DATA NETWORK LANGUAGE DESIGN

By
Xiaoning Yan, B.Sc.

A Thesis
Submitted to the School of Graduate Studies
in partial fulfilment of the requirements for the degree of

Master of Science
Department of Computing and Software
McMaster University

MASTER OF SCIENCE(2000) McMaster University
(Computing and Software) Hamilton, Ontario, Canada

TITLE: Data Network Language Design

AUTHOR: Xiaoning Yan
B.Sc. Shenyang Institutue of Aeronautial Engineering
Shenyang, P.R.China

SUPERVISOR: Dr. David L. Parnas

NUMBER OF PAGES: x, 81

Abstract

> "The effect of computer networks on our ability to access information cannot be overstated. Today, anyone with access to the Internet can quickly view or retrieve any files that any other Internet user has chosen to make available [2]."

When we use this access to do a search, we find that often the results of the search are not easily used to what we want because the data stored on the Internet are unstructured and inconsistent.

This situation might be improved by setting up a new class of public computer networks (Data Networks), where the computers query each other to get answers that people need. A new language, called Data Network Language (DNL), is developed in this thesis as a query language for a Data Network. The data that are stored in a DNL database are structured data, such as sets, relations or mathematical expressions.

In this thesis, the syntax and grammar rules of DNL are developed along with a set of constructors that will be used for retrieval of data stored in DNL database. A DNL lexical analyser and parser (generated by LEX and YACC in UNIX) are also developed. Some illustrative examples are provided.

Acknowledgements

I would like to express my sincere thanks to my supervisor Dr. David L. Parnas for his guidance, encouragement and tremendous support throughout the whole procedure of preparation of this thesis. Thanks to him for giving me the opportunity to accomplish what I have long wanted to do in my life.

I would like to express my appreciation to Pui-Li Li and Feng Cui who are the other team members of the Data Network project for their efforts as part of the team on developing the constructors of DNL. I would like to thank Ou Wei for his helpful discussions and valuable suggestions.

I am grateful to Dr. William M. Farmer and Dr. Ridha Khedri for agreeing to review my thesis and for their thoughtful comments. I also would like to thank Dr. Martin von Mohrenschildt with his kind help on using LEX and YACC and Dr. Ridha Khedri's help on using LaTeX.

Special thanks to my husband, Zhifeng, for his love, encouragement and strong support.

Finally, I would like to thank the Communications and Information Technology Ontario (CITO), Natural Sciences and Engineering Research Council (NSERC), and Bell Canada for their financial support.

Contents

List of Figures

Chapter 1

Introduction

This chapter provides a brief introduction to the thesis including motivation, purpose, and organisation of the thesis.

1.1 Motivation

Networks play a very important role in our daily life. The Internet is designed for interactive use by people. For example, consider travel agents trying to organise a trip for a group of dog lovers. Usually, we will go to Internet and type in the key words "hotel, dog, Toronto". Some engines will return a few pages, but very often, the answer is unacceptable. Some engines even don't allow a user to type in more than one word, making it even harder to find the proper information. From this search, we can see that it takes quite some time to search with every engine, and the results of the search may not be what was it wanted. It is often actually easier to use phone calls to get the required information. This situation could be avoided by setting up a new class of public computer networks where the computers query each other to get the answers that people need. This network can handle various query conditions and do the computations on a distributed computer. We will call this new class of public networks, a "Data Network".

A data network is a widely distributed collection of computers, each one of which makes strictly structured information available to other computers on the network.

Users are able to formulate "queries" which will be answered by reliable computations using data from many of the computers on the network. Please refer to [19] for details about the data network that will be built.

The heart of the data network is the query language. We need a query language for the user and as an interface language between computers. In this project we will develop a kind of query language called Data Network Language (DNL) that can serve both as a query language and as an interface language.

SQL is a widely used query language for databases. Some of its characteristics are:

- The format of the data that it handles can not be described by mathematical expressions.
- The format of performing a query using SQL is more like writing a program than writing a mathematical expression.
- There is only one way of representing data in the database, namely, using a data table.
- Eight operators are used for data retrieval and manipulation. They are projection, product, difference, selection, union, join, divide and intersection [15].

DNL for Data Network has the following features that are different from SQL:

- The data it handles could be data described by listing values, described by a set given by comprehension or a TTS table [1].
- A query can have different formats such as "one-liner" expression (an DNL expression written in one sentence), sequential expressions (a sequence of one-liner DNL expressions separated by semicolon), or combination of one-liner and sequential expressions.
- There are several ways of representing data; refer to [18] on setting up databases.
- More operators can be used for easier queries and computations among relations (detail definitions are given in Chapter 3)

1.2 Purpose

The purpose of this thesis is to develop DNL, a new query language which can handle queries on DNL Databases with data described using relations. The following goals are to be achieved:

- Develop a set of built-in and non-built-in functions applying to relations.
- Develop the syntax of DNL.
- Generate a DNL parser that checks the syntax of DNL.
- Check the functions of DNL parser by printing out each DNL expression or subexpression that has been recognised.

1.3 Organisation of the Thesis

Chapter 2 provides the background describing a number of older tools and ideas that are needed to do the development. Chapter 3 describes the basic properties of DNL and detailed definitions of DNL syntax. Chapter 4 presents newly developed set of functions applying on relations. Chapter 5 provides the design of the DNL parser/syntax Checker. Chapter 6 provides a simple example of the use of DNL and DNL parser/syntax checker. Chapter 7 provides a brief user guide for using DNL parser. Chapter 8 summarizes the results of the design of DNL and the implementation of DNL parser/syntax checker, discusses the limitations, suggests future work and draws some conclusions.

Chapter 2

Background

DNL deals mainly with retrieving and using data stored in a DNL relational database. This chapter introduces many of the mathematical concepts needed for data manipulation on relational databases, and a number of older tools and ideas are introduced as the base of a DNL mathematical theory for further development.

2.1 Databases

Roughly speaking, a database is a collection of data and operations that represents some aspects of the real world. For example, in the database of a university, all the students, courses, professors and other staffs' information could be stored in a formal way so that the users can get the information whenever they need it [16]. A query is a request to get data from a database.

2.2 Relational Database and SQL

2.2.1 Relational Database

> "A modern relational database is viewed as presenting data as a collection of tables. A table consists of data logically arranged in columns and rows.

> Data in the table is logically related, with relationships defined between different tables in database [17]."

The data that DNL deals with are structured data (relations or TTS tables [1]) stored in a relational database.

2.2.2 SQL

SQL, Structured Query Language, is used to communicate with a database. It uses the terms "table", "row" and "column" for "relation", "tuple" and "attribute" respectively [17]. According to ANSI (American National Standards Institute), SQL is the standard language for relational database management systems. SQL statements are used to perform tasks such as updating data in a database, or retrieving data from a database [4].

DNL is also a query language that can be used to retrieve data from databases. SQL inspired many aspects of the way that DNL deals with data.

2.3 APL

The following quotation is taken from a web-site maintained and created by Special Interest Group on APL Programming Language of the ACM (Association for Computing Machinery).

> "APL (A Programming Language) is one of the most powerful, consistent and concise computer programming languages ever devised. The notation consists of a set of symbols (letters, numbers, punctuation, algebra, and special shapes), with a very simple set of rules (syntax) for putting them together to describe the processing of data. The data can be either numeric or literal (which includes words and text handling) [5]."

The data used in APL can be arranged in arrays. Like APL, DNL expressions are formed from concisely composed mathematical functions.

2.4 The web

The WWW (World Wide Web) has become a very important source of information all over the world. Information on many different subjects can be accessed using the WWW. WWW is viewed as a big container holding a large amount of data embedded in web presentations generated from web documents. An Internet browser such as Netscape or Internet Explorer is used to read these web representations.

2.5 Difference between Older Technologies and DNL

2.5.1 Differences

- WWW contains a collection of unstructured data. All the data in a database system are stored as a set of n-ary relations; DNL assumes data structured as a set of binary relations. The reason why binary relations are used in DNL is because binary relation is easy to be manipulated then n-ary relation. For example, we can compose two binary relations by using composition function, but the composition operation is complex for n-ary relations.

  Moreover, DNL is intended to enter many computers via a data network [19], and it is designed to be used (without human intervention) to compute facts from these widely distributed databases.

- SQL deals with n-ary relations which might have "NULL" entries when some items are not relevant or undefined. For example, in the data table of section 2.6, "Parnas" doesn't have a student identification number, which is a NULL value, because he didn't enroll or he has dropped out. The NULL value might cause some problem, such as, running SQL server with a service pack, if your database has tables with text columns that allow nulls, updating to tables with text may result in errors, changes have to be made in order to avoid this problem [21]. Many papers have been written discussing the meaning "NULL" in a relational

database. DNL deals with binary relations where the problem of "NULL" does not arise.

- SQL queries are written more like a program than a mathematical expression; when a query contains many tables or deals with many databases at the same time, DNL expressions can be very concise even for a complicated query, sometimes one line is enough. DNL queries are relational (mathematical) expressions that are evaluated in a way analogous to conventional mathematical expressions. Like APL, complex programs can be written as a "one-liner".

  The reason why DNL is syntactically better than SQL is that every DNL program can be written as one-liner expression (one sentence) which makes the program more concise, but not all the SQL program can be written in one sentence.

- One big difference between DNL and SQL is that in DNL, any computed relation can be used in the same way as any built-in relations as shown in following example.

2.5.2 Example

The following is an example expressed in both DNL and SQL to enable comparison focusing on the format of these two languages.

Example: A shopping list contains both Apples and Bananas, we want to find a single store that is going to give the lowest price for 2 pounds of apples and 3 pounds of bananas.

2.5.2.1 DNL Expression for Solving Above Problem

- Assume we have a relation

$$R_1 = \{(Store, (Item, (Price, Quantity)))|Store \in set\_of\_Store$$

$$\wedge (Item, (Price, Quantity)) \in Items \times (R^+ \times N)\}$$

- There are three ways of using DNL to solve the example above:

  (1) Figure 2.1 shows a "one-liner" expression in DNL.

  From the figure, we can see that the one-liner expression is very concise. As mentioned before, the reason why the "one-liner" expression is better over the other two formats is because it is easy to construct since each return value of a function can be the argument of next function. The whole expression could be more concise if we replace the name of each function as a symbol. For example, if we use " # " to represent "Union" and " $ " to represent "Join", instead of writing Union(Join(a,b),c), we can write #($(a,b),c).

  (2) Figure 2.2 shows a sequential program in DNL.

  From the figure, we can see that this is much like a program. But the advantage of this format is that it is very clear to see the return value of each function call if there are too many steps of using function calls.

  (3) Figure 2.3 shows combination of sequential expressions and "one-liner" expressions in DNL.

  From the figure, we can see that the combination format mixes together the advantages of both one-liner expression and sequential expressions. It is not only easy to read, but also easy to write.

2.5.2.2 SQL Program for Solving Above Problem

Figure 2.4 shows the SQL program. This program code has been compiled and executed correctly under the DB2 system.

2.6 N-ary Relation, Binary Relation and Table

The data used in DNL can be any of the following:

```
<8> OperatorOnFunction(Minimum,
  Range(
   <7> RangeMerge(
          <6> Rearrange(
          <5> Union(
             <3> ArithmeticComp(
                Restriction(
                   Rtemp <- Rearrange(
                   <2> Restriction
                   (R1,
                   GetAttributeName(R1, 1) member
                      <1> Intersection
                         (Domain
                            (Restriction
                              (R1, GetAttributeName(R1, 2.1) = 'Apple'
                                 && GetAttributeName(R1,2.2.2) >2)),
                         Domain
                            (Restriction
                              (R1, GetAttributeName(R1, 2.1) = 'Banana'
                                  && GetAttributeName(R1,2.2.2) >3))))
                   (1, (2.1, 2.2.1))),
                GetAttributeName(Rtemp,2.1) = 'Apple'), 2.2, *, 2),
     <4> ArithmeticComp(
        Restriction( Rtemp ,
           GetAttributeName( Rtemp 2.1) = 'Banana'),2.2, * , 3) ),
     ((1,2.1), 2.2)),1.1, Sum)));
```

Figure 2.1: One-liner Expression of DNL

```
R2 <- Restriction (R1, GetAttributeName(R1, 2.1) = 'Apple'
                && GetAttributeName(R1, 2.2.2) >2 );

R3 <- Restriction(R1, GetAttributeName(R1, 2.1) = 'Banana'
                && GetAttributeName(R1, 2.2.2) >3 );

R4 <- Intersection(Domain(R2), Domain(R3));

R5 <- Restriction(R1, GetAttributeName(R1, 1) member R4);

R6 <- Rearrange(R5, (1, (2.1, 2.2.1)));

R7 <- Restriction(R6, GetAttributeName(R6,2.1) = 'Apple');

R8 <- Restriction(R6, GetAttributeName(R6,2.1) = 'Banana');

R9 <- ArithmeticComp(R7, 2.2,*, 2);

R10 <- ArithmeticComp(R8, 2.2,*, 3);

R11 <- Union(R9,R10);

R12 <- Rearrange(R11,((1,2.1),2.2));

R13 <- RangeMerge(R12, 1.1, Sum);

OperatorOnFunction(Minimum, Range(R13));
```

Figure 2.2: Sequential Expression of DNL

```
R2 <- Rearrange(
          Restriction
          (R1,
          GetAttributeName(R1, 1) member
           Intersection
             (Domain
               (Restriction
               (R1, GetAttributeName(R1, 2.1) = 'Apple'
                        && GetAttributeName(R1, 2.2.2) >2)),
             Domain
               (Restriction
               (R1, GetAttributeName(R1, 2.1) = 'Banana'
                        && GetAttributeName(R1, 2.2.2) >3 ))))
          (1, (2.1, 2.2.1))),

R3 <- Restriction( Rearrange(R2, (1, (2.1, 2.2.1))),
                GetAttributeName(R2,2.1) = 'Apple');

R4 <- Restriction(Rearrange(R2, (1, (2.1, 2.2.1))),

                GetAttributeName(R2,2.1) = 'Banana');

R5 <- ArithmeticComp(R3, 2.2,*, 2);

R6 <- ArithmeticComp(R4, 2.2,*, 3);

R7 <- Union(R5,R6);

OperatorOnFunction(Minimum, Range(
        RangeMerge(Rearrange(R7,((1,2.1),2.2)),1.1,Sum)));
```

Figure 2.3: Combination of One-line and Sequential Expressions of DNL

```
create view v1(store,item,price) \
as \
        select t1.store,t1.item,t1.price*2 \
        from s_info as t1 \
        where (t1.item='apple') and (t1.quantity>2)
create view v2(store,item,price) \
as \
        select t1.store,t1.item,t1.price*3 \
        from s_info as t1 \
        where (t1.item='banana') and (t1.quantity >3)
create view v3(store,item,price,s,i,p) \
as \
        select v1.store,v1.item,v1.price,
        v2.store,v2.item,v2.price \
        from (v1 join v2 on v1.store = v2.store)
create view v4(store,price) \
as \
        select v3.store,v3.price+v3.p \
        from v3
select min(price) \
from v4
```

Figure 2.4: SQL PROGRAM

- An N-ary Relation

  Let $\{V_1, V_2, V_3,..., V_n\}$ be a family of sets. An n-ary relation R on $\{V_1, V_2, V_3,..., V_n\}$ is a subset of the Cartesian product $V_1 \times V_2 \times V_3 \times ... \times V_n$. The tuple $(x_1, x_2, x_3,..., x_n)$ is said to be in the relation R iff $(x_1, x_2, x_3,..., x_n) \in R$ [4, 7].

- A Binary Relation

  A binary relation is an 2-ary relation, that is, it is a subset of a Cartesian product of two sets: $V_1 \times V_2$.

- A Data Table

  A table consists of zero or more rows of data values. For a given table, each data entry contains exactly one scalar value or NULL (for missing, undefined, or not applicable). All the values in a given column have the same data type.

  For example, an example of a table with the attributes of "student last name" , "first name" and "student identification number", is represented as:

Main	Deric	2330
Wang	Hong	3667
Arvind	NULL	88
Parnas	David	NULL
Bond	James	007

- A TTS Table

  TTS table consists of a main grid and coordinate header grids. Each cell of the grids can be a tabular expression itself [1].

  Tabular expressions are multi-dimensional expressions which are often easier to read and understand than equivalent, more traditional scalar expressions. They are grouped into three different categories: atoms (constants or variables),

applications (use of functions or predicates with 1 or more arguments), and tables [1].

The TTS (Table Tool System) is an integrated, extensible system of tools to facilitate the use of tabular expressions in computer systems documentation [1].

2.7 Relation Algebra

Here we want to compare the basic relation algebra and the extended [1] relation algebra that is widely used in relational database system.

2.7.1 Basic Binary Relation Algebra

(1) Definition of empty and universal relation, taken from [6]:

- Empty relation:

$$\emptyset = \{ (x,y) \in V_1 \times V_2 \mid \text{false} \}$$

- Universal relation:

$$\{ (x,y) \in V_1 \times V_2 \mid \text{true} \}$$

(2) Definition of operators of relation algebra, taken from [7]:

Assume $R \subseteq V_1$ and $S \subseteq V_2$.

- CARTESIAN PRODUCT:

$$R \times S \subseteq \{ (x,y) \mid x \in R \wedge y \in S \}$$

- UNION:

$$R \cup S = \{(x,y) \mid (x,y) \in R \vee (x,y) \in S \}$$

[1] Extend here means that something based on other things that existed previously.

- INTERSECTION:

$$R \cap S = \{(x,y) \mid (x,y) \in R \wedge (x,y) \in S \}$$

- COMPLEMENT[2] :

  Let $R \subseteq V_1 \times V_2$,

$$comp(R,V_1, V_2) = \{(x,y) \mid (x,y) \in V_1 \times V_2 \wedge (x,y) \notin R\}$$

- COMPOSITION:

  Let $R \subseteq V_1 \times V_2$ and $S \subseteq V_2 \times V_3$ be relations, their product $R \circ S \subseteq V_1 \times V_3$ is given by:

$$R \circ S = \{ (x,z) \in V_1 \times V_3 \mid \exists\, y \in V_2 : (x,y) \in R \wedge (y,z) \in S \}$$

- INCLUSION:

$$R \subseteq S \Leftrightarrow \forall\, x,y: [(x,y) \in R \rightarrow (x,y) \in S]$$

2.7.2 The Extended Relation Algebra that is the basis of SQL

The extended relational algebra consists of a collection of high-level operators operating on relations. Each such operator takes either one or two relations as its input and produces a new relation as its output. Codd defined a very specific set of eight such operators, traditional set and special groups with four each [15], we select most of them as the base of definition of DNL functions. All the following operators are defined in Chapter 3, section 3.3.

(1) The traditional group

- UNION .
- INTERSECT.
- DIFFERENCE.

[2]Complement is relative to $V_1 \times V_2$. When we need the complement of a relation R, V_1 and V_2 must always be provided, i.e.Complement(R, V_1, V_2).

- PRODUCT.

(2) The special group

- RESTRICT.
- PROJECT.
- JOIN.
- DIVIDE.

The basic relational operators are the basis of relational computation. We want to define new operators such as extended ones as above or some ones that used in DNL based on those basic ones.

Chapter 3

Definitions of Built-in and Non-built-in Functions

This chapter provides the definitions of built-in functions[1] and non-built-in functions used in DNL. This is the core of DNL. An example of shopper's guide is also provided for illustrating the uses of the built-in functions.

Without built-in and non-built-in functions, computations among relations will not be performed and DNL will lose its value. These functions are defined based on the relation algebra and practical relational database systems.

3.1 Representation of Data used in DNL

As mentioned before, the data that DNL deals with are sets, relations, or mathematical expressions (predicate). Representation of a relation is discussed in this section.

Assume we have a relation

$$R_1 = \{(Store, (Item, (Price, Quantity)))|Store \in set\_of\_Store$$

$$\wedge(Item, (Price, Quantity)) \in Items \times (R^+ \times N)\}$$

There are several ways to represent this relation, they are:

[1] built-in functions are the functions that are constructed into language and are not user defined functions.

(1) Traditional Representation:

Enumerating all the elements in the relation is a traditional representation.

$$R_1 = \{(nofrills, (apple, (0.99, 300)), (fortino, (milk, (3.10, 300))),$$
$$(nofrills, (milk, (3.19, 150))), (fortino, (apple, (1.29, 250))),$$
$$(foodbasics, (milk, (2.99, 220))), (nofrills, (banana, (0.37, 200))),$$
$$(foodbasics, (apple, (1.39, 300))), (fortino, (banana, (0.56, 700)))\}$$

(2) Tabular Representation

Using a table to list all the elements of the relation with a heading indicating the meaning of each column.

The table representing R_1 has four columns, each column can be associated with its own attribute name in the first row.

Store	(Item	(Price	Quantity))
nofrills	(apple	(0.99	300))
fortino	(milk	(3.10	300))
nofrills	(milk	(3.19	150))
fortino	(apple	(1.29	250))
foodbasics	(milk	(2.99	220))
nofrills	(banana	(0.37	200))
foodbasics	(apple	(1.39	700))

(3) Tuple-index Representation

We use a *Tuple-index* to identify an element in each tuple in the relation. With a tuple-index, each component of a tuple can be easily indicated and used.

We have I = [1. . .n],

$$R \subseteq \prod_{i \in I} D_i,$$

$\prod$ represents for Cartesian product. i.e.

$$R \subseteq D_1 \times D_2 \times \cdots \times D_n$$

where D_i is a set

(1) Base Step:

Given n-tuple tp = (t_1,t_2, ...,t_n), and j $\in$ [1 ... n], j is a tuple-index of denoting component t_j of tp.

For example, for 4-tuple tp = (a,b,c,d). The tuple-index for a is 1, b is 2, c is 3 and d is 4.

(2) Inductive Step:

Let i $\in$ [1 ... n] be a tuple-index, for tp and tp.i = (C_1, ..., C_m) and j $\in$ [1 ... m].

then i.j is a tuple-index for tp and tp.i.j denotes C_j.

See example bellow.

For example: for R_1, the tuple-index of components of Store is 1, of components of Item is 2.1, of components of Price is 2.2.1, of components of Quantity is 2.2.2.

This is illustrated as

1	*2*		
1	2.1	2.2.1	2.2.2
Store	(Item,	(Price,	Quantity))

3.2 Overview

This section gives an overview of the function that are used in DNL and an illustration of their functionalities and uses. The next section explains those built-in and non-built-in functions. All the signatures associated with the following function are given in section 3.3.

3.2.1 Creating, Inserting or Deleting and Getting Data in DNL Database

Without creating relations and storing them into databases, no queries can be answered, it is just like cooking without any raw materials. So the following functions are used to create, insert and delete relations:

- Create
- Insert
- Delete

The following is used to get further information about a relation:

- GetAttributeName

3.2.2 Basic Operations Among relations

The following are the functions for computing or getting information from one or two relations:

- Product
- Union
- Intersection
- Difference
- Domain and Range
- Image and PreImage
- Cardinality
- Identity

- Index
- Restriction
- Join
- Composition

3.2.3 Advanced Operations Among relations

The following advanced functions are also used for faster and easier computation and manipulation among relations:

- RangeDivide
- RangeMerge
- Rearrange
- OperatorOnFunction
- CreateAbsSRF
- ArithmeticComp

3.2.4 Assignment

In order to allow assignment used in DNL, we have assignment function:

- Assignment function is for assigning a relation to a variable.

3.2.5 Reduction

We have reduction to simplify complex structure of an expression.

- Reduction function is for applying repeated operation to variables.

3.3 Definitions of Built-in Functions

The specifications of each built-in function are as following. An example is also provided for using the function.

3.3.1 Restriction(Set, Predicate)

This function restricts a set by a predicate to get a new set that is a subset of the original set.

Signature:

Set , Predicate $\rightarrow$ Set

Given a Set S and a predicate P over the elements on S,

Restriction(S,P) = $\{ x \mid x \in S \wedge P(x) \}$.

Example:

Given a set S = $\{ 1, 2, 3, 4, 5, 6, 7, 8\}$,

a predicate P: $x > 5$,

Restriction(S, P) = $\{ 6, 7, 8 \}$.

3.3.2 Domain(Relation)

This function is used to get the domain of a relation.

Signature:

Relation $\rightarrow$ Set

Given a relation R,

Domain(R) = $\{ x \mid \exists y : (x,y) \in R \}$.

Example:

Given a relation R = { (1,2), (3,4), (5, 6), (7,8) },

Domain(R) = { 1, 3, 5, 7 }.

3.3.3 Range(Relation)

This function is used to get the range of a relation.

Signature:

Relation → Set

Given a relation R,

Range(R) = { y | ∃ x: (x,y) ∈ R }.

Example:

Given a relation R = { (1,2), (3,4), (5, 6), (7,8) },

Range(R) = { 2, 4, 6, 8 }.

3.3.4 Image(Relation,Set)

This function is used to get the range of the given relation, restricted to a given domain.

Signature:

Relation, Set → Set

Given a relation R and a set S,

Image(R,S) = { y | ∃ x : (x,y) ∈ R ∧ x ∈ S }.

Example:

Give a relation R = { (1,2), (3,4), (5, 6), (7,8) },

a set S = { 1, 3, 5, 29},

Image(R,S) = { 2, 4, 6 }.

3.3.5 PreImage(Relation,Set)

This function is used to get the domain of the given relation restricted to a given range.

Signature:

Relation, Set $\rightarrow$ Set

Given a relation R and a set S,

PreImage(R,S) = { x | $\exists$ y : (x,y) $\in$ R $\wedge$ y $\in$ S }.

Example:

Given a relation R = { (1,2), (3,4), (5, 6), (7,8) },

a set S = { 2, 4, 10},

PreImage(R,S) = { 1, 3 }.

3.3.6 Product(Set, Set)

This function is used to get the Cartesian product of two sets.

Signature:

Set $\times$ Set $\rightarrow$ Relation

Given two sets S_1 and S_2,

Product(S_1,S_2) = {(x,y) | x $\in$ S_1 $\wedge$ y $\in$ S_2 }.

Example:

Given a set S_1 = { 1, 2, 3 } and a set S_2 = { 4, 5, 6 },

Product(S_1, S_2) = { (1,4), (1,5), (1,6), (2,4), (2,5), (2,6), (3,4), (3,5), (3,6)}.

3.3.7 Union(Set, Set)

This function is used to compute the union of two sets.

Signature:

Set , Set $\rightarrow$ Set

Given two sets S_1 and S_2,

Union(S_1,S_2) = $\{ x \mid x \in S_1 \vee x \in S_2 \}$.

Example:

Given a set S_1 = { 1,2,3,4 } and a set S_2 = { 4,5,6 },

Union(S_1, S_2) = { 1,2,3,4,5,6 }.

3.3.8 Intersection(Set, Set)

This function is used to compute the intersection of two sets.

Signature:

Set , Set $\rightarrow$ Set

Given two sets S_1 and S_2,

Intersection(S_1,S_2) = $\{ x \mid x \in S_1 \wedge x \in S_2 \}$.

Example:

Given a set S_1 = { 1,2,3 } and a set S_2 = { 3,2,6 },

Intersection(S_1, S_2) = { 2,3 }.

3.3.9 Difference(Set, Set)

This function is used to compute a set whose elements are in the first set but not in the second set.

Signature:

Set , Set $\rightarrow$ Set

Given two sets S_1 and S_2,

Difference(S_1,S_2) = { x | x $\in$ S_1 $\wedge$ $\neg$ (x $\in$ S_2) }.

Example:

Given a set S_1 = { 1,3,4 } and a set S_2 = { 4,5,6 },

Difference(S_1, S_2) = { 1,3 }.

3.3.10 Insert(Set, Element)

This function is used to insert one element into a designated set. Building up a new relation is usually done using this function to insert new tuples one by one.

Signature:

Set, Element $\rightarrow$ Set

Given a set S_1 and an element x,

Insert(S_1,x) = S_1 union { x }.

Example:

Give a Set S = { 1,2,3 },

Insert(S, 4) = { 1,2,3,4}.

3.3.11 Delete(Set, Element)

This function is used to delete one element from a designated set. This can be used for removing an old relation or some specific tuples.

Signature:

Set, Element $\rightarrow$ Set

Given a set S_1 and an element x,

Delete(S_1,x) = S_1 difference { x }.

Example:

Give a set S = { 1,2,3 },

Delete(S, 3) = { 1,2 }.

3.3.12 Identity(Set)

This function is used to compute the identity relation of a set.

Signature:

Set $\rightarrow$ Relation

Given a set S,

Identity(S) = { (x,x) | x $\in$ S }.

Example:

Given a set S = { 1,2,3 },

Identity(S) = { (1,1),(2,2),(3,3) }.

3.3.13 Cardinality(Set)

This function is used to compute the number of the elements in a set.

Signature:

Set $\rightarrow$ Integer

Given a Set S,

Cardinality(S) = $\sum_{x \in S}$ 1.

Cardinality(S) is undefined if S is infinite.

Example:

Give an set S = { 1,2,3 },

Cardinality(S) = 3.

3.3.14 Join(Relation, Relation)

This function is used to combine two relations as defined below using elements that are in the range of the first relation and in the domain of the second relation.

Signature:

Relation , Relation $\rightarrow$ Relation

Given two relations R_1: A $\times$ B and R_2: C $\times$ D,

$$Join(R_1,R_2) = \{ (x,(y,z)) \mid (x,y) \in R_1 \wedge (y,z) \in R_2 \}.$$

Example:

Given a relation R_1 = { (1,3), (6,7),(4,8) },

a relation R_2 = { (3,5),(9,10),(7,8) },

Join(R_1, R_2) = { (1,(3,5)), (6,(7,8)) }.

3.3.15 Composition(Relation, Relation)

This function is used to compose two relations.

Signature:

Relation , Relation $\rightarrow$ Relation

Given two relations R_1: A $\times$ B and R_2: C $\times$ D,

$$Composition(R_1,R_2) = \{ (x,z) \mid \exists\ y : (x,y) \in R_1 \wedge (y,z) \in R_2 \}.$$

Example:

Given a relation R_1 = { (1,3), (6,7),(4,8) },

a relation R_2 = { (3,5),(9,10),(7,8) },

Composition(R_1, R_2) = { (1,5), (6,8) }.

3.3.16 RangeDivide(Relation)

This function is used to reconstruct a relation whose range is a set of pairs.

Signature:

Relation → Relation

Let R be a relation, such that Range(R) $\subseteq V_1 \times V_2$, where V_1 and V_2 are sets,

RangeDivide(R) = { (x,y) | $\exists$ z : ((x,(y,z)) $\in$ R $\vee$ (x,(z,y)) $\in$ R }.

Example:

Given a relation R = { (a, (b,c)), (e,(f,g)), (h,(i,j)) },

RangeDivide(R) = { (a,b),(a,c) (e,f), (e,g), (h,i),(h,j) }.

3.3.17 RangeMerge(Relation, Tuple-index, Operator)

This function is used to apply particular operators to the range of a relation if the associated domains are the same.

Signature:

Relation, Tuple-index, OP → Relation

Given a Relation R_1, a Tuple-index ti, and an operator OP,

RangeMerge(R,ti,Op) = {(x, y) | $\exists$t $\in$ R_1 : y = OperatorOnFunction(Op, Range(Restriction(R1, t.ti = x))) }.

Example:

Given a relation R = { (a, 1), (a,2), (b,25), (b,7), (c,9), (c,34) },

RangeMerge(R, 1, Sum) = { (a,3), (b,32), (c,43) }.

3.3.18 Index(Set, Set, Relation)

The result of this function is a set of pairs where the first element is an element in the Index set(second set), the second element is a member of the first set.

Signature:

Set, Index_Set, Relation $\rightarrow$ Set

Given a set S, a Index Set I, Cardinality(I) = Cardinality(S) = n, a total order relation O on S. Let R $\subseteq$ I $\times$ S, we have

Index(S,I,O) = { (i,x), (j,y) $\in$ R | x O y $\Rightarrow$ i $<$ j }.

Example:

Given a set S = { a, c, b, d, g, r, z } and

a Index Set I = IN^+ = { x | x $\in$ integer $\wedge$ x $>$ 0 },

a total order relation : $<$ (LESS THAN),

Index(R,IN^+,$<$) = { (1,a), (2,b), (3,c), (4,d), (5,g), (6, r), (7, z) }.

3.3.19 Rearrange(Set, Template)

This function is used to reconstruct a relation by giving a template (see definition at following).

- Tuple-index (ti) see definition in section 3.1.
- Dot Operation($\bullet$)

Let i = (j,. . ., k,. . .,l) where j $\neq$ k $\neq$ l,

Let X = (x_1, . . ., x_j,. . ., x_k, . . ., x_l, . . ., x_n)

We have

$$X \bullet i = (x_j, \ldots, x_k, \ldots, x_l)$$

.

e.g. suppose tp = (x,y,u,w,z), ti= (2, 3, 5),

tp • ti = (y, u, z).

- template(tm)

  A template is a n-tuple of tuple indices or template and can be represented by following BNF [14],

  <template> ::= (<Tuple-index>

  | (<Tuple-index>, <template>)

  | (<template>, <Tuple-index>)

  | (<template>, <template>)

- Rearrange_Tuple

  Given an n-tuple X and a template tm = (tm_1,tm_2, . . .,tm_n) on X,

  (1) Base Case

  Rearrange_Tuple(X,(tm_1,tm_2, . . .,tm_n))

  = (X•tm_1,X•tm_2,. . .,X•tm_n)

  if tm has only one 1-tuple index.

  (2) Inductive Step

  Rearrange_Tuple(X,(tm_1,tm_2, . . .,tm_j, . . .,tm_n))

  = (X•tm_1, . . ., (X•α_i, X•α_j), . . .,X•tm_n)

  if tm_j = (α_i, α_j).

- Rearrange

  Signature:

  Relation, template → Relation

  Given a Relation R, a template tm, we have

  Rearrange(R, tm) = { X | ∃ y ∈ R : X = Rearrange_Tuple(y,tm) }.

  Example:

  Given a relation R = { (a, (c,e)), (b,(f,g)), (c,(h,j)) },

  a template tm = ((2.1,1),2.2),

  Rearrange(R,tm) = { ((c,a),e), ((b,f),g), ((h,c),j) }.

3.3.20 OperatorOnFunction(Operator, Function(Set))

Signature:

Operator, Function(Set) → Value

Given Operator Op, F a function, X a set,

OperatorOnFunction(Op,F(X))$_{X \subseteq DomainF}$ ≡

(1) if cardinality(X)=0 neutral element for Op;

(2) if cardinality(X)>=1 Op(F(x_1), OperatorOnFunction(Op,F(x)$_{(x \in X - \{x1\})}$))
where x_1 ∈ X.

note: Op and return Value should have the same type, which means, if the Op is for arithmetic operation, like +, -,*, and /, the return value should be a real number; if the Op is for a set operation, like Union, Intersection and difference, the return Value should be a set.

Example:

Given a relation R = { (a,1), (b,2), (c,3) } and a Op sum,

OperatorOnFunction(Sum,Range(R)) = 6.

3.3.21 Create(RelationName,list of arguments)

Create a new relation with a label which represents the name and each attribute of the relation. The label of the relation will be used for future reference.

This function will let user to give the information on a set of tuples. The first tuple must be the name of a relation, the rest of tuples are all four tuples specifying the information of each tuple of the relation. The format is:

(Name, (Tuple-index, AttributeName, Type, Size), (Tuple-index, AttributeName, Type, Size),...,(Tuple-index, AttributeName, Type, Size)).

For example:

(1) Create a labelled relation named Shop with (item,(price, store))

Create (Shop, (1, item, char, 10), (2.1, price, float, 8), (2.2, store, char, 10)).

(2) Create a labelled set named Item with one attribute item

Create (Item, (1(default), item, char, 10).

(3) Create a labelled table named Shops-info with attributes of (store, item, price, inventory-quantity)

Create(Shop-info, (1, store, char, 10), (2, item, char, 10), (3, price, float, 10), (4, inventory-quantity, char, 10)).

3.3.22 CreateAbsSRF(Set, Set, Predicate)

This function is to create a set, or a relation or a function using predicate.

Given a Set S_1, a Set S_2, a Predicate P on the elements of S_1 and S_2.

Signature:

Set, Set, Predicate $\rightarrow$ Set

For example:
INT is a set of integer.

CreateAbsSRF(INT, INT, x>y), we can get a relation

$R = \{ (x,y) \mid x \in INT \wedge y \in INT \wedge x > y \}$.

CreateAbsSRF(INT, INT, y=x+1), we can get a function

$R = \{ (x,y) \mid x \in INT \wedge y \in INT \wedge y = x+1 \}$.

CreateAbsSRF(INT,$\emptyset$,x>3), we can get a set

$R = \{ x \mid x \in INT \wedge x>3 \}$.

3.3.23 GetAttributeName(Relation,Tuple-index)

Get an attribute name (same as the attribute name for SQL tables) of a corresponding relation. This is used when a specific tuple is needed.

Given a Relation R and a Tuple-index on R, get the attribute name corresponding to the Tuple-index.

Signature:

Relation, Tuple-index $\rightarrow$ AttributeName

3.3.24 ArithmeticComp(Relation, Tuple-index, Arithmetic-Operator, Value)

This function is to apply arithmetic operator to a specific tuple.

Signature:

Relation, Tuple-index, ArithmeticOperator, Value → Relation

Given an Arithmetic Operator ao, Relation R, a Tuple-index ti on R, and a value v, then apply the arithmetic operator to the element designated by the ti of R using v.

For example:
Given a relation R_1,

R_1 = { (store,(item,price)) | store ∈ set_of_store ∧ (item,price) ∈ item × R^+ }.

We want to get a new relation contains all items whose name is apple with price tripled.

Assume we have:

R_1 = { (nofrills,(apple,1.25)), (fortino, (apple,1.45)) },

ArithmeticComp(R1, 2.2, *,3) = {(nofrills, (apple, 3.75)),(fortino,(apple,4.35)) }.

3.3.25 Assign(<−)

This is used for assignment application.

Given two Variables V_1 and V_2, and let 'V and V' denote the values of variable V before or after $V_2 \leftarrow V_1$ is executed then:

$V_2 \leftarrow V_1$: { V_2'= 'V_1 ∧ V_1' = 'V_1 }.

3.3.26 Reduction

This function is for applying functions to many variables.

Signature:

Function, Variable list → Value

For example:

Given three variables V_1, V_2, V_3, a funtion Union, we have,

Reduction(Union, V_1, V_2, V_3) = Union(Union(V_1, V_2), V_3).

3.4 Non-built-in Functions

This section illustrates the definition and use of non-built-in functions.

We need to use user-computed function applications because we want to get information about a relation or a function. These applications will only take one or two arguments. They are defined as following:

(1) Function application with One Argument

This function application is to get the value in the range associated with the value that is given in the domain. "F*" is added at the beginning of the function name in order to differentiate from the data value.

Signature:

Function, Value $\rightarrow$ Value

Example:

Given a Function $R_1 = \{ (x,y) \mid x \in N \wedge y \in N \wedge y = x+1 \}$,

$F^*R_1(1) = 2$.

The example are given in section 4.1.2.

(2) Predicate application with two arguments

This predicate application is to get the information of true or false if the given tuple is a member of the given relation or not. "P*" is added at the beginning of the relation name in order to differentiate from the data value. A pair is a two-tuple.

Signature:

Relation, pair $\rightarrow$ Value(**true** or **false**)

Example:

Give a Relation R_1 = { (1,3), (6,7),(4,8) },

P*R_1(1,3) = **true**.

P*R_1(27,92) = **false**.

3.5 An Example

In this section, a specific example is discussed, in order to give a feel for DNL. This example illustrates the uses of built-in and non-built-in functions, and the general format of "one-liner" query, sequential expressions and a combination of both.

Assume we have a relation

$$R_1 = \{(Store, (Item, (Price, Quantity)))|Store \in set\_of\_Store$$

$$\wedge(Item, (Price, Quantity)) \in Items \times (R^+ \times N)\}$$

3.5.1 Problem

A shopping list contains both Apples and Bananas, we want to find a single store that is going to give the lowest price for 2 pounds of apples and 3 pounds of bananas.

3.5.2 Solution

The solution has been given in Figure 2.1, Figure 2.2 and Figure 2.3 respectively in Chapter 2. The following sections explain the return value of each sub-expression and function call in each figure.

3.5.2.1 DNL program written as "one-liner" expression

The result of each sub-expression in Figure 2.1 is explained as following:

<1>[2] Intersection(Domain(Restriction(R1,

[2]a number enclosed by a pair of <> represents the sub-expression with that number

```
        GetAttributeName(R1,2.1) = 'apple'
          && GetAttributeName(R1,2.2.2) >2)),
    Domain(Restriction(R1,
        GetAttributeName(R1,2.1) = 'banana')
          && GetAttributeName(R1,2.2.2) > 3)))
```

This expression gets a set containing the stores that sell both apples and bananas and the quantity available is greater than 2 and 3 respectively.

<2> Restriction(R1, GetAttributeName(R1,1) member <1>)

This Restriction function call gets a relation with schema the same as R1, but restricted to tuples whose domain (store) is in the set described above.

<3>ArithmeticComp(Rearrange(Restriction(<2>, GetAttributeName((2),2.1) = 'apple'), (1,(2.1,2.2.1))), 2.2, *, 2)

This ArithmeticComp function call is to compute double the price of the relation(Rearranged from a relation(Restricted from <2> whose item is banana) with the schema as (store,(item,price))).

<4> ArithmeticComp(Rearrange(Restriction(<2>,GetAttributeName(<2>, 2.1) = 'banana'),(1,(2.1,2.2.1))),2.2,*,3)

This ArithmeticComp function call is to compute triple the price of the relation(Rearranged from a relation(Restricted from <2> whose item is banana) with the schema as (store,(item,price))).

<5> Union(<3>,<4>)

This Union function call computes a relation that is a union of <3> and <4>.

<6> Rearrange(<5>,((1,2.1),2.2))

This Rearrange function call reconstructs the positions of the tuples of <5> to a relation with schema as ((store,item),price).

<7> RangeMerge(<6>,1.1,Sum)

This RangeMerge function call sums the tuples in the range(doubled price of apple

and tripled price of banana) of by store.

<8> OperatorOnFunction(Minimum,Range<7>).

This OperatorOnFunction call gets a value that is the minimum value of the tuples in the range of <8>

3.5.2.2 DNL program written as a sequence of expressions

The result of each operation in Figure 2.2 is explained as following:

- R2: a relation with the schema as (store, (item,(price,quantity))) where each store in the domain sells Apples and the quantity of apples is greater than 2.
- R3: a relation with the schema as (store, (item,(price,quantity))) where each store in the domain sells Bananas and quantity of bananas is greater than 3.
- R4: a set of stores where each store sells both Apples and Bananas.
- R5: a relation with the schema as (store,(item,(price,quantity))) whose store satisfies with the condition that sells both apple and banana and the quantity of them is greater than 2 and 3 respectively.
- R6: a relation R6 with the schema as (store, (item, price)) and is rearranged from R5.
- R7: a relation with the schema as (store, (item,price)) is restricted from R6 where each store sells Apples.
- R8: a relation with the schema as (store, (item,price)) is restricted from R6 where each store sells Bananas.
- R9: a relation with the schema as (store, (item,price)) is computed from R7 where the item is apple and its corresponding price doubled.

- R10: a relation with the schema as (store, (item,price)) is computed from R8 where the item is banana and its price tripled.

- R11: a relation with the schema as (store,(item,price)) is a union from R9 and R10, a tuple of R11 could be ((apple,nofrills),2.7).

- R12: a relation with the schema as ((store,item),price) is Rearranged from R11. A tuple of this relation could be ((nofrills,apple),2.7)).

- R13: a relation with the schema as (store, sum of price) such that the range is the price for the 2 pounds of apples plus 3 pounds of bananas in each store.

- Final: a minimum price among stores.

3.5.2.3 DNL program written as a combination of "one-liner" and sequential expressions

This DNL program in Figure 2.3 is written as a mixture of using "one-liner" expressions to compose several functions together and sequential expressions for those functions that can easily to be confused while writing "one-liner" expression. We don't explain each step here since it is easy to understand after Figure 2.1 and Figure 2.2 have been understood.

Chapter 4

Design of DNL

This chapter provides an example of DNL and illustrates the syntax notation, the lexical structure, and the grammar of DNL.

4.1 Introduction

In this section, a brief summary of DNL design principles and a small example are provided.

4.1.1 Summary of DNL

DNL is intended to be a query language used to retrieve data from widely distributed database systems. In order to get a simpler and more practical language for users, DNL is based on mathematical expressions using nested function evaluation. e.g. $f_1(f_2(f_3(x)))$. Loops and conditional statements are not permitted. The followings are the detailed properties of DNL:

- There are two modes of execution.

  (1) Program execution mode: Program is executed after the complete program file is created.

(2) Command execution mode: Commands are executed one at a time, the user can see the result after each command execution.

- Lexical-graphics

  (1) Spaces are permitted between items, except that a function-name must be followed immediately by a left parenthesis. Spaces are not permitted within items such as names.

  (2) The language is case sensitive.

- Variable assignment is allowed.

- Programs can evaluate functions on relations and sets (see built-in function and non-built-in function definitions in Chapter 3), computing new functions and relations which can then be applied exactly as built-in functions or relations.

4.1.2 Features of DNL

- The DNL language has a flexible way of writing such as "one-liner" or sequential expressions or combination of two. Any of them is accepted as DNL program and will yield the same query result. The users can choose which way they may use on the basis of their own taste.

- From the program body, we can see that each DNL expression, from the whole point of view, is composed of either an assignment expression or a function call.

- In each function, each argument can be a return value of another function call. A relational expression like (a > b) or a template like (1.1, 2.2), which will be discussed later, can also be treated as an argument according to the requirement of function.

- The number of nested functions is not limited as long as the memory is sufficient.

- Applying a computed function and a predicate to a relation will get a result that can be used as a built-in function. This makes DNL more powerful, easier to use and get a proper result faster.

For example:

When we get R13 (in Figure 2.2), we can apply function or relation calls to it so that various results will get for different queries.

(1) F*R13(fortino), we can get the total price given by fortino.

(2) P*R13(fortino, OperatorOnFunction(Minimum,Range(R13))), we can see if fortino is one of the stores that gives the lowest price among stores if the return value is **true**.

(3) Domain(Restriction(R13,GetAttributeName(R13,2)= OperatorOnFunction(Minimum,Range(R13))), we can get a set of stores that all will give a lowest price.

(4) PreImage(R13, GetAttributeName(R13,2)=OperatorOnFunction(Minimum, Range(R13))), we can also get a set of stores that all give a lowest price.

4.2 Syntax Notation

The followings are the syntactic notations [8] used in this chapter:

- Syntactic variables are written as one or more characters with the leading letter of each word capitalised and are compounded by a pair of < >.

  For example: <Expression>.

- " ::= " is read "may be composed of".

- Alternative categories are separated by a bar "|".

- keywords in boldface are required by the syntax.

  For example: **char**, **int** etc. are the reserved words that can not be used as the name of other variables.

4.3 The Lexical Structure

The definitions of DNL lexical structure are adopted from [12]. DNL program is a text file, consisting of a sequence of characters. The lexical elements of DNL are the identifiers, keywords, constants, special symbols, and other separators such as white spaces characters.

(1) An identifier is a sequence of letters and digits. The first character must be a letter; the underscore "_" considered as a letter. Identifiers are case sensitive. They may be arbitrarily long, constrained only by the limits imposed by the internal stack of different computer systems. The definition of identifier is shown by:

```
<Identifier> ::= <Alpha>
                | <Identifier> <AlphaNum>
<AlphaNum> ::= <Alpha> | <Digit>
<Alpha> ::= A | ... | Z | a | ... | z | _
<Digit> ::= 0 | ... |9
```

(2) The keywords and identifiers are case sensitive. Note that identifiers may have keywords embedded in them, thus "intchar" is a valid identifier and will not be confused with the two embedded keywords, "int" and "char". The meaning of these words will be given in the appropriate sections. They are shown in Figure 4.1.

(3) The constants are listed in section 4.4. Each has a data type defined in the following section.

(4) The special operators are listed in Figure 4.4. All of these symbols are separators; they separate identifiers, constants, and built-in function applications.

(5) The aggregation operators that are used as arguments of built-in function applications are shown in Figure 4.2.

int	char	float	bool
Maximum	Minimum	Sum	Pi
diff	member	n_mem	subset
eq_subset	union	intersect	Image
Create	Insert	Delete	Domain
Range	RangeDivide	Rearrange	Join
RangeMerge	Product	OperatorOnFunction	Union
Difference	Identity	Cardinality	Restriction
Composition	Intersection	ArithmeticComp	Index
Reduction	GetAttributeName	CreateAbsSRF	PreImage

Figure 4.1: Reserved Keywords

Maximum		Minimum	Sum	Pi
+	-	*	/	
union		intersect	diff	

Figure 4.2: Special Operators

(6) The delimiter symbols that are used to group expressions, arguments, etc. are listed in Figure 4.3.

(7) The white space characters are space, tab, newline and return, they are used to separate other lexical elements.

4.4 Grammar

The DNL grammar employs ideas and machinery from two books [9, 10].

4.4.1 Terminal Symbols

- Letters and Digits

  See section 4.3 (1)

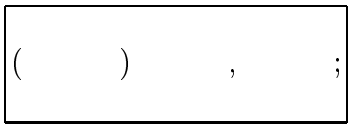

Figure 4.3: Delimiter Symbols

- Relational Operators

  >= (greater than or equal); <= (less than or equal);

  = (equal); != (not equal);

  < (less than); > (greater than);

  || (Or); && (And);

  member ($\in$); **_mem** ($\notin$);

  subset ($\subset$) **eq_subset** ($\subseteq$);

- Type Specifier

  int (integer);

  float (float);

  char (character);

  bool (boolean);

- Delimiter

  ; (semicolon);

  , (comma);

  . (period);

  ( ) (left and right brackets);

- Special symbols used in standard function

  Maximum (get maximum);

  Minimum (get minimum);

  Sigma ($\sum$);

  Pi ($\prod$);

  + (Plus);

  - (Minus);

  ***** (Multiply);

  / (Divide);

  union (union);

  intersect (intersection);

  diff (difference);

4.4.2 Constants

- <Decimal> ::= <Digit> | <Decimal> <Digit>
- <FloatLit>::= <Decimal> . <Decimal> | . <Decimal>
- <CharLit> ::= <AlphaNum> | ' <CharLit> <AlphaNum> '
- <BoolLit> ::= **true** | **false**

4.4.3 Program Syntax

This DNL grammar is acceptable to the YACC [3] parser-generator.

- <Program> ::= <ExpressionList>

- <ExpressionList> ::= <Expression> ;
  | <ExpressionList> <Expression> ;
- <Expression> ::= <LogicalOrExpr>
  | <Expression> <− <LogicalOrExpr>
- <LogicalOrExpr> ::= <LogicalAndExpr>
  | <LogicalOrExpr> || <LogicalAndExpr>
- <LogicalAndExpr> ::= <RelationalExpr>
  | <LogicalAndExpr> && <RelationalExpr>
- <RelationalExpr> ::= <SecondaryExpr>
  | <RelationalExpr> = <SecondaryExpr>
  | <RelationalExpr> != <SecondaryExpr>
  | <RelationalExpr> <= <SecondaryExpr>
  | <RelationalExpr> >= <SecondaryExpr>
  | <RelationalExpr> < <SecondaryExpr>
  | <RelationalExpr> > <SecondaryExpr>
  | <RelationalExpr> **member** <SecondaryExpr>
  | <RelationalExpr> **n_mem** <SecondaryExpr>
  | <RelationalExpr> **subset** <SecondaryExpr>
  | <RelationalExpr> **eq_subset** <SecondaryExpr>
- <SecondaryExpr> ::= <PrimaryExpr>
  | <FuncCall>
  | (<ArgumentExprList>)
- <PrimaryExpr> ::= <Constant> | <Identifier>
- <Constant> ::= <Decimal> | <FloatLit> | <CharLit> | <BoolLit>

- <FuncCall> ::= **Create** (<Identifier>, <a-set>)
  | **Insert** (<argument-two-list>)
  | **Delete** (<argument-two-list>)
  | **Range** (<argument-one-list>)
  | **Rearrange** (<argument-two-list>)
  | **Domain** (<argument-one-list>)
  | **Image** (<argument-two-list>)
  | **PreImage** (<argument-two-list>)
  | **RangeMerge** (<argument-three-list>)
  | **GetAttributeName** (<argument-two-list>)
  | **Restriction** (<argument-two-list>)
  | **OperatorOnFunction** (<argument-two-list>)
  | **Product** (<argument-two-list>)
  | **Union** (<argument-two-list>)
  | **Difference** (<argument-two-list>)
  | **Intersection** (<argument-two-list>)
  | **Identity** (<argument-one-list>)
  | **Cardinality** (<argument-one-list>)
  | **Join** (<argument-two-list>)
  | **Composition** (<argument-two-list>)
  | **RangeDivide** (<argument-one-list>)
  | **Index** (<argument-three-list>)
  | **Reduction** (<ArgumentExprList>)
  | **CreateAbsSRF** (<argument-three-list>
  | **ArithmeticComp** (<argument-four-list>)
  | **F***<Identifier> (<argument-one-list>)
  | **P***<Identifier> (<argument-two-list>)

- <base-set> ::= (<Tuple-index>, <Identifier>, <TypeSpecifier>, <Decimal>)
- <a-set> ::= <base-set> | <a-set> , <base-set>
- <argument-one-list> ::= <Expression>
- <argument-two-list> ::= <Expression>, <Expression>
- <argument-three-list> ::= <Expression>, <Expression>, <Expression>
- <argument-four-list> ::= <Expression>, <Expression>,

  <Expression>, <Expression>
- <ArgumentExprList> ::= <ArgumentExpr> | <ArgumentExprList> , <ArgumentExpr>
- <ArgumentExpr> ::= <Expression>

  | <TypeSpecifier>

  | <Tuple-index>

  | <Op>
- <Op> ::= **Maximum** | **Minimum** | **Sum** | **Pi** | **+** | **-** | ***** | **/** | **union** | **intersect** | **diff**
- <TypeSpecifier>::= **int** | **float** | **char** | **bool**
- <Tuple-index>::= <Digit> | <Tuple-index> . <Digit>

4.5 Highlights of DNL grammar syntax

This section will give highlights on DNL grammar syntax, the ungrammatical syntax that will be easily understood by reading the grammar will not be discussed in this section.

4.5.1 Constants

In DNL, we allow four kinds of constants, they are integer constants, float constants, character constants and boolean constants.

- Decimal: (integer constant) A Decimal consists of a sequence of digits, i.e. 12.
- FloatLit: (float constant) A FloatLit consists of an integer part, a decimal point and a fraction part, i.e. 5.6.
- CharLit: (character constant) A CharLit consists of a sequence of one or more characters compounded by a single quotes, i.e. 'abc'.
- BoolLit: (boolean constant) A BoolLit has a value of true or false, i.e. **true** (DNL value of true).

4.5.2 Expressions

The DNL language offers the usual collection of expression constructs, including assignment expressions and function applications.

DNL has a number of predefined operators. They are listed in Figure 4.4, along with their relative precedence from lowest to highest [12]. They will be used in function applications.

Operators	Associativity	Precedence
<-	none	none
&&, ||	Left	low
=, != <=,>=,<,>,member,n_mem, subset,eq_subset	Left	high

Figure 4.4: Precedence Table of Operators

4.5.3 Secondary Expressions

Secondary expressions have three forms:

- Primary expressions

- Two kinds of function applications:

  1. Built-in function calls, see Figure 2.2 first line Restriction (. . .).

  2. Non-built-in function calls. In DNL user can define new function and relations and use these just as they were built-in functions. If a relation R is functional, then F*R(a), where a is in the Domain of R, is a value in the Range of R and (a, F*R(a)) $\in$ R. If R is any binary relation, then R*R(a,b) is a program that will return true iff (a,b) $\in$ R.

  A function call is a function name followed by a pair of parenthesis enclosing an argument list. To make it easy to check the syntax of function calls, a detailed definition of each function has been given. The name of each built-in function and the number of arguments of each function have been specified except for functions where the number of arguments can vary while writing the DNL program.

  (1) <a-set> and <base-set> are used to set up the arguments of the Create function which needs an identifier as the first argument and a list of <base-set> followed.

  (2) <argument-one-list>, <argument-two-list>, <argument-three-list> and <argument-four-list>are used to restrict the number of arguments to one, two, three or four arguments as allowed in the function.

- ArgumentExpressionList. It is enclosed by a pair of brackets and can be a list of argument expressions.

4.5.4 Type Specifiers

The built-in type[1] specifiers are **int**, **char**, **bool** and **float** which specify type integer, character, boolean, and float number.

[1]The user can construct a type such as (int, float) for their own needs. This is the future work and is not allowed in current phase.

4.5.5 Operators

<Op> indicates the operators that can be used specifically in computation on a set. It is an argument of the built-in function call OperatorOnFunction, and RangeMerge.

4.5.6 Tuple-index

Tuple-index is used to select elements of a complex pair. For example, in Figure 2.2 second to last line, in the argument list of the function RangeMerge, 1.1 is a tuple index and used as an argument.

4.5.7 Relational Expressions

In DNL, a relational expression yields **true** or **false** if the specified expression is true or false.

This type of expression is used as an argument of built-in function. For example, in Figure 2.2 first line, " GetAttributeName(R1,2.1) = 'apple' " is considered as a relational expression (return value is **true** or **false**) which is an argument of function Restriction.

4.5.8 Logical AND Expressions

In DNL, the logical AND operator "&&" is used to evaluate two expressions. It guarantees left-to-right evaluation: the first operand is evaluated, including all side effects; if it is false, the value of the expression is **false**. Otherwise, the right operand is evaluated, and if it is false, the expression's value is **false**, otherwise **true**.

This type of expression is also used as an argument of built-in function. For example, in Figure 2.2 fourth line, " GetAttributeName(R1,1) member R4 && (GetAttributeName(R1,2.1) ='apple' " is considered as a logical and expression (return value is **true** or **false**) which is an argument of function Restriction.

4.5.9 Logical OR Expressions

In DNL, the logical OR operator "||" is used to evaluate two expressions. It guarantees left-to-right evaluation: the first operand is evaluated, including all side effects; if it is true, the value of the expression is **true**. Otherwise, the right operand is evaluated, and if it is true, the expression's value is **true**, otherwise is **false**.

Chapter 5

Design of DNL Parser/Syntax Checker

This chapter discusses the design of DNL parser/syntax checker. For executing DNL, several steps are needed in order to produce a correct and runnable program. Those steps are taken from [3]:

- Lexical analysis.
- Parsing.
- Semantic analysis.
- Intermediate code generation.
- Optimisation.
- Code generation.

This thesis only deals with the lexical analysis and syntax check (parser) parts, which are first two steps before executing code. There are two tools, called LEX [13] and YACC [13], that we use to generate a lexical analyser and parser for DNL according to the grammar of DNL.

5.1 Lexical Analysis and Parsing

5.1.1 Lexical Analyser

The lexical analyser (or scanner) will translate the input of the program file as a sequence of characters into an internal format (a sequence of tokens) which is readable by a parser for further analysis, e.g. to check whether the input stream complies with the grammar rules or not. The lexical analyser performs three main tasks [3]:

- Identifies the basic lexical units of the program file, which are called tokens.
- Removes extraneous characters such as newlines, carriage returns, and blanks.
- Detects errors in the input stream. Errors prevent the identification of tokens.

For DNL, we have decided to use LEX [13] to generate our lexical analyser.

5.1.2 Parsing

For any programming language, we should be given a set of grammar rules characterising the correct form of programs in that language. The parser or syntactic analyser accepts the output of the lexical analyser (scanner), which is a sequence of tokens, and verifies that the source program satisfies the grammatical rules of the language. Using a grammar, a program can be mechanically classified as belonging to the programming language or not. However, the grammar rules do not completely characterise the language. There are additional semantic restrictions. A string may fit the grammar, and still violate the semantic rules [3].

For DNL, we have decided to use YACC to generate our parser.

5.2 Design of the DNL Lexical Analyser Module

LEX [13] is a lexical analyser generator that is available as a standard tool in virtually all versions of Unix. Originally developed at Bell Labs, LEX supports powerful lexical pattern matching producing fairly efficient analysers [3]. The scanner will take a group

of strings of characters denoting identifiers, constants, or language words, etc. into tokens, which will be used later for parser or other applications.

- Secret of Lexical Analyser Module

  The secret of this module is the definitions of the tokens, includes the rules that define what is a token.

- Input of Module

  The input of this module is the text file (source code) of DNL.

- Output of Module

  The output of this module is a sequence of tokens that have been recognised by the scanner.

5.2.1 Definition of Input Strings

The following tokens which are used in DNL can be recognised by the scanner.

- The operators are listed in Figure 4.2 and Figure 4.4.
- The reserved words are shown in Figure 4.1.
- The type specifiers such as “int”, “float”, “char”, and “bool”.
- The constants such as integers, float numbers, boolean values and strings (a sequence of characters enclosed by a pair of single quotes) .
- The identifiers are a sequence of characters or digits starting with a letter.
- The delimiters such as “(”, “)”, “;” , and “,”.

5.2.2 Definition of Output Tokens

For all the symbols that have been recognised, each corresponding token name will be given.

For example:

when "<-" is recognised, a token name ASSIGN is given to represent this symbol in subsequent processing.

5.3 Design of DNL Syntax Checker/Parser Module

YACC [13] is a tool used together with LEX [13] to generate an LR(1) [3] parsing program from input consisting of a context-free grammar specification. Usually the generated program will be the parser for the language. The parser will take a sequence of tokens that are the output of the lexical analyser, check if the sequence complies with the language grammar rule in a syntactic way. If a parser could be generated, but with shift/reduce and reduce/reduce conflicts [13], then the grammar rules need to be refined because there are many cases that parser can not check due to those conflicts.

- The Parser Module

  The parser module consists two sub-modules:

  (1) Grammar rule module

  Each language has its own grammar rules that need to be followed by program designer. So the secret of this module is the grammar rules.

  (2) Name routine module

  The secret of the module is the data structure that used to store components of a piece of grammar rule. Here in this thesis, a set of trees is set up for storing each component that has been recognised.

- Input of Parser Module

The input of this module is a sequence of tokens recognised by the lexical analyser.

- Output of Module

  The output of this module is:

  (1) A set of error messages indicating that there is something wrong with the syntax of the program, or a single message saying that this program is syntactically right.

  (2) A sequence of messages saying that a grammatical structure and each grammatical component has been recognised.

5.3.1 Description of Input Tokens

The input tokens are the same as the output tokens of scanner.

5.3.2 Description of Output

5.3.2.1 Syntax Errors

The following are the main syntax errors:

- Failure to use semicolon to separate expressions.
- An illegal symbol, i.e. one that does not conform to the naming convention or is not defined in the syntax part.
- Bracket mismatch.
- Incorrect number of arguments of function call.
- Any other syntax errors such as wrong function name call and etc..

5.3.2.2 Output of the Parser

- The grammar rule structure will be printed out first, such as, "ASG_EXPR", which means an assignment expression is going to be evaluated.
- Each of the components composing the expression structure will be printed in the order of recognition, and the meaning of the component, such as ARG means an argument, will also be displayed before the actual token.
- An indented method of display is used in order to have a nice appearance and a better view of structure of each expression structure and each component that has been recognised.

Refer to the examples in following chapter for more details.

Chapter 6

A Simple Example of DNL and Parser/Syntax Checker Use

This chapter is going to provide a simple example of DNL and the parser/syntax checker used to illustrate DNL.

6.1 Simple Example

This example gives a full procedure from setting up relations for retrieving and computing relations.

- Problem

  Consider three relations:

  $R1 = \{(town, hotel) | town \in set\_of\_town \wedge hotel \in set\_of\_hotel\}$

  $R2 = \{(hotel, dog\_allowed) | hotel \in set\_of\_hotel \wedge dog\_allowed \in \{true, false\}\}$

  $R3 = \{((day, (hotel, town)), num\_of\_free\_rooms) | ((day, (hotel, town)) \in set\_of\_day \times set\_of\_hotel \times set\_of\_town \wedge num\_of\_dogee\_room \in INT^{+}\}$

  Compute a new relation

  $R = \{((town, day), num\_of\_dogee\_room) | (town, day) \in set\_of\_town \times set\_of\_day \wedge num\_of\_dogee\_room \in INT^{+}\}$

```
Create( R1, (1, Town ,char, 15), (2, Hotel, char, 20));

Create(R2, (1, Hotel, char, 20), (2, Dog-allowed, int, 1);

Create(R3, (1.1, Date, char, 10), (1.2.1, Hotel, char, 20),
       (1.2.2, Town, char, 15), (2, Num-Room, int, 4));
```

Figure 6.1: Creating Relations

- Solution

  A DNL program could be the following:

  (1) Creating three relations R1(Town, Hotel); R2(Hotel, Dog-allowed); R3((Date, (Hotel, Town)), Num-Room) at Figure 6.1.

  (2) Defining elements of each relation at Figure 6.2.

  (3) Calling built-in functions to perform functions computation to get the desired relation, see Figure 6.3.

  The following steps are performed:

  a. Restrict R2 with a condition of allowing dogs or not, and get the Domain (a set of hotels which allow dogs) of the restricted relation.

  b. Rearrange R3 into a new relation with schema ((Hotel, Town), (Date, Num-Room))).

  c. Restrict above relation with the condition of hotels belonged to the set of allowing dogs' hotel).

  d. Rearrange above relation into a new relation with schema ((Town, Date), (Hotel, Num-Room)).

  e. Sum up the Num-Room according to (Town,Date) to get a new relation with schema((Town, Date), Num-Room).

```
Insert(R1, ('Hamilton', 'Holiday-Inn'));
Insert(R1, ('Hamilton', 'Village-Inn'));
Insert(R1, ('Hamilton', 'Sheraton'));
Insert(R1, ('Toronto', 'Hilton'));
Insert(R1, ('Toronto', 'Sheraton'));
Insert(R1, ('Toronto', 'Days-Inn'));
Insert(R1, ('Burlington', 'Village-Inn'));
Insert(R1, ('Burlington', 'Holiday-Inn'));
Insert(R2, ('Holiday-Inn', 'true'));
Insert(R2, ('Village-Inn', 'true'));
Insert(R2, ('Sheraton','false'));
Insert(R2, ('Hilton', 'false'));
Insert(R2, ('Days-Inn', 'true'));
Insert(R3, (('03/08/2000', ('Village-Inn', 'Hamilton')), 10));
Insert(R3, (('03/08/2000', ('Holiday-Inn', 'Hamilton')), 5));
Insert(R3, (('03/08/2000', ('Sheraton', 'Hamilton')), 7));
Insert(R3, (('03/08/2000', ('Hilton', 'Toronto')), 20);
Insert(R3, (('03/08/2000', ('Sheraton', 'Toronto')), 15);
Insert(R3, (('03/08/2000', ('Days-Inn', 'Toronto')), 8));
Insert(R3, (('03/08/2000', ('Village-Inn', 'Burlington')), 3));
Insert(R3, (('03/08/2000', ('Holiday-Inn', 'Burlington')), 4));
```

Figure 6.2: Inserting Elements Into Relations

```
a. Stemp <- Domain(Restriction(R2, GetAttibuteName(R2, 2) = 'true'));

b. Rtemp <- Rearrange(R3,((1.2.1, 1.2.2), (1.1,2)));

c. Rtemp <- Restriction(Rtemp , GetAttributeName(Rtemp , 1.1) member Stemp);

d. Rtemp <- Rearrange( Rtemp , (((1.2, 2.1),1.1) ,2.2));

e .RangeMerge(Rtemp, 1.1, Sum);
```

Figure 6.3: Retrieve and Compute Relations in Sequential expressions

Here only the sequential expressions of DNL is provided for better understanding, the other two ways like one-line expression and combination of one-line expression with sequential expressions can also be easily written and are provided in Appendix A.3. All three DNL files are recognised by the DNL parser.

6.2 Parser/Syntax Checker Use

This section will give samples of syntax error messages and a display message of recognised components of a DNL expression.

6.2.1 Sample Syntax Error Messages

- "Line * column * syntax error before or at ' ' ".
- "Line * column * syntax error before or at ' ', ';' is expected ".
- "naming Identifier violation".
- "Line* column * syntax error before ' ', brackets mismatch".
- "Line* column * syntax error before ' ', wrong number of argument, 1(2,3,4) arguments are expected".

6.2.2 Sample Display Message of Recognised Components

A simple expression is given to show the way of what display module does.

- DNL expression:

  Stemp <- Domain(Restriction(R2,GetAttributeName(R2,2) = 'true'));

- Output of the display module is the following:

```
ASG_EXPR
    IDENTIFIER Stemp
    ASSIGN
    FUNC_CALL
      FUNC_DOMAIN
      ARG
        FUNC_CALL
          FUNC_RESTRICTION
          ARG_LIST
            ARG
              IDENTIFIER R2
            ARG
            EQ_EXPR
              FUNC_CALL
                FUNC_GET_ATTRIBUTE_NAME
                ARG_LIST
                  ARG
                    IDENTIFIER R2
                  ARG
                    2
              EQ
              BOOL VALUE : true
```

Chapter 7

User's Guide

This chapter is a user's guide that tells a user how to use the DNL parser and how to change the sub-module of name-routine in order to do further work, such as type-checking and code generation.

7.1 Files in the DNL Parser

- Makefile - a file containing a sequence of Unix script commands for compiling all related files.
- lexer.l - a LEX source code file, is used to generate DNL lexical analyser.
- parser.y - a YACC source code file, is used to generate DNL parser.
- main.c - a c program file, is used to read in DNL program.
- parser - a DNL parser file, which is an executable file. If someone doesn't like this name, go to Makefile and change it.
- catTree.h, catTree.c - two c program files, used to print out the components that the parser has recognised. The programs might or might not be useful for further application.
- test - a folder that includes a list of DNL test files.

7.2 Using or Modifying the DNL Parser

7.2.1 Using the DNL Parser

Like executing any other executable file, type in:

> parser DNL-file-name return-key

Note: parser is the name for an executable file (like a.out for C/C++ language), DNL-file-name is the name of DNL that needs to be checked.

7.2.2 Modifying DNL Parser Files

The following files are likely to be modified for specific applications or uses.

(1) Makefile - Changes can be made if more files need to be added.

(2) lexer.l - Changes can be made if more characters of DNL source code need to be translated into tokens or the sequence of characters of DNL source code need to be modified. Makefile must be re-run to get a new version of lex.yy.c and lex.yy.o files before executing DNL parser.

(3) parser.y - Changes can be made if DNL syntax needs to be modified or more routines, such as type-checking or code generating, need to be added or replaced after each grammar rule. Makefile must be re-run to get y.tab.h and y.tab.c files before executing DNL parser.

(4) main.c - Changes can be made if the format of the reading is required to modify. Makefile must be re-run before executing DNL parser.

(5) catTree.h, catTree.c - an application program of specific use can replace these two files. Corresponding Makefile needs to be modified too.

Chapter 8

Results and Conclusions

This chapter summarises the design and the implementation of the DNL parser, discusses the limitations of DNL, describes a set of testing files (Appendix), suggests future work and draws some conclusions.

8.1 Results

The DNL parser is the base for other parts of the compiler that we are developing, such as type checking and code generating; it checks the syntax error of source code and generating a sequence of tokens that can be used later. All the other components will be developed using the parser.

The DNL lexical analyser and parser provides two functions that are syntax checking and displaying the recognized components. The output of this parser has been implemented successfully and return the results that are expected.

8.2 Limitations

Many English words, such as "Create", "Restriction" and etc., have been used as reserved words for special usage. They can not be used in any other way. The DNL parser can only take in a text file as an input. DNL can't do computations on matrix.

8.3 Testing

A set of testing files has been generated for testing the DNL parser since it is the core part of executing DNL program.

Testing files are listed in Appendix, they can be not only used to test the existing parser but also to later test if there are modifications to the DNL syntax.

8.4 Future Work

As more complicated examples tested and more real needs are proposed, we might find that some modifications need to be made in order to refine or extend the DNL that we have developed. The followings are suggested for future work related to the DNL design.

- Data type definitions. Allow the users to define their own data type such as relation, set and etc.
- More built-in function definitions will be needed if DNL allows more data type stored in its databases.
- User defined functions. Users might want to define functions that perform specific needs according to their tastes.
- More features of DNL syntax might be added to make the language more convenient, for example allow arithmetic computations or conditional expressions.

8.5 Conclusions

The DNL language has been designed. The grammar of DNL has been developed and the DNL parser has been implemented successfully.

Many mathematical theories (such as relation algebra) and modern technologies (such as relational databases, SQL, APL and World Wide Web) have been incorporated in the design of DNL. The DNL parser allows the user to check the syntax errors of the DNL program, it also creates a tree for semantic analysis.

This thesis is the first step towards building a Data Network; it is an initial proposal for a new developed language. DNL has not been completely implemented in this phase, some new structures or ideas might be added in the future.

We have shown how relation algebra can serve as a basis for a powerful query language that can be used to integrate data bases that are distributed across a data network.

APPENDIX

Appendix A

Test Files

All the test files listed in Appendix give solutions to specific problems. All the relations, sets are constructed before they are used.

A.1 Shopper's Guide

We still use the example given in chapter 3, but no longer require that apples and bananas are bought of the same store because one store might not give lowest price for both apple and banana.

$$R_1 = \{(Store, (Item, (Price, Quantity)))|Store \in set\_of\_Store$$

$$\wedge(Item, (Price, Quantity)) \in Item \times (R^+ \times N)\}$$

A.1.1 Problem

Find which stores (it's not necessary to be one store) will give lowest price for 2 pounds of apples and 3 pounds of banana.

A.1.2 Solution

A.1.2.1 "One-liner" Expression Solution

```
Union
    (Restriction
     (R_temp1 <- Rearrange
        (ArithmeticComp
         (Restriction(R_1,
          GetAttributeName(R_1, 2.1) = 'apple'
           && GetAttributeName(R_1, 2.2.2) > 2), 2.2.1, *, 2),
         ((1, 2.1), 2.2.1)),
     GetAttributeName(R_temp1, 2) =
                OperatorOnFunction(Minimum, Range(R_temp1))),
     Restriction(R_temp2 <- Rearrange
        (ArithmeticComp
         (Restriction(R_1,
          GetAttributeName(R_1, 2.1) = 'banana'
           && GetAttributeName(R_1, 2.2.2) > 3), 2.2.1, *, 3),
        ((1, 2.1), 2.2.1)),
     GetAttributeName(R_temp2, 2) =
                OperatorOnFunction(Minimum, Range(R_temp2))));
```

A.1.2.2 Sequential Expressions Solution

```
R_2 <- Restriction(R_1, GetAttributeName(R_1, 2.1) = 'apple'
        && GetAttributeName(R_1, 2.2.2) > 2);
R_3 <- Restriction(R_1, GetAttributeName(R_1, 2.1) = 'banana'
        && GetAttributeName(R_1, 2.2.2) > 3);
R_4 <- ArithmeticComp(R_2, 2.2.1,*, 2);
R_5 <- ArithmeticComp(R_3, 2.2.1, *, 3);
R_6 <- Rearrange(R_4, ((1, 2.1), 2.2.1));
R_7 <- Rearrange(R_5, ((1, 2.1), 2.2.1));
R_8 <- Restriction(R_6,
```

GetAttributeName(R_6, 2) =
OperatorOnFunction(Minimum, Range(R_6)));
R_9 <- Restriction(R_7,
GetAttributeName(R_7, 2) =
OperatorOnFunction(Minimum, Range(R_7)));
R_{10} <- Union(R_8, R_9);

A.1.2.3 Combination of "One-liner" and Sequential Expressions Solution

R_2 <- ArithmeticComp
(Restriction(R_1,
GetAttributeName(R_1, 2.1) = 'apple'
&& GetAttributeName(R_1, 2.2.2) > 2), 2.2.1, *,2);
R_3 <- ArithmeticComp
(Restriction(R_1,
GetAttributeName(R_1, 2.1) = 'banana'
&& GetAttributeName(R_1, 2.2.2) > 2), 2.2.1, *, 3);
R_4 <- Restriction
(R_{temp1} <- Rearrange(R_2, ((1, 2.1), 2.2.1)),
GetAttributeName(R_{temp1},2) =
OperatorOnFunction(Minimum, Range(R_{temp1})));
R_5 <- Restriction
(R_{temp2} <- Rearrange(R_3, ((1, 2.1), 2.2.1)),
GetAttributeName(R_{temp2}, 2) =
OperatorOnFunction(Minimum, Range(R_{temp2})));
R_6 <- Union(R_4, R_5);

A.2 Airplane Maintenance Information

Air-planes need to be maintained regularly. In order to get a quick search of which airplane needs to be maintained, a set of relations is constructed:

$$S_1 = \{PlaneId | PlaneId \in INT\}$$

$$R_1 = \{(Airplane, set\_of\_PlaneId) | Airplane \in set\_of\_Airplanes \wedge set\_of\_PlaneId \subseteq S_1\}$$

$$R_2 = \{(PlandId, MaintenanceInfo) | PlaneId \in S_1 \wedge MaintenanceInfo \in \{true, false\}\}$$

$$R_3 = \{(Airport, PlaneId) | Airport \in set\_of\_Airport \wedge PlaneId \in S_1\}$$

$$R_4 = \{(PlaneId, Number\_of\_Seats) | PlaneId \in S_1 \wedge Number\_of\_Seats \in INT^+\}$$

A.2.1 Problem

Find the number of seats available in airport = "Toronto" where the maintenance condition is "true".

A.2.2 Solution

A.2.2.1 "One-liner" Expression Solution

```
OperatorOnFunction(Sum,
      Range(
       Restriction(
        R4,
        GetAttributeName(R4,1) member
         Range(
          (Restriction(R2,
           GetAttributeName(R2,1) member (Intersection(Domain(R2),
           Range(Restriction(R3,
            GetAttributeName(R3,1) = 'Toronto')))
             && GetAttributeName(R2,2) = 'true'))
              )));
```

A.2.2.2 Sequential Expressions Solution

```
R5 <- Restriction(R3, GetAttributeName(R3,1) = 'Toronto');
    S1 <- Range(R5);
    R6 <- Restriction(R2,
            GetAttributeName(R2,1) member (Intersection(Domain(R2),S1))
            && GetAttributeName(R2,2) = 'true');
    S2 <- Domain(R6);
    R7 <- Restriction(R4,
            GetAttributeName(R4,1) member S2 );
    OperatorOnFunction(Sum, Range(R7));
```

A.2.2.3 Combination of "One-liner" and Sequential expressions Solution

```
S1 <- Range(Restriction(R3,
                GetAttributeName(R3,1) = 'Toronto'));
    S2 <- Range(Restriction(R2,
            GetAttributeName(R2,1) member (Intersection(Domain(R2),S1))
            && GetAttributeName(R2,2) = 'true'));
    OperatorOnFunction(Sum, Range(Restriction(R4,
                    GetAttributeName(R4,1) member S2 )));
```

A.3 Hotel Information

The example is illustrated in section 6.1.

A.3.1 "One-liner" Solution

```
RangeMerge(
    Rearrange(
        Restriction(
            Rearrange(R3, ((1.2.1, 1.2.2), (1.1,2))),
```

GetAttributeName(R_{temp} , 1.1) member
Domain(Restriction(R_2,GetAttributeName(R_2, 2) = true))),
(((1.2, 2.1),1.1),2.2)) , 1.1, Sum);

A.3.2 Combination of "One-liner" and Sequential expressions Solution

S_1 <- Domain(Restriction(R_2, GetAttributeName(R_2,2) = 'true'));
R_4 <- Restriction(R_{temp} <-Rearrange(R_3,((1.2.1,1.2.2),(1.1,2))),
GetAttributeName(R_{temp},1.1) member S_1);
RangeMerge(Rearrange(R_4,(((1.2,2.1),1.1),2.2)), (1.1), Sum);

Bibliography

Bibliography

[1] Software Engineering Research Group *"Tabular Tool System Developer's Guide"*. CRL Report No.339, Telecommunication Research Institute of Ontario(TRIO), Mcmaster University, Hamilton, Ont., January 1997.

[2] David L. Parnas. *"Draft of Data Network Design"*. McMaster Software Engineering Group, Hamilton, Ontario, February 1998.

[3] Arthur B. Pyster. *"Compiler Design and Construction"*. Van Nostrand Reinhold Company Inc., 1988.

[4] Ramez Elmasri, Shamkant B. Navahte. *"Fundamentals of Database Systems"*. Assison-Wesley. An imprint of Addison Wesley Longman, Inc. 2000.

[5] "http://www.acm.org/sigapl/whyapl.htm" A web-document created by Special Interest Group on APL programming language of ACM (Association for Computing Machinery).

[6] Harold S. Stone. *"Discrete Mathematical Structure and Their Applications."* Science Research Associates Inc. 1973.

[7] Gunther Schmidt, Thomas Strohlein. *"Relations and Graphs, Discrete Mathematics for Computer Scientists"*. Springer-Verlag, Berlin Heidelberg 1993.

[8] Jame E. Hendrix. *"A Small C Compiler, Language, Usage, Theory, and Design"*. M&T Pulishing Inc. 1988.

[9] Brian W. Kernighan, Dennis M. Ritchie. *"The C Programming Language"*. Bell Laboratories, Incorporated, 1988.

[10] Backus J. W., F. L., Green J. Katz C., McCarthy J. Naur, P. (ed.), Perlis A. J., Rutishauer H., Samelson K., Vauquois B., Wegstein J. H., van Wijingaarden A., and Woodger M.. *"Report on the Algorithmic Language ALGOL 60"*. Commun. ACM 3:299-314. 1960.

[11] Robin Hunter. *"Compilers Their Design and Construction - Using Pascal"*. John Wiley & Sons Ltd. Reprinted December 1986.

[12] S. Owre, N. Shankar and J. M. Rushby. *"The PVS Specification Language (Beta Release)"*. Computer Science Laboratory, SRI International, Menlo Park, CA 94025. April 12, 1993.

[13] Editioral/production supervision: Karen Skrable Frotgang. Manufacturing Buyer: Mary Ann Gloriande. *"UNIX System V Release 3.2 Programmer's GUIDE, Vol. 1"*. Published by Prentice-Hall Inc. A division of Simon-Schuster, Englewood Cliffs, New Jersey 07632, 1989.

[14] Watt, David A. *"Programming Language Syntax and Semantics"*. Prentice Hall International(UK) Ltd., 1991.

[15] C.J. Date. *"An Introduction to Database System. VOLUME I, Fifth Edition"*. Addison-Wesley Publishing Company, Inc., 1990.

[16] Barry E. Jacobs *"Applied Database Logic. Vol. I, Fundamental Database Issues"*. Prentice Hall, Inc., Englewood Cliffs, New Jersey 07632, 1985.

[17] Susan M. Visser *"Teach Yourself DB2 Universal Database in 21 Days"*. Sams Publishing, 1998.

[18] Feng Cui *"Proposal of Data Network Design"*. McMaster Software Engineering Group, Hamilton, Ont., 2000.

[19] Pui-li Li *"Proposal of Data Network Design"*. McMaster Software Engineering Group, Hamilton, Ont., 2000.

[20] David Lorge Parnas *"Electrical Engineering 768, Professional Software Development"*. Course-Ware. McMaster University, Hamilton, Ont., September 1998.

[21] "http://www.pinpub.com/sqlpro/tips.htm". A web site on Microsoft SQL Server Professional Tips. Pinnacle Publishing, 1998.