine what actual type a union type domain is really used and do the assignment. Let's start with an example. Suppose we have the following domain declaration:

```
domain c1 n1|string;
domain n1(a,b);
relation Tc(a,c1)<-{(1,{(1,"1"),(2,"2")}),(2,{(2,"2")}),(3,"hi")};
```

This example is quite similar to what we've discussed in last section except we change the second attribute to union type domain c1 and add one more tuple, with the second attribute of type string. The new tuple is reflected in dotted circle in figure 4.3.2, the third node of root node A. Now, when we do the assignment to a tuple, we still need to check each of its domain type as it originally does, in addition we need to check to see if it is union type domain. In case of it is union type domain, we need to figure out what actual type the initialization string matches with. Please take a look at figure 4.3.5 the dotted circle part. Of course, we need to deal with union type domain also in assignLiteral function if the literal value is assigned to a union type domain. We will discuss this part later after we finish this U part. If we find a tupleList node from the initialization string, before we deal with it as a nested relation, we need to check if the domain is union type, if this is the case, we need to pick one from the list of domains that matches the initialization string, in this case, a nested relation, and then do the assignment. For this example, when we see {(1,"1"),(2,"2")}, we know it is a nested relation and is represented as a TupleList node in the parse tree. What we need to do in this function is we need to check c1's type, since c1 is a union type domain, we need to expand its declaration, which are either n1 or string type. We can ignore string type at once, since it will not match with a nested relation. We will check n1's declaration to see if it matches with the input string, and if so, we do the assignment to .n1, and put surrogate to the relation Tc just as what the function used to do. One more thing we need to add with the surrogate is the name of the nested domain "n1", so that we know that the value in this c1 field is stored in .n1 relation. The following figure 4.3.8 is the flowchart of U part. This

part is actually expanding upon the declaration of union type domain, and checks if each type is a nested domain, if it is, the program will call the **match** function to see if they match each other. If they match, the program will call **relationalInitializaiton** to do the initialization on this nested relation and then assign the name of matched domain and surrogate that this call returned, to this union type domain of the tuple.

Function match takes each tuple of the nested relation from input parameter and checks each attribute with each domain from the nested relation to see if it matches the initialization string. If every domain in the nested relation matches with the domains of every input tuple, the **match** function will return true, otherwise it will return false. The **match** function is itself a recursive function, and the reason is that a domain in the nested relation can be of union type or nested domain type. So the **match** function is needed to expand union type domains in the nested relation and requires a recursive call if any one of the domains is of union type and contains nested domains. Figure 4.3.9 and Figure 4.3.10 is the flowchart of the **match** function.

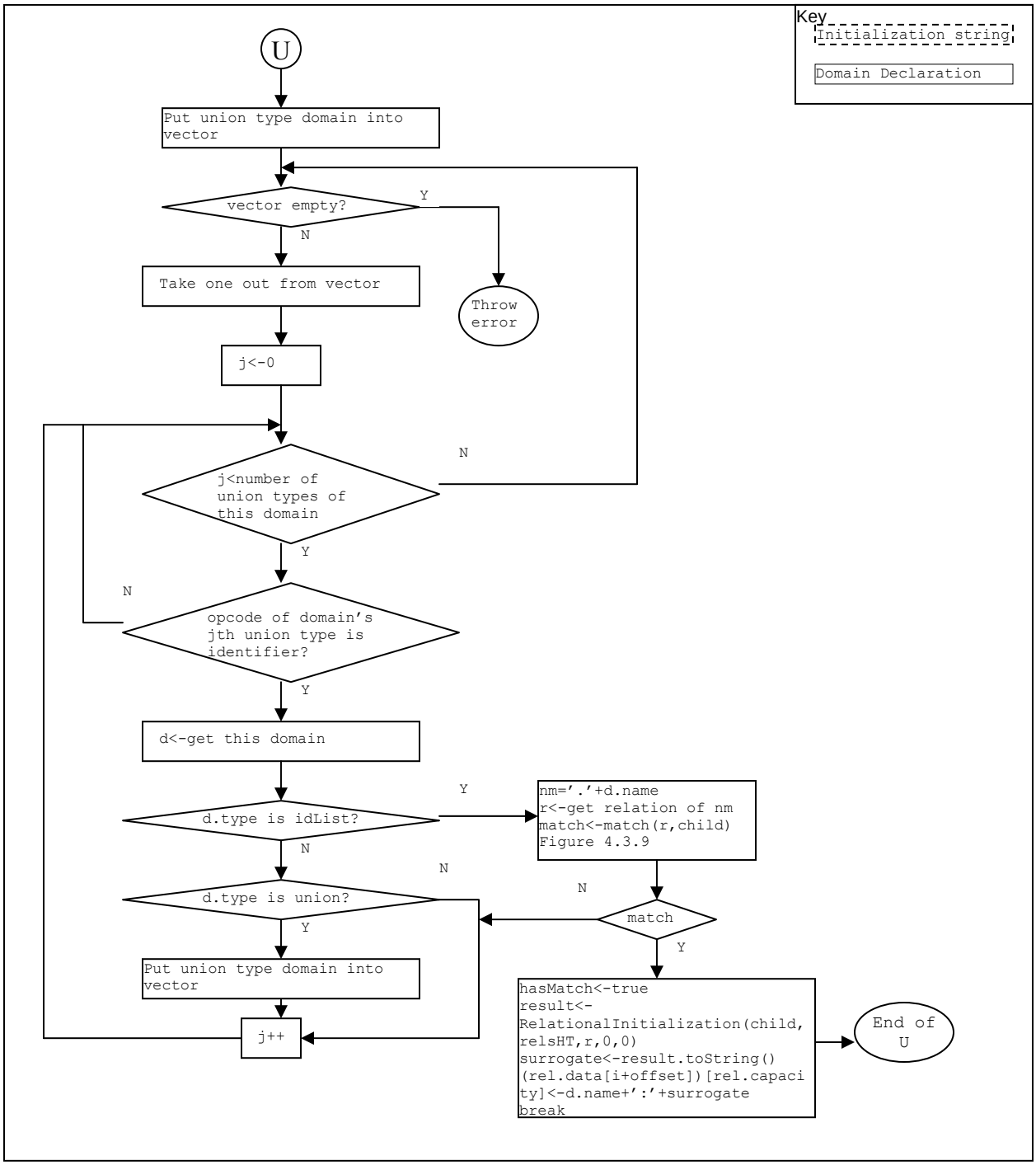


Figure 4.3.8 **RelationalInitializtion**: "U" in Figure 4.3.5 (union type)

boolean **match**(Relation r, SimpleNode child)

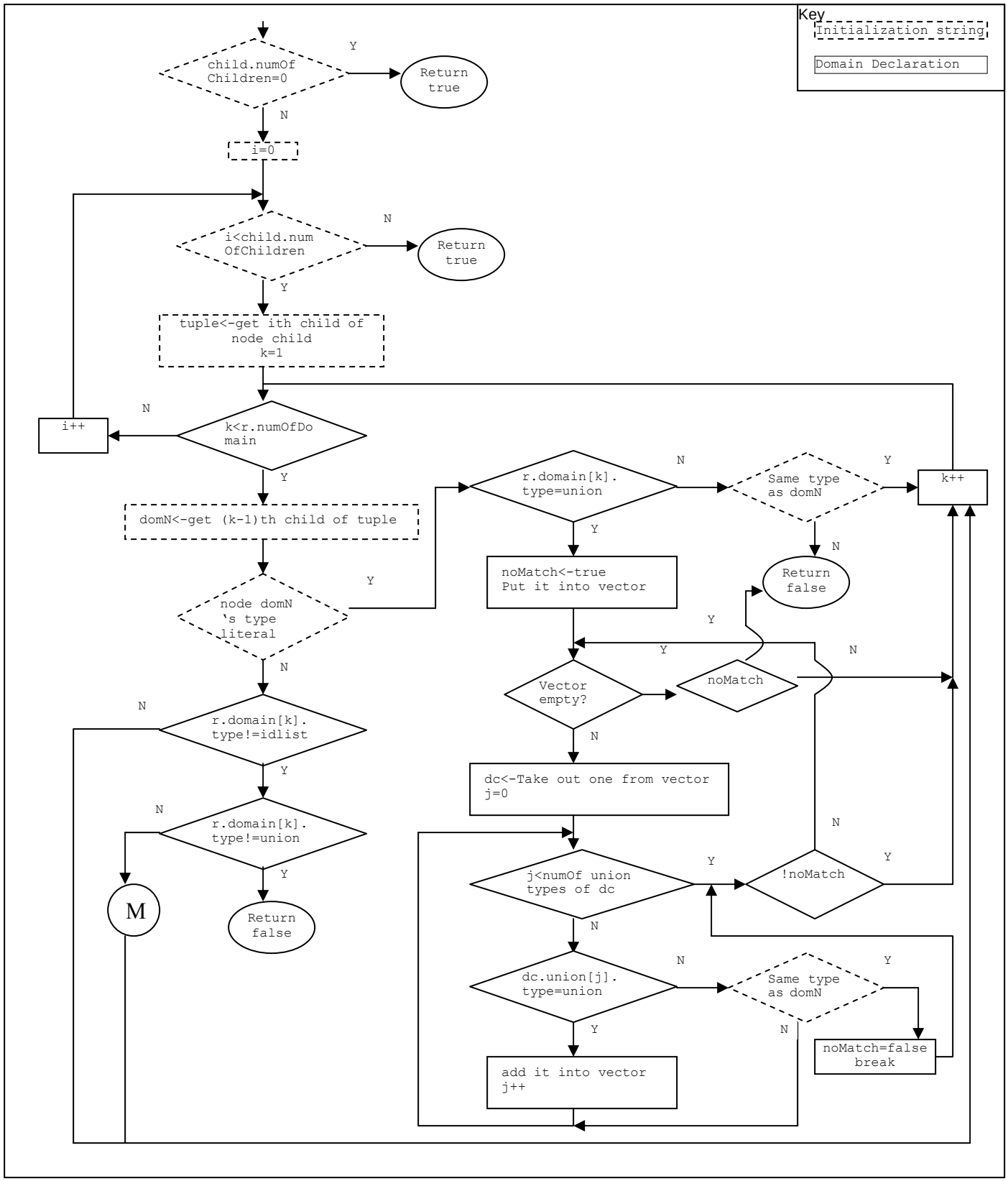


Figure 4.3.9 flowchart of the **match** function

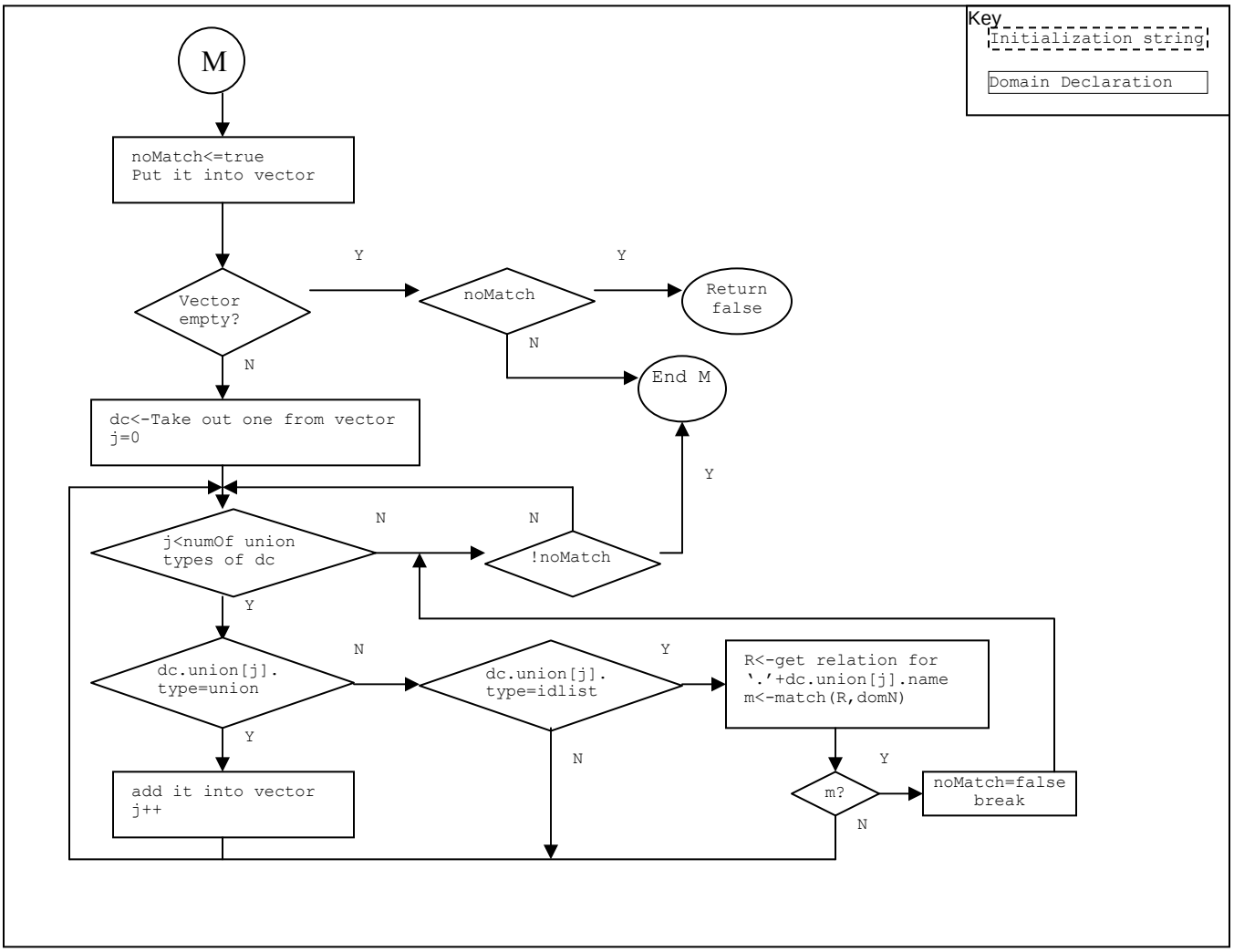


Figure 4.3.10 continue of the **match** function

Now, we can walk through the flowchart shown in figure 4.3.8 and function match in figure 4.3.9 and figure 4.3.10 to see how the initialization with union type domain works. Please see the dotted area in figure 4.3.7. The following is the content of R1.

```
pr R1;
+-------------+----------------------+
| a           | c1                   |
+-------------+----------------------+
| 1           | n1:1                 |
| 2           | n2:2                 |
| 3           | string:Hi            |
+-------------+----------------------+
relation R1 has 3 tuples
```

The function **assignLiteral** is relatively simple since the input string(node type) is literal, it only needs to take one of the expanded union type to see if they match each other. If they match, it will do the assignment with the type information.

The following is the function **assignLiteral** with union type domain

Function: **assignLiteral** `(Relation rel, int i, int offset, SimpleNode node)`
With union types

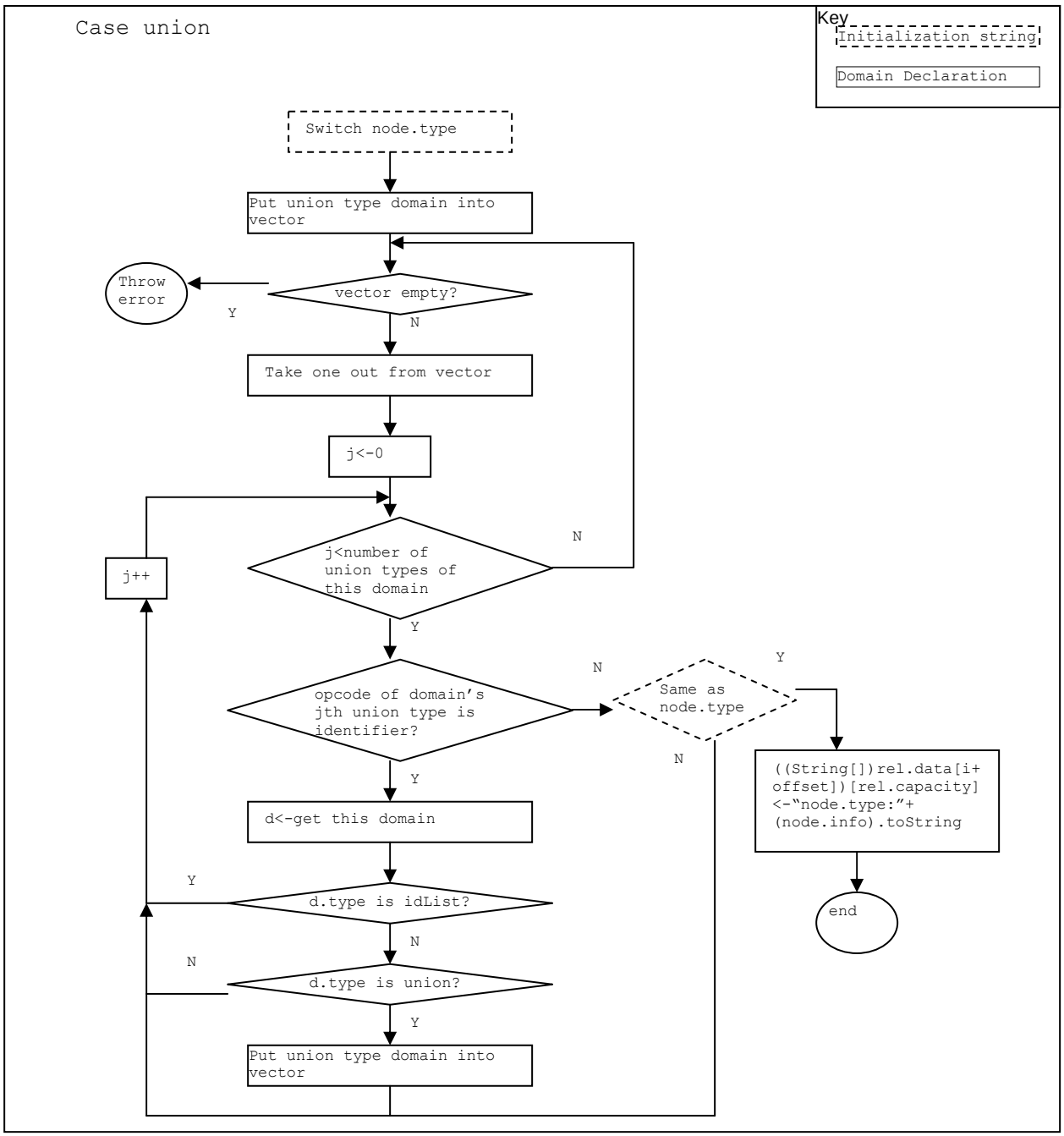


Figure 4.3.11 the function **assignLiteral**(Figure 4.3.6,union type)

4.4 Implementation of the top level scalar

Top level scalar is a primitive type variable with a scope the same as relation. As shown in section 3.3, once declared, it can be used as a relation and a virtual domain.

In order to implement top level scalar, we need to introduce 'declare' in the parser, and we need to store the declaration in both .rel and .dom file so that we can use it as relation and domain as well.

4.4.1 Syntax of the top level scalar declaration and initialization

In parser, we need to add:

`<declare> IDList DType` (DType can be any primitive type)

Top level scalar is declared as the following:

```
>declare s1 integer;
```

After declaration, we can assign it a value using:

```
>s1<-1;
```

Or, we can initialize it when declare it:

```
>declare s3 integer<-3;
```

A top level scalar can be defined on other top level scalar, for example:

```
>declare s4 integer;
>s4<-s1+s3;
```

4.4.2 implementation of the top level scalar

When the interpreter see the top level declaration, it will call the function **DeclareDeclaration**

We also need to add one more possible value of rvc(`RELATION, VIEW or COMPUTATION`) in class Relation, namely `SCALAR`.

The function **DeclareDeclaration**:

It takes the declaration root node from parser. Figure 4.4.1 is a parse tree of top level scalar declaration. It saves the definition of each scalar in both .rel and .dom, because top level scalar is considered at the same level as a relation and at the same time can be used as a virtual domain.

```
>declare s1 integer;
```

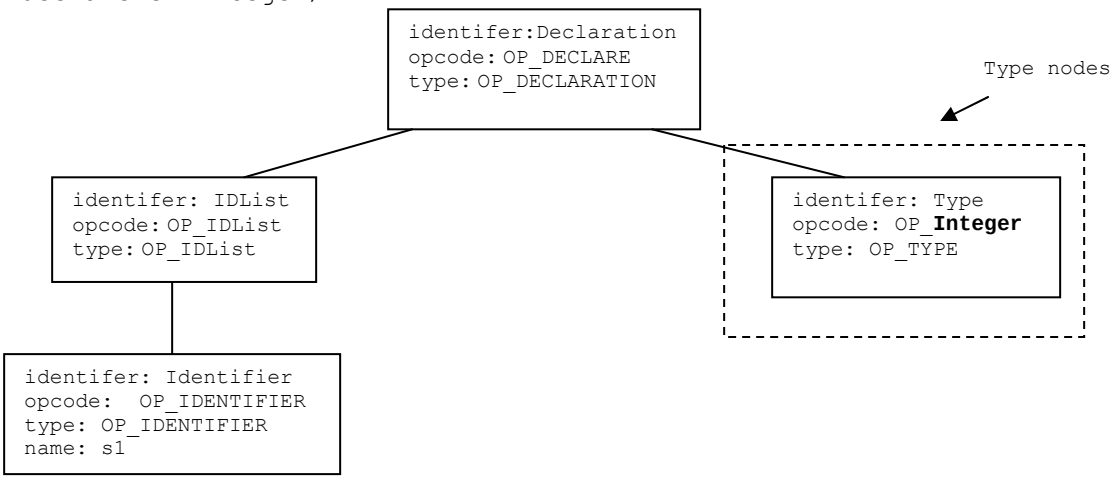


Figure 4.4.1 An example of the top level scalar declaration tree

The following is a flowchart of the function **DeclareDeclaration**(node):

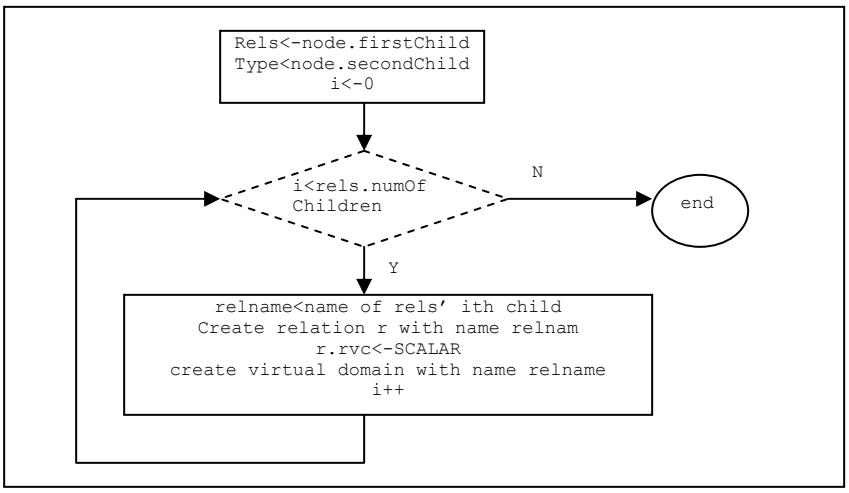


Figure 4.4.2 the function **DeclareDeclaration**

Besides the implementation of declaration of top level scalar, there are some other modification made where is needed. For example, when we print a top level scalar, we treat it like a relation; the system will implicitly convert it to a relation which has the same domain name as its relation name. For example,

```
>pr s1;
+-------------+
| s1          |
+-------------+
| 1           |
+-------------+
relation s1 has 1 tuple
```

the top level scalar is created to work with relational algebra and domain algebra. These are several examples introduced in section 3.3. Since it can work at both relation level and domain level, we have also made some changes in the main function interpreter where a top level scalar can be used. We take advantage of virtual domain so that we make the smallest possible impact on the existing system.

4.5 Implementation of the Substring Function

Substring function is used with relation algebra and domain algebra. A virtual domain can be defined using Substring. To implement substring function, we need introduce its syntax in the parser, and we need to build a new function **actSubstr** in the function **Actualizer** where virtual domain is declared and actualized.

4.5.1 Syntax of the substring function

Substring function has similar syntax with Java substring function; we can use it as the following example:

>let sub1 be substr(z,0,1);

The two integer type parameters have the same meaning as Java substring function.

The syntax is the following:

<SUBSTRING> (<Identifier> , <ILITERAL> (, <ILITERAL>)?)

< SUBSTRING> : "substr" | "substring"

<ILITERAL>: <INTEGER_LITERAL> | <IDENTIFIER>

Among the parameters, x is an identifier that can represent a domain, virtual domain or top level scalar. X has the type of string. Parameters a and b are integers, a indicates the start position in a string, b indicates one position after the end position in the string. The second integer b can be omitted just like Java does with the substr function, which means that the substring will start at the first position and continue to the end of the string.

Example in Figure 3.4.1

```
let sub1 be substr(z,0,1)
```

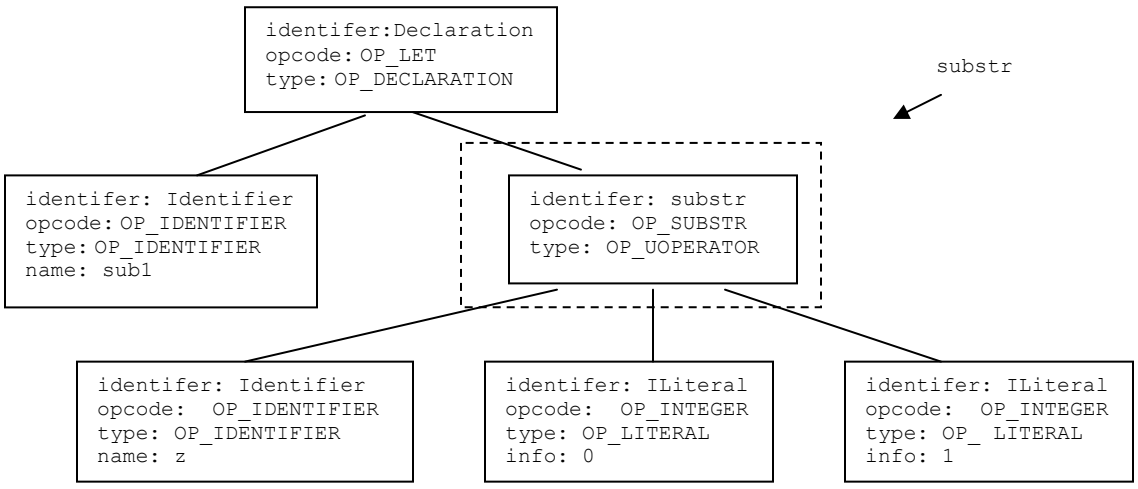


Figure 4.5.1 An example of the **substring** function tree

4.5.2 Implementation of the substring function

Upon implementation, we need to add substr or substring keyword to the parser, so that it will recognize the function. We also need to add OP_SUBSTR to Constants class to indicate the operation of substring.

The function **actSubstr**(SimpleNode node)

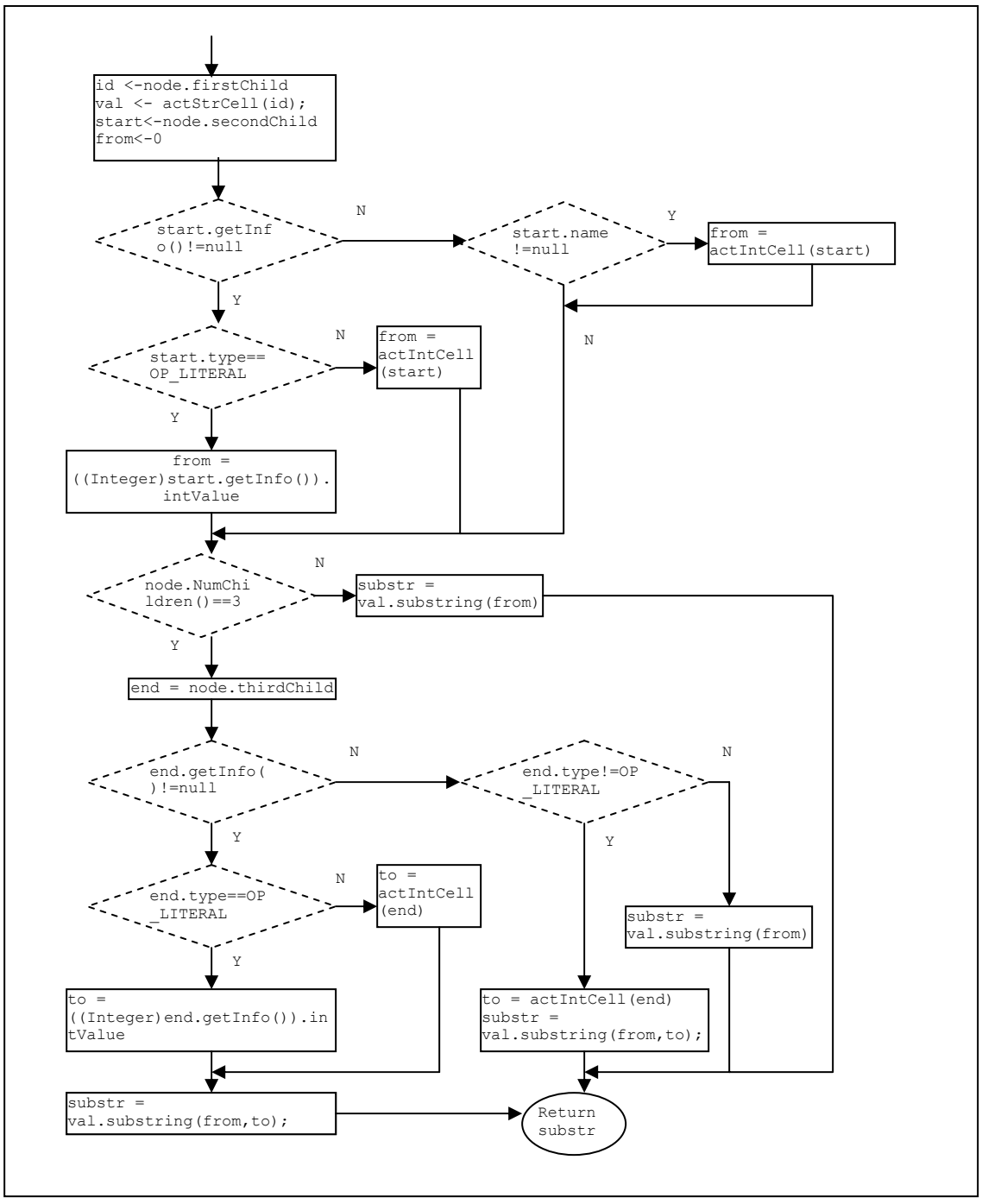


Figure 4.5.2 the **actSubstr** function

Chapter 5 Conclusions

We will give a summary of the work that has been accomplished in this thesis, followed by a discussion of future work.

5.1 Summary

What we contribute to the current JRelix in this thesis are the following new features:

- Grep Command

The grep command works on a relation and will return a relation as the result. The purpose of the grep command is to search a particular regular expression pattern in the database tables and return the exact match values with information such as in which attribute and position they are found. In addition to the pattern and relational expression (e.g. relation name), the grep command can take two parameter lists. All the parameters in these two lists will show up in the result relation as attributes. One parameter list can have up to 4 parameters to indicate the attribute, type of attribute, position and value of where the match is found. The other parameter list is a user defined parameter list. Each parameter in this list also appears in the pattern. It represents part of the match values.

- Union Type

We implement type polymorphism in a relational database. Briefly, union type means that an attribute can have more than one type and that these types can be primitive or complex. They can be defined on other previously defined types. Union type provides some flexibility in the rigid schema definition in a relational database and this may be very useful when dealing with semistructured data.

- Top Level Scalar

A Top level scalar is a primitive type variable that can be declared at the same level as a relation. It can be treated equally as virtual domain or as a singleton unary relation. We can use it with nested relations as shown in the example of Figure 3.3.5.

- Substring Function

The substring function working with domains of type of string. It gets parts of a string value. Like other domain algebra functions, it can be used in any calculation with domains.

5.2 Future work

- Grep

We can take a relation and a top level scalar as the search string, but it is not treated as a real pattern. For example, we have a top level scalar that is defined as string value "he.*o" and we want to find match with string "hello". It does not return a result because "he.*o" is not treated as a pattern that will match "hello", "heo" or any string that has "he" and "o" and the length of 0 to many any characters in between. It is considered as "." and "*", and is not taken as a wild card. We probably want both ways of interpretation. If we want to treat it as real pattern, we need to have a new type "pattern". Every domain with type of pattern will be treated automatically as a regular expression pattern. In addition, we can create a new function "toPattern", which will convert a string toa regular expression pattern.

Currently we use a relation as a container to store a set of strings that we want to search. The number of attributes that the relation has does not matter. We treat each cell equally as one of the search string. Alternatively, we can cross the boundary of the attributes to concatenate them as one string. For example, relation R has two attributes A and B. In a tuple, the value of A is "hello", the value of B is "there". In the current implementation, the search string will be "(hello)|(there)", but we can interpret it as "hello"and "there" and make the search program to find any two attributes which have "hello" in the first attribute and "there" in the second attribute. If we do this, we also need to consider how we output these pair of attributes.

Another thing that might be interesting is to grep in a nested relations. The current implementation of the grep does not look into the nested domain, because we design it to work with relational algebra. It might also be useful to have a grep that can work as domain algebra operator. Let's take a look at the following example: suppose a relation has an attribute with name of address. It is a union type domain, it can be string, or a nested relation that has street address, and zipcode as its attributes. Let's say we want to find all the tuples that have names like "cote", the current grep operator will only look within the flat domain, it will not look within the nested domain, so it will miss some of the matches. Of course we can write the command to recursively grep on the nested

domain, but it might be desirable to have the ability to grep into nested relations directly. We might also want to integrate path expression with the grep operator in order to search particular levels of the nesting as lorel[AQMWW97] does. For instance, we might want to have the ability to say in the where clause:

where address (.streeAddress)? grep '.*cote.*'

Bibliography

[AA04] Tony Abou-Assaleh and Wei Ai, Faculty of Computer Science, Dalhousie University, “Survey of Global Regular Expression Print (GREP) Tools” March 2, 2004

[ABC+76] M.M. Astrahan, M.W. Blasgen, D.D. Chamberlin, K.P. Ewaran, J. Gray, P.P. Griffiths, W.F. King, R.A. Lorie, P.R. McJones, J.W. Mehl, G.R. Putzolu, I.L. Traiger, B.W. Wade, and V. Watson. System R: *Relational Approach to Database Management*. ACM Trans. on Database Systems, 1(2):97--137, 1976.

[Abiteboul97] 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.

[ABS00] S. Abiteboul, P. Buneman, and D. Suciu, “Data on the Web : From Relations to Semistructured Data and XML”, Morgan Kaufmann, 2000.

[AC75]Alfred V. Aho, Margaret J. Corasick, "Efficient string matching: an aid to bibliographic search", *Commun. ACM*, 18, #6, June 1975, pp. 333-340.

[ACC+97] Serge Abiteboul, Sophie Cluet, Vassilis Christophides, Tova Milo, Guido Moerkotte, Jérôme Siméon “Querying Documents in Object Databases”, 1997. Int. J. on Digital Libraries

[Aho90] A.V. Aho. 1990. Algorithms for finding patterns in strings. In Handbook of Theoretical Computer Science . *Algorithms and Complexity*, Vol. A, ed. by J. van Leeuwen, Elsevier, Amsterdam, The Netherlands, pp. 255.300.

[AQMWW97] 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.

[AWK88]Alfred V. Aho, Brian W. Kernighan, Peter J. Weinberger, “The AWK Programming Language” Addison Wesley (January 1, 1988)

[BBW92] Val Breazu-Tannen, Peter Buneman, and Limsoon Wong. “Naturally embedded query languages.” In J. Biskup and R. Hull, editors, *LNCS 646: Proceedings of 4th International Conference on Database Theory, Berlin, Bermany, October, 1992*, pages 140-154. Springer-Verlag, October 1992.

[BCF+04] S. Boag, D. Chamberlin, M.F. Fern´andez, D. Florescu, J. Robie, and J. Simeon. “XQuery 1.0: An XML Query Language.” *World Wide Web Consortium*, October 2004. http://www.w3.org/TR/xquery/

[BDHS96] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. "A query language and optimization techniques for unstructured data". In *Proceedings of ACMSIGMOD International Conference on Management of Data*, pages 505-516, Montreal, Canada, June 1996.

[BDS95] Peter Buneman, Susan Davidson, and Dan Suciu. "Programming constructs for unstructured data". In *Procecclings of 5th International Workshop on Database programming Languages,* Gubbio, Italy, September 1995.

[BFS00] Peter Buneman, Mary Fernandez, Dan Suciu "UnQL: a query language and algebra for semistructured data based on structural recursion" The VLDB Journal — The International Journal on Very Large Data Bases Volume 9 , Issue 1 (March 2000) Pages: 76 - 110 ISSN:1066-8888

[BLS+94] P. Buneman, L. libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87-96, March 1994.

[BM77] BOYER R.S., MOORE J.S., 1977, "A fast string searching algorithm". *Communications of the ACM*. 20:762-772.

[BNTW95] Peter Buneman, Shamim Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with collection types. *Theoretical Computer Science*, 149:3-48, 1995.

[BPM+04] Tim Bray, Textuality and Netscape, Jean Paoli, Microsoft, C. M. Sperberg-McQueen, W3C, Eve Maler, Sun Microsystems, Inc., François Yergeau, Third Edition, W3C Recommendation 04 February 2004

[Brennan91] Mike Brennan, "mawk", 1991, http://foldoc.doc.ic.ac.uk/foldoc/foldoc.cgi?mawk

[Brzozowski64] Janusz A. Brzozowski, Derivatives of Regular Expressions, Journal of the ACM (JACM), v.11 n.4, p.481-494, Oct. 1964

[Buneman97] Peter Buneman, "Semistructured data", *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems,* Tucson, Arizona, United States Pages: 117 – 121, 1997 ISBN:0-89791-910-6

[CCGJ+92]Crochemore, M., Czumaj A., Gasieniec L., Jarominek S., Lecroq T., Plandowski W., Rytter W., 1992, Deux méthodes pour accélérer l'algorithme de Boyer-Moore, in Théorie des Automates et Applications, Actes des 2e Journées Franco-Belges, D. Krob ed., Rouen, France, pp 45-63, PUR 176, Rouen, France.

[CD92] Sophie Cluet and Claude Delobel. "A general framework for the optimization of object oriented queries". In M. Stonebraker, editor, Proceedings *ACM-SIGMOD InternationalConference on Management of Data,* pages 383-392, San Diego, California, June 1992.

Bibliography

[CD99] James Clark, Steve DeRose, “XML Path Language (XPath)” Version 1.0, W3C Recommendation 16 November 1999, http://www.w3.org/TR/xpath

[Codd70] E.A. Codd. “A relational model for large shared databanks”. *Communications of the ACM,* 13(6):377--387, 1970

[Codd71] E.F. Codd, "Normalized Data Base Structure: A Brief Tutorial", in [SIGFIDET 71].

[Codd72] E.F. Codd, "Further Normalization of the Data Base Relational Model", in R. Rustin (ed.), *Data Base Systems* (Courant Computer Science Symposia 6), Prentice-Hall, 1972. Also IBM Research Report RJ909.

[CR91] Debra Cameron , Bill Rosenblatt, “Learning GNU Emacs”, O'Reilly & Associates, Inc., Sebastopol, CA, 1991 . See also http://www.gnu.org/software/emacs/emacs.html

[DFFS98] A.Deutsch, M.Fernandez, D.Florescu, A.Levy, and D.Suciu. XML-QL: A query language for XML, 1998. http://www.w3.org/TR/NOTE-xml-ql

[DFH+99] Eduard Derksen (CSCIO), Peter Fankhauser (GMD-IPSI), Ed Howland (DEGA), Gerald Huck (GMD-IPSI), Ingo Macherius (GMD-IPSI), Makoto Murata (Fuji Xerox), Michael Resnick (Object Design, Incorporated),Harald Schöning (Software AG) “XQL (XML Query Language)” August 1999. http://www.ibiblio.org/xql/xql-proposal.html

[DR97] Dale Dougherty, Arnold Robbins, “sed & awk”, Second Edition,1997

[Eckel00]Bruce Eckel, “Thinking in Java” ,*Pearson Education* 2000

[Ellard97] Dan Ellard, http://www.eecs.harvard.edu/~ellard/Q-97/HTML/root/node45.html

[FG85] Patrick C. Fischer, Dirk Van Gucht, “Determining when a Structure is a Nested Relation” *VLDB 1985*: 171-180

[Findutils05] GNU Findutils http://www.gnu.org/software/findutils/findutils.html

[Friedl97] Jeffrey E. F. Friedl, “Mastering Regular Expressions; Powerful Techniques for Perl and Other Tools” 1st Edition January 1997

[Glantz57] Herbert T. Glantz 1957. On the recognition of information with a digital computer. *Journal of the ACM*, Vol. 4, No. 2, pp. 178–188.

[GSS04] Rick Greenwald, Robert Stackowiak, Jonathan Stern, Oracle Essentials, 3e: Oracle Database 10g (Paperback) O’RELLY 2004

[GNUgrep05] GNU grep, http://www.gnu.org/software/grep/grep.html

[Guo05] Fan Guo, "Implementing Attribute Metadata Operators to Support Semistructured Data", School of Computer Science, McGill University, January 2004. http://www.cs.mcgill.ca/~tim/cv/theses/fGuoThesis.ps.gz

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

[HH96] Michael Hauben, Ronda Hauben, "On the Early History and Impact of Unix, Tools to Build the Tools for a New Millenium", 1996, http://www.columbia.edu/~rh120/ch106.x09

[Horspool80] Horspool R.N., 1980, "Practical fast searching in strings", Software - Practice & Experience, 10(6):501-506.

[Hume88] Andrew Hume. 1988. A tale of two greps. *Software – Practice and Experience*, Vol. 18, No. 11, pp. 1063–1072

[IEEE03a] The IEEE and The Open Group. 2003. Regular Expressions. *The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2003 Edition, Base Definitions Volume*.

[IEEE03b] The IEEE and The Open Group. 2003. Utilities: grep. *The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2003 Edition, Shells and Utilities Volume*.

[ISO2002] (ISO-ANSI Working Draft) XML-Related Specifications (SQL/XML) http://xml.coverpages.org/SQLX-5wd-14-xml-2002-08.pdf

[JS82] B . Jaeschke, H .J . Schek . "Remarks on the algebra of non first normal form relations" . Proc. list ACM PODS, 124—138 (1982) .

[KD71] K. Thompson, D. M. Ritchie, "UNIX PROGRAMMER'S MANUAL" First edition, November 3, 1971, command part2, page 7-13. http://cm.bell-labs.com/cm/cs/who/dmr/man12.pdf

[KMP77] Knuth D.E., Morris (Jr) J.H., Pratt V.R., 1977, Fast pattern matching in strings, SIAM Journal on Computing 6(1):323-350.

[KR87] Karp R.M., Rabin M.O., 1987, "Efficient randomized pattern-matching algorithms". IBM J. Res. Dev. 31(2):249-260.

[Lecroq92]Lecroq T., 1992, "A variation on the Boyer-Moore algorithm", *Theoretical Computer Science* 92(1):119--144.

Bibliography

[Lecroq95]Lecroq, T., 1995, “Experimental results on string matching algorithms”, *Software - Practice & Experience* 25(7):727-765.

[Libes95] Libes, D., "Exploring Expect: A Tcl-Based Toolkit for Automating Interactive Applications", O'Reilly & Associates, Inc., pp. 602, January 1995. See also http://expect.nist.gov/

[LR98] Linda Lamb, Arnold Robbins, Learning the vi Editor, Sixth Edition, November 1998

[MAG+97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. “Lore: A database management system for semistructured data”. Technical report, Stanford University Database Group, February 1997.

[Mak77] A. Makinouchi. “A consideration on normal form of not necessarily normalised relation in the relational data model”. In *VLDB*'77, pp 447--453.

[MB90] Tony Mason , Doug Brown, Lex & yacc, O'Reilly & Associates, Inc., Sebastopol, CA, 1990. See also http://www.mkssoftware.com/products/ly/

[Merrett84] T. H. Merrett, “Relational Information Systems”, Reston Publishing Co., Reston, VA, 1984.

[Merrett00] T. H. Merrett, Computer Science 308-612A Database Systems, Lecture notes, http://www.cs.mcgill.ca/~cs612/homepage.html

[Merrett03] T. H. Merrett, “A Nested Relation Implementation for Semistructured Data”, School of Computer Science, McGill University, December 2003. http://www.cs.mcgill.ca/~tim/semistruc/recnest.ps.gz

[MKSawk] MKS awk command manual http://www.mkssoftware.com/docs/man1/awk.1.asp

[MKSegrep]] MKS awk egrep manual http://www.mkssoftware.com/docs/man1/grep.1.asp

[MySql] MySQL Reference Manual, 3.3.4.7. Pattern Matching http://dev.mysql.com/doc/mysql/en/pattern-matching.html

[Nudelman05] Mark Nudelman, “The LESS” http://www.greenwoodsoftware.com/less/

[ODMG 3.0] Edited by R. G. G. Cattell, Douglas K. Barry, Mark Berler, Jeff Eastman, David Jordan, Craig Russell, Olaf Schadow, Torsten Stanienda, and Fernando Velez; “The Object Database Standard:ODMG 3.0”. *Morgan Kaufmann*, San Mateo, California, 2000. http://www.odmg.org/

[Ousterhout94] John K. Ousterhout, Tcl and the Tk toolkit, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1994

[Paxson00] Vern Paxson, "GNU flex" http://freeware.sgi.com/fw-5.3/fw_GNUflex/GNUflex.html

[Polymorphism05] webopedia, "term: Polymorphism05", 2005, http://www.webopedia.com/TERM/p/polymorphism.html

[RCF00] Jonathan Robie (Software AG), Don Chamberlin (IBM Almaden Research Center), Daniela Florescu (INRIA), "Quilt: an XML Query Language", 31 March 2000 http://www.almaden.ibm.com/cs/people/chamberlin/quilt_euro.html

[Rischert03] Writing Better SQL Using Regular Expressions *By Alice Rischert* http://www.oracle.com/technology/oramag/webcolumns/2003/techarticles/rischert_regexp_pt1.html

[Robbins98] Arnold D. Robbins, "Gawk: The GNU AWK Users' Guide, 2nd edition", Free Software Foundation , 1998. Also see: http://www.gnu.org/software/gawk/

[Rossum] http://www.python.org/~guido/

[Shienbrood92] Eric Shienbrood, UC Berkeley, unix more. http://unixhelp.ed.ac.uk/CGI/man-cgi?more

[SHWK76]Michael Stonebraker , Gerald Held , Eugene Wong , Peter Kreps, "The design and implementation of INGRES", ACM Transactions on Database Systems (TODS), v.1 n.3, p.189-222, Sept. 1976

[SQL99] ANSI/ISO/IEC International Standard (IS)
Database Language SQL—Part 2: Foundation (SQL/Foundation)
http://www.ncb.ernet.in/education/modules/dbms/SQL99/ansi-iso-9075-2-1999.pdf

[TF86] Stan J. Thomas, Patrick C. Fischer: "Nested Relational Structures". *Advances in Computing Research* 3: 269-307 (1986)

[Thompson68] Ken Thompson, Bell Telphone Labs, Inc., Murray Hill
"Regular expression search algorithm", *Communications of the ACM* archive Volume 11 , Issue 6 (June 1968) table of contents Pages: 419 - 422

[TMD92] J. Thierry-Mieg and R. Durbin. Syntactic definitions for the ACeDB data base manager. Technical Report MRC-LMB xx.92, MRC Laboratory for Molecular Biology, Cambridge, UK, 1992.

[Todd76] S.Todd. The PeterLee Relational Test Vehicle. *IBM Systems Journal* vol. 15 no 4 1976

Bibliography

[Wall00] Larry Wall , Programming Perl, O'Reilly & Associates, Inc., Sebastopol, CA, 2000

[Wikipedia05] Wikipedia, "Boyer-Moore string search algorithm", 2005, http://en.wikipedia.org/wiki/Boyer-Moore_string_search_algorithm

[Xie05] Jiantao Xie, "Text Operators in a Relational Programming Language" School of Computer Science, McGill University, January 2005. http://www.cs.mcgill.ca/~tim/cv/theses/xieThesis.pdf.gz

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

[Yuan98] Zhongxia Yuan, "Java Implementation of the Domain Algebra for Nested Relations", 1998. http://www.cs.mcgill.ca/~tim/cv/theses/yuan.ps.gz

Appendix: partial JRelix syntax

Tokens:

```
< ASSIGN : "<-" >
< DOMAIN : "domain" >
< GREP: "grep" >
< OR : "|" >
< UNIVERSAL : "univ" | "universal" >
< TYPE : "type" >
< ATTRIBUTE : "attr" | "attribute" >
< BOOLEAN : "bool" | "boolean" >
< SHORT : "short" >
< INTEGER : "intg" | "integer" >
< LONG : "long" >
< STRING : "strg" | "string" >
< FLOAT : "float" | "real" >
< DOUBLE : "double" >
< NUMERIC : "number" >
< TEXT : "text" >
< STATEMENT : "stmt" | "statement" >
< EXPRESSION : "expr" | "expression" >
< COMP : "comp" | "computation" >
< LETTER : ["a"-"z", "A"-"Z"] >
< OTHERS : ["_", "'"] >
< DOTUS : [".", "_"] >
< DIGIT : ["0"-"9"] >
< INTEGER_LITERAL : (<DIGIT>)+ (["s","S","i","I","l","L"])? >
< FLOAT_LITERAL :
            (<DIGIT>)+ "." (<DIGIT>)* (<EXP>)? (["f", "F", "d", "D"])?
                |
                "." (<DIGIT>)+ (<EXP>)? (["f", "F", "d", "D"])?
                |
                (<DIGIT>)+ <EXP> (["f", "F", "d", "D"])?
                |
                (<DIGIT>)+ (<EXP>)? ["f", "F", "d", "D"] >
< NUMERIC_LITERAL : "\"" (["+","-"]) (<DIGIT><DIGIT><DIGIT>) ["+", "-"]
(<DIGIT>)+ ["N"] "\"" >
< STRING_LITERAL : "\"" (~["\"","\n","\r"])* "\"" >
```

```
< AND : "and" | "&" >
< CAT : "cat" >
< EVAL : "eval" >
< QUOTE : "quote" >
< TRANSPOSE : "transpose" >
< NOP : "nop" >
< IJOIN : "ijoin" | "natjoin" >
< UJOIN : "ujoin" >
< DJOIN : "djoin" >
< SJOIN : "sjoin" >
< LJOIN : "ljoin" >
< RJOIN : "rjoin" >
< DLJOIN : "dljoin" >
< DRJOIN : "drjoin" >
< ICOMP : "icomp" | "natcomp" >
< EQJOIN : "eqjoin" >
< GTJOIN : "gtjoin" >
< GEJOIN : "gejoin" | "sup" | "div" >
< LTJOIN : "ltjoin" >
< LEJOIN : "lejoin" | "sub" >
< IEJOIN : "sep" | "iejoin" >
< MAX : "max" >
< MIN : "min" >
< POW : "**" >
< PLUS : "+" >
< MINUS : "-" >
< MUL : "*" >
< DIV : "/" >
< MOD : "mod" >
< NOT : "not" | "!" >
< LET : "let" >
< SUBSTRING : "substr" | "substring" >
```

Domain declaration:

```
<DOMAIN> <IDList> <Type> (<OR> <Type>)*
<Type> :
      <UNIVERSAL>|<TYPE>|<ATTRIBUTE>|<BOOLEAN>|<SHORT>|<INTEGER>|
      <LONG>|<FLOAT>|<DOUBLE>|<NUMERIC>|<STRING>|<TEXT>|<STATEMENT>|
```

```
      <EXPRESSION>|<COMP> "(" [<IDList>] ")" | "(" <IDList> ")" |
      <Identifier>
<IDList> : <Identifier> ( "," <Identifier> )*
<Identifier> : (<DOTUS>)? <LETTER> (<LETTER>|<DIGIT>|<OTHERS>)*
```

Relation declaration:

```
<RELATION> <IDList> "(" <IDList> ")" [ <Initialization> ]
<Initialization> : <ASSIGN> "{" <ConstantTupleList> "}" | <Identifier>
<ConstantTupleList> : [  <ConstantTuple> ( "," <ConstantTuple> )* ]
<ConstantTuple> : "(" <Constant> ( "," <Constant> )* ")"
<Constant> : <Literal>|"{" <ConstantTupleList> "}"
<Literal> : <NULL> | <DC> | <DK> | <TRUE> | <FALSE> | ("+"|"-")? |
<INTEGER_LITERAL> | <FLOAT_LITERAL> | <NUMERIC_LITERAL> |
<STRING_LITERAL>
```

Top level scalar declaration:

```
<DECLARE> <IDList> <DType> [ <TInitialization> ]
<DType> :
<BOOLEAN>|<SHORT>|<INTEGER>|<LONG>|<FLOAT>|<DOUBLE>|<NUMERIC>|<STRING>|<
TEXT>
<TInitialization> : <ASSIGN> <Literal>|<Expression>
<Expression> : <Disjunction>
<Disjunction> : <Conjunction> ( <OR> <Conjunction> )*
<Conjunction> : <Comparison>  ( <AND> <Comparison> )*
<Comparison> : <Concatenation> (<ComparativeOperator> <Concatenation>)?
<ComparativeOperator> : <EQ> | <NEQ> | <GT> | <LT> | <GE> | <LE>
<Concatenation> : <MinMax>  (<CAT> <MinMax> )*
<MinMax> : <Summation> (  <MinMaxOperator> <Summation>   )*
<MinMaxOperator> : <MIN> | <MAX>
<Summation> : <JoinExpression> ( <AdditiveOperator> <JoinExpression>)*
<AdditiveOperator> : <PLUS> | <MINUS>
<JoinExpression> : <Projection>   (<JoinOperator> <Projection>
                |
                "[" <ExpressionList> ":" <JoinOperator> (":")?
      <ExpressionList> "]" <Projection> )*
<Projection> : <Projector> (( <IN> | <FROM> ) <Projection>|<Selection>)
             |
             <Selection>
<Projector> : [ <QuantifierOperator> ] "[" ( <ExpressionList> )? "]"
<Selection> : <Selector> | <QSelector> | <Term>
```

Appendix

```
<Selector> : ( <WHERE> | <WHEN> ) <Expression> ( <IN> | <FROM> )
<Projection>  | <EDIT> [ <Projection> ] | <ZORDER> <Projection>
<QSelector> : <QUANT> [( <WHERE> | <WHEN> ) <Expression>]
   ( <IN> | <FROM> ) <Projection>
<Term> : <Factor>   ( <MultiplicativeOperator> <Factor>  )*
<Factor> : <UnaryOperator> <Factor> | <Power>
<UnaryOperator> : <PLUS> | <MINUS> | <NOT>
<MultiplicativeOperator> : <MUL> | <DIV> | <MOD>
<Power> : <Primary>    (<POW> <Power> )*
<Primary> : <Identifier> | <Literal> | <QuantifierOperator>
       | <ArrayElement>
       | <PositionalRename> <Cast> | "("  <Expression> ")"
       | <Pick>  | <AttribsOf> | <grep> | <substr>
       | <Quote> | <Transpose> | <Function> | <IfThenElseExpression>
       | <VerticalExpression>
<ExpressionList>: <Expression> ("," <Expression>)*
<JoinOperator> : <NOP> | <MuJoin> | [ <NOT> ] <SigmaJoin>
<MuJoin> : <IJOIN> | <UJOIN> | <DJOIN> | <SJOIN> | <LJOIN> |
        <RJOIN> | <DLJOIN> | <DRJOIN>
<SigmaJoin> : <ICOMP> | <EQJOIN> | <GTJOIN> | <GEJOIN> | <LTJOIN>
| <LEJOIN> | <IEJOIN>
<GREP> ("("<GIDList>")")? <Literal>|<Identifier> <IN> <Selection>
<GIDList> : <LIDList>( ";" <IDList> )?
<LIDList> :   (<Identifier> "," <Identifier> )*)?
```

Virtual domain declaration:

```
<LET> <identifier> be <Expression>
```

Substring function:

```
<SUBSTRING> "(" <Identifier> "," <ILITERAL> ("," <ILITERAL>)? ")"
```