Interpreting Data Network Language on a Single Processor

By
Feng Cui, B.Eng.

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

Master of Science
McMaster University

© Copyright by Feng Cui, June 25, 2001

Abstract

There is such a large amount of varied information available on the Internet that it is hard for a human to query this information without efficient tools. Using structured relational data can greatly improve our ability to search for data and data compute new information across networks. The McMaster University Software Engineering Research Group (McSERG) proposes to use binary relations to organise data on networks and use binary relation operations to do information query. A new language, Data Network Language (DNL) [16], has been developed by McSERG to apply this idea. This thesis presents the design and development of the Data Network Language Interpreter System, which is the implementation of DNL. By means of examples, this thesis shows how the use of DNL can improve the consistency and completeness in information retrieve.

Acknowledgements

I would like to express my sincere thanks to Dr.David L. Parnas for his guidance, advice, and enthusiasm throughout this thesis. He brought the new idea to use binary relations on the Internet technology to us which has played a major role in this work. Without his constant support and encouragement, it would have been impossible for me to finish this work.

A great deal of help, and many valuable suggestions and comments for this work from Dr.Ridha Khedri are highly appreciated. I would also like to thank Dr.Stavros Kolliopoulos reading this thesis and making very helpful suggestions. Dr.Robert Baber, Dr.Emil Sekerinski, and Dr.Martin von Mohrenschildt are appreciated for their help on this work, especially on Yacc and compiler design. To my colleagues and friends in McMaster Software Engineering Research Group (McSERG), especially Pui-Li Li, Ou Wei, Hongyu Wu, and Xiaoning Yan, I say a heartfelt thank you.

Special thanks to my wife, Rong Wu and my family for their love, encouragement and support.

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

Contents

Abstract i

Acknowledgements ii

List of Figures vii

List of Tables viii

1 Introduction 1

1.1 Motivation 1

1.2 Limitations of Current Approaches 2

1.2.1 Why Not Use HTML? 2

1.2.2 Why Not Use SQL? 3

1.2.3 How to Use Binary Relations 4

1.3 The Structure of the DNL Effort 5

1.4 Thesis Scope 6

1.5 Outline of This Thesis 7

2 Review of Notation and Terminology 8

2.1 Sets and Tuples 8

2.2 Relations, Domain, and Range 9

2.3 Attributes, Tuple Index, and Templates 10

2.4 Functions and Predicates 11

2.5 Function Applications and Terms 11

2.6 Primitive Relation and Predicate Expression 11

2.7 Set and Relational Algebra 12

2.8 Inductively Defined Predicates 13
2.9 Data Tables 13
2.10 Tabular Expressions and the TTS 13

3 Software Tools and Approaches 16
3.1 Lex and Yacc 16
3.2 Application Programming Interface 17

4 Review of Data Network Language 19
4.1 Basic Operations on Sets and Binary Relations 19
4.1.1 Assignment 19
4.1.2 Create 20
4.1.3 Insert 20
4.1.4 Delete 21
4.1.5 Union 21
4.1.6 Intersection 22
4.1.7 Difference 22
4.1.8 Cardinality 22
4.1.9 Domain 23
4.1.10 Range 23
4.1.11 Identity 24
4.1.12 Composition 24
4.1.13 Join 24
4.1.14 Index 25
4.1.15 Restriction 25
4.1.16 Image 26
4.1.17 PreImage 26
4.2 Advanced Operations 27
4.2.1 OperationOnFunction 27
4.2.2 RangeDivide 27
4.2.3 RangeMerge 28
4.2.4 Rearrange 28
4.2.5 CreateAbsSRF 29
4.2.6 ArithmeticComp 30

4.2.7 Reduction 31
4.3 Data Access Operation 31
4.3.1 GetAttribute 31

5 Design of the Data Network Language Interpreter 32
5.1 System Overview 32
5.2 Requirements (Informal) 33
5.2.1 User Interface 34
5.2.2 DNL Interpreter Application Programming Interface 35
5.2.3 Anticipated Changes 36
5.3 Module Decomposition 36
5.4 Module Guide and Module Interface 37
5.4.1 Error Module (DN_Error.c) 37
5.4.2 Display Module (Display.c) 38
5.4.3 Information Module (DN_Info.c) 39
5.4.4 Record Module (DSFile.cc) 43
5.4.5 Set Module (Set.h) 44
5.4.6 Extensive Set Module (ExSet.h) 46
5.4.7 Intensive Set Module (DN_Idp.h) 47
5.4.8 Relation Module 48
5.4.9 Operator Module (Operator.c) 61
5.4.10 Place Holder (Holder.cc) 63
5.4.11 Parser Module(p.y and main.c) 65
5.5 Algorithms for Compiling DNL 65

6 Illustrative Examples 67
6.1 Town-Hotel Example 67
6.2 Shopper's Guide Example 71

7 Results and Conclusions 74
7.1 Results 74
7.2 Limitations 75
7.2.1 Performance Optimization 75
7.2.2 Data Type in Data Network Language 75

7.3 Future Work 76
7.4 Conclusions 76

A Data Network Language Makefile Reference **77**
A.1 Parser Makefile 77
A.2 Makefile for Linking with API 79

List of Figures

1.1 A 3-ary relation and 2 derived binary relations 4

2.1 A Relation 10
2.2 Traditional expression $f(x, y)$ 14

5.1 Requirement of Data Network Language Interpreter System 34
5.2 DNL Top Level Module and Uses Relation 37
5.3 Secret of Relation Module 49
5.4 Data Structure of an Extensive Relation Element 56

List of Tables

2.1 Tabular expression $f(x, y)$ 14

5.1 Access Program Interface for Error Module 38
5.2 Access Program Interface for Display Module 38
5.3 Access Program Interface for class Field 39
5.4 Access Program Interface for class Node 40
5.5 Access Program Interface for class List 41
5.6 Access Program Interface for class FStruct 42
5.7 Access Program Interface for class Record 44
5.8 Access Program Interface for class Set 45
5.9 Access Program Interface for class Extension Set 46
5.10 Access Program Interface for class IDP 47
5.11 Access Program Interface for class TupleDS 50
5.12 Access Program Interface for class RelaDS 51
5.13 Access Program Interface for class InRelation 53
5.14 Access Program Interface for ExRelation 57
5.15 Access Program Interface for class Relation 58
5.16 Access Program Interface for class DTable 59
5.17 Access Program Interface for class Operator 61
5.18 Access Program Interface for class Holder 64

A.1 File Dependencies and Compiler Generation Procedure 78
A.2 File Dependencies and Compiler Generation Procedure 79

Chapter 1

Introduction

This thesis proposes a new way of getting information to people from the Internet and presents a prototype of the new language interpreter, Data Network Language Interpreter. This work is closely related to X. Yan's thesis, "Data Network Language Design"[16] and P. Li's on-going thesis "Networking Design of Data Network Language" [10]. X. Yan defined the syntax and semantics of the Data Network Language. P. Li is doing research on the networking part for this language.

1.1 Motivation

Information gathering, processing, and distribution has become a key technology in our world. Data communication is one of the fundamentals of information technology. We have seen the construction of world-wide networks built using telephone networks, cable, computers, and communication satellites. Resource sharing, making data available to anyone on the network without regard to the physical location of the resource and the user, is an important topic for information technology.

Both world-wide networks and local networks gather data about such diverse subjects as on-line shopping, sports news, and education. It is hard for humans to query such large and diverse information without efficient tools.

Use of structured relational data can greatly improve the ability of data searching

and data computation on networks. People can use relational algebra to compute exactly the data they seek. We will discuss details in section 1.2) The Software Engineering Group (SERG) at McMaster University supervised by Dr.Parnas is working to design, implement, and document a new language: Data Network Language (DNL). Our research on DNL is going to provide an approach for applying binary relations to retrieve information in networks.

1.2 Limitations of Current Approaches

1.2.1 Why Not Use HTML?

Today when people retrieve information from the Internet, most use keyword search on Hypertext Markup Language (HTML) web pages. HTML is one of the essential languages for sharing information on the Internet. In HTML documents are linked from one piece to another. Users have two ways to find interesting documents, either by following links or by using keyword search engines, such as Yahoo, to match some documents that contain the keywords. However, HTML is a markup language, suited to allow human beings to view files. It has limited capability to do an intelligent search. For instance, if a Dog-lover's Club wants to have a meeting in New York City, without building custom-tailored programs, the user has to download and view a lot of unrelated information to search for "the cheapest hotel rooms in New York City that allow guests to bring dogs".

In most cases, keyword searching comes up with a lot of junk information. In the worse cases, users have to download all the information and view it before knowing it is useless. The result of keyword searching is imprecise and inconsistent. Unless used in a deliverily restricted way HTML is not useful for structured data queries.

Another approach is to build a hyperlink structure on the web and use tools to mine the hyperlink structure [4]. However, this method also uses keywords to interpret the content and needs a human to view the pages.

1.2.2 Why Not Use SQL?

Relational database technology and Structured Query Language (SQL) can answer the query we present above. However, they still have problems for precise searching on the Internet. The reason is as follows. The basic idea of a relational database system is using two-dimensional tables to manage all kinds of data and relations. In certain cases this may not always be true because of exceptions. When a lot of information is set up together in a database management system, the information is usually complicated and could be represented by multiple dimension relations. Some kinds of data which belong to a certain set may not fit the two-dimensional tables so well.

For instance, assuming there is a two dimensional relation table containing product information in a supermarket database management system ("product name", "product number", "price" ...). In the real world, however, there are always complications in the table because of exceptions. Some products are new with a price that has not been decided yet (unavailable exception). Some are so competitive that the price changes so rapidly (unfixed exception). Others may be free. There may be other special cases for the price. What should be filled in for the product "price" for the exceptional cases? One may say "0" for a "free" case, "-1" for "null" or "unavailable" cases, and so on. This works well if no other program uses the data. It may cause problems, however, when a computer passes this kind of information to another computer. Some software may interpret this way: "0" for "free to have" and "-1" for giving you one dollar when you have it. A note can explain what "0" and "-1" means in this case. But it is hard for others (people or programs) to find and understand the note. This kind of problem makes things worse when many programs interpret the data produced by other programs. From this example we can see it is not easy to apply two dimensional tables to store data that needs to be accessed by programs across networks.

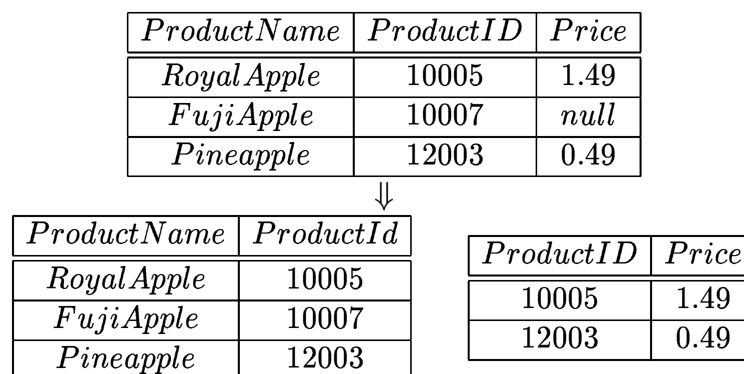
Another reason for not using relational database systems across networks is that most database systems associate with restricted data entry programs. Without knowing the data entry programs, it is hard to do information comparison on

different database systems.

Finally, two dimensional tables are N-ary relations. N-ary relations sometimes make it very difficult to compute across networks. The algebra of n-ary relations is not as simple and intuitive as Thski's rules for binary relations [15].

1.2.3 How to Use Binary Relations

Any information stored as an N-ary relation can be viewed as a set of binary relations. For instance, from the relation (product name, product ID, price) we can determine a few binary relations, such as (product name, product ID) and (product ID, price). The binary relation is the basic (simplest) form of relations. If we decompose an N-nary relation to many binary relations, in each binary relation all "null", "unavailable", or "don't know" cases are eliminated in the domain of binary relations. We just keep all the "do know" cases in our binary relation sets and solve the problem easily. All data in the binary relation sets will be precise and consistent. It is relatively easy, in addition, to apply binary relations representing functions. We present this in the following chapters. Generally speaking, using only binary relations is a good idea because of their simple structure (syntax) and precise semantics.



ProductName	*ProductID*	*Price*
RoyalApple	10005	1.49
FujiApple	10007	*null*
Pineapple	12003	0.49

⇓

ProductName	*ProductId*
RoyalApple	10005
FujiApple	10007
Pineapple	12003

ProductID	*Price*
10005	1.49
12003	0.49

Figure 1.1: A 3-ary relation and 2 derived binary relations

To avoid arbitrariness (complication by exception) in transferring information and

to support a database system using networks technology, Parnas proposed using only binary relations instead of general tables in querying and retrieving data[1]. The McMaster University Software Engineering Research Group is creating a new language to apply binary relations and network technology. This language is called "Data Network Language" (DNL). The fundamental idea is that the concept of binary relations can be used to improve the consistency and accuracy of information retrieving.

1.3 The Structure of the DNL Effort

The DNL project has been divided into three parts, Language Definition, DNL Interpreter Design and Development and DNL Networking Part. The three parts are being developed by three DNL team players respectively. Xiaoning Yan has defined the grammar of the language and developed a language parser [16]. Pui-Li Li is responsible for developing the DNL networking part [10]. In this work, we discuss the design and implementation of the interpreter for DNL on a single processor. In other words, the purpose of this thesis is to develop a prototype language interpreting system without considering the networking issues. Pui-Li Li is supposed to use this interpreter to interpret DNL expressions in the networking part.

Each part of the DNL system has a "hidden" implementation. The DNL networking part concerns how information is exchanged between servers. The language definition part hides language syntax and grammar. DNL interpreter design and development deals with relation operations, set operations, and DNL data retrieving algorithms. Using the principle of information hiding in software engineering, each module of the DNL processing system will not change if the "secret" of that module does not change. The syntax is the major part of the interfaces among the three components. The semantics is the essential for both the single processor interpreter implementation and the network aspects.

[1] Parnas proposed this idea in our group meeting in May 1999.

1.4 Thesis Scope

The objective of this thesis is to design a language interpreter to interpret Data Network Language. This language interpreting system is composed of two parts. The first part works as a simple compiler for Data Network Language by using Lex and Yacc. Given a Data Network text file, the first part of the DNL Interpreting System can compile it and translate it into C++ code. The second part of the DNL Interpreting System works as DNL Application Program Interface (DNL API). This part provides the functions and programs evaluating the C++ codes that the first part produces. The final results can be evaluated by executing the C++ code.

In the first part of the Interpreter System, we use Lex as the compiler's lexical analyzer tool to read the input stream and use Yacc to generate the parser programs to check Data Network Language grammar. The Lexical Analyzer is not part of this thesis. It has been defined in [16]. The language grammar is also defined in [16]. Part of the Yacc file is a part of this thesis because Yacc file is a part of the DNL compiler. Once Yacc recognizes the grammar rules, the Interpreter System uses syntax-directed translation algorithms to translate Data Network Language into C++ code.

The second part of this thesis is called Data Network Language Application Programming Interface (DNL API). This DNL API can either stand alone as a software library or be used as a part of the DNL Interpreter system. This part provides the user the final Data Network Language query results. The interpreter system interprets DNL based on the definitions which are documented in [16].

In the remaining chapters this thesis reviews the syntax and semantics of DNL operations, and uses two examples to illustrate how to use the DNL Compiler and the DNL Application Program Interface.

1.5 Outline of This Thesis

Chapter 2 reviews the terminology that is used in this thesis. Chapter 3 describes the compiler tool Lex and Yacc. Chapter 4 reviews the syntax and semantics of the DNL operators. The design of the DNL interpreting system including a DNL compiler Application Program Interface are presented in Chapter 5. Chapter 6 gives two examples to illustrate the use of DNL interpreting system. Chapter 7 discusses the results of the work including limitations of this work. It also proposes some future work.

Chapter 2

Review of Notation and Terminology

This chapter reviews the notation and terminology used in this thesis. The definition of these notations and terminologies is adopted from [12] and [8].

2.1 Sets and Tuples

A *set* is a well-defined collection of different elements. The elements may be natural numbers, names, or whatever objects are of interest to the specifier. A set can be finite or can be infinite.

There are two methods to specify a set, either by explicitly listing all its elements or by comprehensive specification. For example, $\{1, 2, 3, 4, 5\}$ is an explicit listing of the set containing those integers larger than 0 and smaller than 6. The general form of comprehensive specification is like the following:

$$\{TERM \mid Predicate, Signature\}$$

Thus, the comprehensive specification $\{x \mid x > 0 \wedge x < 6, x \in \mathbb{N}\}$ denotes the set of natural numbers $\{1, 2, 3, 4, 5\}$.

Within any given set A there exist other sets which can be obtained by removing some of the elements of A. These are called *subsets* of the set A.

We assume the existence of a finite set of values called the universe U. A *simple tuple* is an ordered list of one or more members of U. A *simple n-tuple* is an ordered list of n members of U. We make no distinction between a simple 1-tuple and member of U. A *tuple* is an ordered list of one or more simple tuples. An *n-tuple* is a tuple containing n elements, each of which is a simple tuple. A 1-tuple whose element is a simple n-tuple is the same as that simple n-tuple.

For example, "(a, b, c, d)" represents a simple 4-tuple, and "$((a, b), c, d)$" represents a 3-tuple that is not a simple 3-tuple.

2.2 Relations, Domain, and Range

Given two sets S and T. The *Cross product* or *Cartesian product* of S and T, noted $S \times T$, is defined as

$$S \times T \triangleq \{(a, b) \mid a \in S \wedge b \in T\}$$

A relation R on $A \times B$ is a subset of the *Cartesian product* of $A \times B$, that is, $R \subseteq A \times B$.

$$domain(R) = \{x \mid \exists(y \mid y \in B \wedge (x, y) \in R\}$$

$$range(R) = \{y \mid \exists(x \mid x \in A \wedge (x, y) \in R\}$$

A *tuple* is an element of the *Cartesian product.* An *n-ary relation* is a set of n-tuples. A *binary relation* is a set of 2-tuples. The *domain* of a binary relation, denoted as $domain(R)$, is the set of values that appear as the first element of R. The *range* of binary relation R, denoted as $range(R)$, is the set of values that appear as the second element of R. A member of the binary relation R can be denoted as (x, y), where $x \in domain(R)$ and $y \in range(R)$.

2.3 Attributes, Tuple Index, and Templates

This section illustrates our use of the concepts defined above and terminologies that we use in this thesis.

Given a relation $R \subseteq \Pi_i X_i$ $(1 < i < m)$, an X_i is an *attribute* of the relation R or given an element $(x_1, x2, ...x_i, ...x_m)$ of relation R, an x_i is an *attribute* of the element.

For example, the following is a relation,

Item	Unit Price	Date
Fuji Apple	$ 1.29	12/04/00
Royal Delicious	$ 1.49	12/06/00
Panama Banana	$ 0.49	12/03/00
Florida Orange	$ 0.99	12/05/00

Figure 2.1: A Relation

An *attribute* corresponds to a column of such a table and a *tuple* to a row. An *attribute* is an identifier for a set in a relation. In Figure 2.1, the attribute "Unit Price" identifies the set that are listed in relation R in position 2, and also identifiers the second tuple "$1.29" to the first row.

A *tuple index* is used to identify an element in a tuple. Given n-tuple $t_p = (t_1, t_2, t_3, ...t_n)$, we define tuple inductively as follows:

(1) Base:
Let $j \in [1, 2, 3, ...n]$, j is a tuple index of representing the jth tuple of t_p.

(2) Induction:
Let $i \in [1, 2, ...n]$ be a tuple index for t_p, and $t_i = (C_1, C_2, ...C_j, ...C_m)$ and $j \in [1, 2, ...m]$, then $i.j$ is tuple index for t_p and $i.j$ represents C_j.

For example, in tuple $(((a, b), c), d)$, tuple index 1 denotes $((a, b), c)$, 2 denotes d, 1.1 denotes (a, b), 1.2 denotes c, 1.1.1 denotes a and 1.1.2 denotes b.

A *template* is an n-tuple of tuple indices or a template with a tuple index, and can be represented by following BNF,

<template> ::= (<TupleIndex>)
| (<TupleIndex>, <template>)

2.4 Functions and Predicates

A *function* F is a binary relation with the following property: $\forall x$, if $(x, y) \in F$ and $(x, z) \in F$, then $y = z$. Like sets and relations, there are at least two ways to specify a function, either by explicitly listing the relationship of the domain and range, or by a comprehensive mathematical specification. For example, $\{2x \mid x > 1 \wedge x < 6, x \in \mathbb{N}\}$ and $\{(2,4),(3,6),(4,8),(5,10)\}$ represent the same function.

A *predicate* is a function whose range contains no members other than *true* and *false*.

2.5 Function Applications and Terms

A *function application* is a function name together with its actual arguments, which are terms. For example, if f is a function, and x and y are terms, $f(x, y)$ is a function application, f is a function name, and x and y are arguments.

A *term* is a constant, variable, or function application.

2.6 Primitive Relation and Predicate Expression

Primitive relations include a small set of relations, such as $<$, $>$ and $=$. All primitive expressions are *predicate expressions.* If P and Q are predicate expressions and x is

a variable, then $(\forall x, P), (P \wedge Q), (P \vee Q)$, and $(\neg P)$ are also predicate expressions. The predicate expressions $((\neg P) \vee Q)$ and $(\neg(\forall x, \neg P))$ can also be written in an equivalent format:

$$((\neg P) \vee Q) \equiv (P \Rightarrow Q)$$

$$(\neg(\forall x, \neg P)) \equiv (\exists x, P)$$

2.7 Set and Relational Algebra

From [5], we adopted the following definition. For given set S and T,

(1) Subset:

$$S \subseteq T \equiv \{\forall x \mid x \in S \Rightarrow x \in T\}$$

(2) Union:

$$x \in S \cup T \equiv x \in S \vee x \in T$$

(3) Intersection:

$$x \in S \cap T \equiv x \in S \wedge x \in T$$

(4) Difference:

$$x \in S - T \equiv x \in S \wedge x \notin T$$

(5) Cardinality:
The *cardinality* of a set is the number of elements in the set.

(6) Composition:
From [3], we adopted the definition of relation composition.
Let R_1 be a relation on $A \times B$ and R_2 a relation $B \times C$. Then we define a relation $R_1 \circ R_2$ on $A \times C$, called the *composite* of R_1 and R_2, as follows: $(x, z) \in R_1 \circ R_2$ if and only if there is a $y \in B$ such that $(x, y) \in R_1$ and $(y, z) \in R_2$.

2.8 Inductively Defined Predicates

An *inductively defined predicate* (IDP) [13] on <type> is defined as the characteristic predicate of a set, S, which is formed in the following way.
Given a triple, $\{I, G, Q\}$, where:

I is a finite set of elements of <type>;
G is a function, G: <type> $\rightarrow$ <type>;
Q is a predicate on <type>, and

$$\forall x \in I \wedge j, m \in \mathbb{N}, \exists m, ((\forall j, (0 < j < m \Rightarrow G^j(x) \in \mathit{domain}(G))) \wedge \neg Q(G^m(x)))).$$

$$G^j(x) \equiv G(G^{j-1}(x)) \text{ and } G^1(x) \equiv G(x)$$

Then S is the smallest set formed by the following rules:

(1) $I \subseteq S$;

(2) $\forall x \in S[Q(x) \Rightarrow G(x) \in S]$.

For example, an IDP $S = \{I, G, Q\}$
$I = \{1\}$, $G(i) = i + 2$, and $Q = (i < N)$,
defines the odd natural number set smaller than N[1].

2.9 Data Tables

A *Data Table* is a table used to store relational data and represent the relations between data. The table in Figure 2.1 is a data table.

2.10 Tabular Expressions and the TTS

In section 2.1 and 2.5 we have shown that there are at least two methods to represent a set. Both of the methods are eligible representation in DNL Interpreter. Users can

[1]Note: A set defined by an IDP should always be finite.

$$f(x,y) = \begin{cases} 0 & \text{if } x \geq 0 \wedge y = 10 \\ x & \text{if } x < 0 \wedge y = 10 \\ y^2 & \text{if } x \geq 0 \wedge y > 10 \\ -y2 & \text{if } x \geq 0 \wedge y < 10 \\ x+y & \text{if } x < 0 \wedge y > 10 \\ x-y & \text{if } x < 0 \wedge y < 10 \end{cases}$$

Figure 2.2: Traditional expression $f(x, y)$

$f(x,y) =$ H_2

	$y = 10$	$y > 10$	$y < 10$
$x \geq 0$	0	y_2	$-y_2$
$x < 0$	x	$x + y$	$x - y$

H_1 G

Table 2.1: Tabular expression $f(x, y)$

choose either of the two ways they like to use in DNL. Instead of redesigning how to represent comprehensive mathmatical specification in DNL, we adopt the way that the Table Tool System uses to represent mathmatical expression as the standard representation in the DNL Interpreter.

Mathematical expressions can be represented by tabular expressions. Tabular expressions are multi-dimensional expressions. Tabular expressions are constructed recursively from scalar expressions and grids [11]. A tabular expression consists of a main grid G and the header grids H_1, H_2, ..., $H_{dim(G)}$, where $dim(G)$ is the dimemsionality (the number of dimensions) of the grid G [9].

For example, the expression $f(x, y)$ in Figure 2.2 can be represented by the tabular expression in Table 2.1.

The Table Tool System (TTS) is a software tool system to support the use of tabular expression. TTS consists of a set of tools which include a construction tool, editor tool, printing tool, composition tool, type checking tool, etc. The TTS is

continually being developed by Software Engineering Research Group (SERG) at McMaster University. A detailed description of the TTS is provided in the TTS Developer's Guide [6].

Chapter 3

Software Tools and Approaches

This chapter gives a brief overview of the tools that we used in our approach to the DNL Interpreter Design. More information about these software tools can be found in [2].

3.1 Lex and Yacc

Lex is a software tool that recognizes different strings of characters and determines whether or not they satisfy certain characteristics. This tool can be used to build a lexical analyzer in the compiler.

The lexical analyzer is the first phase of a compiler. Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis. To build a lexical analyzer, a lex specification consists of two sections, a specification section and an action section. In the specification part you need to specify the patterns you are interested in. In the action part you need to provide subroutines to do things once the lexical analysis recognizes a string matching your specification. Lex stores every character string that it recognizes in a character array called *yytext*[].

Yacc ("Yet Another Compiler Compiler") generates a parser to ensure that input is syntactically correct and to call C language routines when a syntactic element has

been recognized. This process is called parsing. Parsing is the process of calling the low level input scanner, called a lexical analyzer, and then determining if a string of tokens can be generated by grammar rules. When one of the rules is recognized, Yacc invokes a user's action. The actions are usually fragments of C language code. They can return values and make use of values returned by other actions.

To prepare a parsing specification, the following needs to be specified [2]:

- A set of rules to describe the elements of the input;
- The code to be invoked when a rule is recognized;
- Either a definition or declaration of a low-level routine to examine the input.

With each grammar rule, an action is performed when the rule is recognized. An action is an arbitrary C language program. An action is specified by one or more statements and subroutines enclosed in curly braces, ("{" and "}").

Yacc turns the specification file into C language procedures.

3.2 Application Programming Interface

In a general sense, an Application Programming Interface (API) is a library of programming tools used by software developers to write applications that are compatible with a specific user environment. With an application programming interface, software developers can use standard functions to do complicated jobs without knowing the details of algorithms and implementation.

There are several reasons why it is useful to define a standard API for a DNL kernel:

- Widely dispersed programming language can link with DNL through interface adapter (translator) programs without changing the existing applications.

- With API specification, it is easy to make sure that code works when combined with other applications.
- Changes and fixes in DNL can be greatly absorbed inside the DNL kernel, without rewriting the user's applications.
- People with limited time and resources can still contribute significant functionality to their applications.
- Users only need to learn ready-to-be-used functions and components without a complete understanding of their implementation.
- The functionalities of DNL can be added without changing existing DNL applications.

With API technology, the use of DNL kernel looks so simple as binary relation operations. All set and relation functions defined in [16] map C++ functions in DNL API. For example, the user can use C++ statement

"Set<int> IntegerSet1, IntegerSet2(Set<int> s);"

to create two integer sets. One is created by a default empty integer set and the other is created by copying from the existing integer set s. We can use another statement

"IntegerSet1 == IntegerSet2;"

to compare whether IntegerSet1 has the same member as IntegerSet2[1]. The detail API specification is defined in Chapter 5.

[1]DNL API is defined in C/C++ language. The syntax of DNL API has some difference from the syntax of DNL.

Chapter 4

Review of Data Network Language

This chapter reviews the definitions of Data Network Language (DNL). DNL is based on set theory and relational algebra. The DNL kernel is a set of operations on sets and relations. The DNL and these operations is defined in [16]. We do not discuss why and how we define them in this chapter. For details, we refer the reader to [16].[1]

4.1 Basic Operations on Sets and Binary Relations

4.1.1 Assignment

Assignment, denoted by $\leftarrow$, is a function. This function is used to assign a set or a binary relation to a variable. The type of the variable is restricted by the right operand of the assignment function.

Given a variable, V and a term T,

$$V \leftarrow T$$

assigns the value of term T to the variable V.

[1] Beause the initial work on DNL design was done by a team consisting of Pui-li Li, Xiaoning Yan and myself. This chapter contains the essence of chapter 3 and chapter 4 of [16].

4.1.2 Create

Create is used to construct a set or a binary relation. The user provides all the information as arguments to build a set. This information includes a set name, attribute tuple index, attribute name, attribute data type, and attribute size for all the attributes listed in the set. The general format for *create* is:

$$Create(set_name, (tuple_index, attribute_name, attribute_type, attribute_size)$$
$$[, (tuple_index, attribute_name, attribute_type, attribute_size)[, ...]]);$$

For example,

(1) To create a relation named $Shop$ with $(ItemId, (Price, Date))$,
$Create(Shop, (1, ItemId, char, 10), (2.1, Price, float, 7), (2.2, Date, char, 8));$

(2) To create a name set named $StudentName$,
$Create(StudentName, (1, StudentName, char, 20));$

(3) To create a data table named $StudentFile$,
$Create(StudentFile, (1, StudentId, char, 7), (2, StudentName, char, 20),$
$(3, Address, char, 50), (4, Telephone, int, 10)).$

4.1.3 Insert

This function is used to insert elements, also known as records, into a set or relation. Signature:

$$Set, Element \rightarrow Set$$

Given a set S and an element x,

$$Insert(S, x) = S \cup \{x\}.$$

Examples:
Given a integer set $S = \{1, 2, 3\}$
and $Insert(S, 4) = \{1, 2, 3, 4\}$

Given the relation *Shop* defined above, and an element
$e = (FujiApple, (1.99, 08/04/00))$, you could
$Insert(Shop, (FujiApple, (1.99, 08/04/00)))$.

4.1.4 Delete

This function deletes elements from a set or relation.
Signature:

$$Set, Element \to Set$$

Given a set S and an element x,

$$Delete(S, x) = S - \{x\}.$$

Examples:
Given a integer set $S = \{1, 2, 3\}$
and $Delete(S, 3) = \{1, 2\}$
Given a relation *Shop*, and an element $e = (FujiApple, (1.99, 08/04/00))$,
$Delete(Shop, (FujiApple, (1.99, 08/04/00)))$.

4.1.5 Union

This function computes the union of two sets.
Signature:

$$Set, Set \to Set$$

Given two sets S_1 and S_2,

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

For example,
Given two sets $S_1 = \{a, b, c\}$ and $S_2 = \{c, d, e\}$,
$Union(S_1, S_2) = \{a, b, c, d, e\}$

4.1.6 Intersection

This function is to compute the intersection of two sets.
Signature:

$$Set, Set \rightarrow Set$$

Given two sets S_1 and S_2,

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

For example,
Given two sets $S_1 = \{a, b, c\}$ and $S_2 = \{b, c, d, e\}$,
$Intersection(S_1, S_2) = \{b, c\}$

4.1.7 Difference

This function computes a set whose elements are in the first set but not in the second.
Signature:

$$Set, Set \rightarrow Set$$

Given two sets S_1 and S_2,

$$Difference(S_1, S_2) = \{x \mid x \in S_1 \land \neg(x \in S_2)\}.$$

For example,
Given two sets $S_1 = \{a, b, c\}$ and $S_2 = \{b, c, d, e\}$,
$Difference(S_1, S_2) = \{a\}$

4.1.8 Cardinality

This function computes the number of elements in a set.
Signature:

$$Set \rightarrow Integer$$

Given a set S,

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

For example,
Given a set $S = \{a, b, c, d, e\}$,
$Cardinality(S) = 5$.

4.1.9 Domain

This function computes the domain of a relation.
Signature:

$$Relation \rightarrow Set$$

Given a relation R,

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

For example, given a relation
$Price = \{(apple, 1.99), (banana, 0.99), (peach, 2.49), (apple, 2.99)\}$
$Domain(Price) = \{apple, banana, peach\}$.

4.1.10 Range

This function computes the range of a relation.
Signature:

$$Relation \rightarrow Set$$

Given a relation R,

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

For example, given a relation
$Price = \{(apple, 1.99), (banana, 0.99), (peach, 2.49), (peach, 1.99)\}$
$Range(Price) = \{1.99, 0.99, 2.49\}$.

4.1.11 Identity

This function computes the identity relation whose domain is a given set.
Signature:

$$Set \rightarrow Relation$$

Given a set S,

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

For example, given a set $S = \{a, b, c\}$,
$Identity(S) = \{(a, a), (b, b), (c, c)\}$

4.1.12 Composition

This function computes the composition of two relations.
Signature:

$$Relation, Relation \rightarrow Relation$$

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

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

For example, given a relation $R_1 = \{(1, a)(3, b), (2, c)\}$, and a relation $R_2 = \{(a, 10), (b, 20), (d, 30)\}$,
$Composition(R_1, R_2) = \{(1, 10), (3, 20)\}$.

4.1.13 Join

This function is 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: $C \times D$,

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

For example, given a relation $R_1 = \{(1, a)(3, b), (2, c)\}$,
and a relation $R_2 = \{(a, 10), (a, 40), (b, 20), (d, 30)\}$,
Join$(R_1, R_2) = \{(1, (a, 10)), (1, (a, 40)), (3, (b, 20))\}$.

4.1.14 Index

This function computes a new relation from the existing set and a total order relation. The new relation is a set of pairs where the domain of the result relation is called the *index set*, the range of the result relation is called the *indexed set* [7].
Signature:

$$Set, Index_Set, Relation \rightarrow Indexed_Set$$

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

$$Index(S, I, O) = \{(i, x_i) \mid (i, x_i) \in R \wedge (j, x_j) \in R \wedge i < j \Rightarrow x_i O x_j\}.$$

For example: given a set $S = \{1003, 4, 288, 143, 2005, 89\}$
an Index set $I = \{1, 2, 3, 4, 5, 6\}$
a total order relation: $<$ (less than),
$Index(S, I, <) = \{(1, 4), (2, 89), (3, 143), (4, 288), (5, 1003), (6, 2005)\}$.

4.1.15 Restriction

This function restricts a set by a predicate to generate 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,

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

For example, given a set $S = \{2, 4, 5, 50, 78, 90\}$,
a predicate P: $x < 50$,
$Restriction(S, P) = \{2, 4, 5\}$.

4.1.16 Image

This function computes the range set of the given relation by restricting its domain to a given domain.
Signature:

$$Relation, Set \rightarrow Set$$

Given a relation R and a set S,

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

For example, given a relation $R = \{(1, a), (2, b), (3, c), (3, d)\}$, a set $\{2, 3, 5\}$,
$Image(R, S) = \{b, c, d\}$.

4.1.17 PreImage

This function computes the domain set of the given relation by restricting its range to a given range.
Signature:

$$Relation, Set \rightarrow Set$$

Given a relation R and a set S,

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

For example, given a relation $R = \{(1, a), (2, b), (3, c)\}$, a set $S = \{b, c, e\}$,
$PreImage(R, S) = \{2, 3\}$.

4.2 Advanced Operations

4.2.1 OperationOnFunction

Signature:

$$Operator, Function, Set \rightarrow Value$$

Given an operator Op, a function F, and a set $X \subseteq DomainF$,

$$OperatorOnFunction(Op, F(X))_X \equiv$$

(1) if $cardinality(X) = 0$, identity element for Op; ($e.g.$ 0 for $+$, 1 for $\times$)

(2) if $cardinality(X) > 0$, $Op(F(x_1), OperatorOnFunction(Op, F(x)_{x \in \{X-\{x_1\}\}}))$ where $x_1 \in X$.

For example, given a relation $R = \{(a,1),(b,2),(c,3)\}$ and an Op $+$,
$OperatorOnFunction(+, Range(R))$
$= +(1, OperatorOnFunction(+, Range(\{(b,2),(c,3)\})))$
$= +(1, +(2, OperatorOnFunction(+, Range(\{(c,3)\}))))$
$= +(1, +(2, +(3, 0))) = 6$.

4.2.2 RangeDivide

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

$$Relation \rightarrow Relation$$

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

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

For example, given a relation $R = \{(1,(a,b)),(1,(f,g)),(2,(c,d)),(3,(d,e))\}$,
$RangeDivide(R) = \{(1,a),(1,b),(1,f),(1,g),(2,c),(2,d),(3,d),(3,e)\}$.

4.2.3 RangeMerge

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

$$Relation, TupleIndex, OP \rightarrow Relation$$

Given a Relation R, a TupleIndex t, and an operator op,

$$RangeMerge(R, t, op) = \{(x, y) \mid \exists r \in R :$$

$$y = OperatorOnFunction(op, Range(Restriction(R, r.t = x)))\}$$

For example, given a relation $R = \{(a, 1)(a, 2)(b, 4), (b, 2), (c, 3), (c, 4)\}$,
$RangeMerge(R, \text{“}1'', +) = \{(a, 1 + 2), (b, 4 + 2), (c, 3 + 4)\}$
$= \{(a, 3), (b, 6), (c, 7)\}$.

4.2.4 Rearrange

This function is used to reconstruct a relation using a template[2]. The *rearrange* function is defined using the *Dot Operation* and *Rearrange_Tuple*.

Dot Operation ($\bullet$):
Given a tuple $T = (t_i, ..., t_j, ..., t_l)$ and a tuple index $i < j < l$
$T \bullet j = t_j$
For example,
$T = (x, (y, u, (w, z)))$, $j = 2.3$,
$T \bullet j = (T \bullet 2) \bullet 3 = (y, u, (w, z)) \bullet 3 = (w, z)$.

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

[2] Template is defined in Section 2.3.

(1) Base Case:
$Rearrange_Tuple(X, (tm_1, tm_2, ..., tm_n)) = (X \bullet tm_1, X \bullet tm_2, ..., X \bullet tm_n)$
if tm has only one n-tuple index.

(2) Inductive Step
$Rearrange_Tuple(X, (tm_1, tm_2, ..., tm_j, ..., tm_n))$
$= (X \bullet tm_1, ..., (X \bullet \alpha_i, X \bullet \alpha_j), ..., X \bullet tm_n)$
if $tm_j = (\alpha_i, \alpha_j)$.

Rearrange:
Signature:

$$Relation, template \rightarrow Relation$$

Given a Relation R, a template tm, we have

$$Rearrange(R, tm) = \{X \mid \exists Y \in R : X = Rearrange_Tuple(Y, tm)\}.$$

Example: given a relation $R = \{(a, (c, e)), (b, (f, g)), (c, (h, j))\}$,
a template $tm = ((2.1, 1), 2.2)$,
$Rearrange(R, tm) = \{((c, a), e), ((f, b), g), ((h, c), j)\}$.

4.2.5 CreateAbsSRF

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

$$Set, Set, Predicate \rightarrow Set$$

Given two sets S_1, S_2, and a Predicate P on the elements of S_1 and S_2, in which $x \in S_1$ and $y \in S_2$,
$CreateAbsSRF(S_1, S_2, P)$, we can get a relation,

$$R = \{(x, y) \mid x \in S_1 \wedge y \in S_2 \wedge P(x, y)\}$$

If we change P to be a function F, $CreateAbsSRF(S_1, S_2, F)$, we can get a function,

$$R = \{(x, y) \mid x \in S_1 \wedge y \in S_2 \wedge y = F(x)\}$$

If we use only one set, $CreateAbsSRF(S_1, \emptyset, P)$, we can get a set,

$$S = \{x \mid x \in S_1 \wedge P(x)\}$$

Examples, given two sets $S_1 = \{1, 2, 3, 4\}$, $S_2 = \{2, 3, 5, 6\}$, and $>$,
(1) $CreateAbsSRF(S_1, S_2, >) = \{(3, 2), (4, 2), (4, 3)\}$
(2) $CreateAbsSRF(S_1, S_2, (y = x + 1)) = \{(1, 2), (2, 3), (4, 5)\}$
(3) $CreateAbsSRF(S_1, \emptyset, (x > 2)) = \{3, 4\}$

4.2.6 ArithmeticComp

This function is to apply an arithmetic operation to a specific tuple.
Signature:

$$Relation, TupleIndex, ArithmeticOperator, Value \rightarrow Relation$$

Given an Arithmetic Operator Op, Relation R, a TupleIndex i on R, and a value v, then apply the arithmetic operator to the element designated by the i of R using v.

$$ArithmeticComp(R) = \{(x, y) \mid \exists (x, z) \in R \wedge y = Op((x, z) \bullet i)\}$$

For example: given a relation R,

$$R = \{(store, (item, price)) \mid store \in set_of_store \wedge (item, price) \in item \times R^+\}.$$

Assume we have:

$$R = \{(nofrills, (apple, 1.25)), (fortino, (apple, 1.45))\},$$

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

4.2.7 Reduction

This function is for applying functions to many variables.
Signature:

$$Function, Variablelist \rightarrow Value$$

Given variables $V_1, V_2, ...V_i, ...V_n$, a function Op, we have,

$$Reduction(Union, V_1, V_2, ...V_i, ...V_n) = Op(...Op(...Op(V_1, V_2)..., V_i)..., V_n).$$

For example,
$Reduction(Union, S_1, S_2, S_3) = Union(Union(S_1, S_2), S_3)$.

4.3 Data Access Operation

4.3.1 GetAttribute

This is a function to get attribute information from a relation set or a data table. The definition of attribute is defined in section 2.3.

Given a Relation R and a TupleIndex on R, get the current element's attribute information corresponding to the TupleIndex.
Signature:

$$Relation, TupleIndex \rightarrow AttributeInfo$$

For example, given relation $R = \{(item, (price, date))\}$, and TupleIndex 2.1,
$GetAttribute(R, 2.1) \rightarrow price$.

In this chapter, we discussed the operations in DNL. DNL grammar will not be discussed here. We refer the reader to [16] to get the DNL grammar. The design and implementation of DNL focuses on these operations we defined above. In chapter 5, we present the design of DNL interpreter tool.

Chapter 5

Design of the Data Network Language Interpreter

We developed an interpreter system for our Data Network Language which interprets the functions we described in Chapter 4. This chapter discusses the design of the Data Network Language Interpreter. The first section of this chapter gives the overview of the interpreter system. The second section of this chapter describes the informal requirements, user interface, and anticipated changes of this language. The third and forth subsection describe the module decomposition and the access programs for each module. The final section presents the algorithms used for the Compiler part of Data Network Language design.

5.1 System Overview

Data Network Language Interpreter System has been developed for computing information from DNL databases. Like most other programming languages, all expressions are saved in a text file. The file is called a DNL expression. The query file will be compiled into an executable file and users run the executable file to get the results they want. From a user's perspective, the process of querying information from a DNL database involves the following three steps:

1) Prepare DNL queries in a text file;

2) Compile the text file using DNL Interpreter to generate an executable file;

3) Run the executable file using the current DNL data to get the final results. Step 3 can be repeated whenever the database changes.

Examples of the text files and the process of how to get executable files are illustrated in Chapter 6.

From the designer's perspective, the process of interpreting a DNL query includes the following steps:

1) Check the grammar rules and translate the DNL text file into C++ source code including the API functions while parsing the DNL text file. Report errors in the output during this step;

2) Link all the C++ functions generated by the first step with the functions prepared by the DNL API. Compile the C++ source code to an executable file;

3) Provide the final results generated by executing the executable codes.

The process is illustrated in Figure 5.1.

5.2 Requirements (Informal)

The Data Network Language Interpreter system contains two parts. The first part is a compiler which was built by using Lex and Yacc. For a given DNL query file, the compiler will translate DNL statements into C++ code. The second part of the interpreter system provides DNL API. It contains programs and functions that can be called to resolve all the operations the compiler translated.

The following figure also illustrates the design requirements.

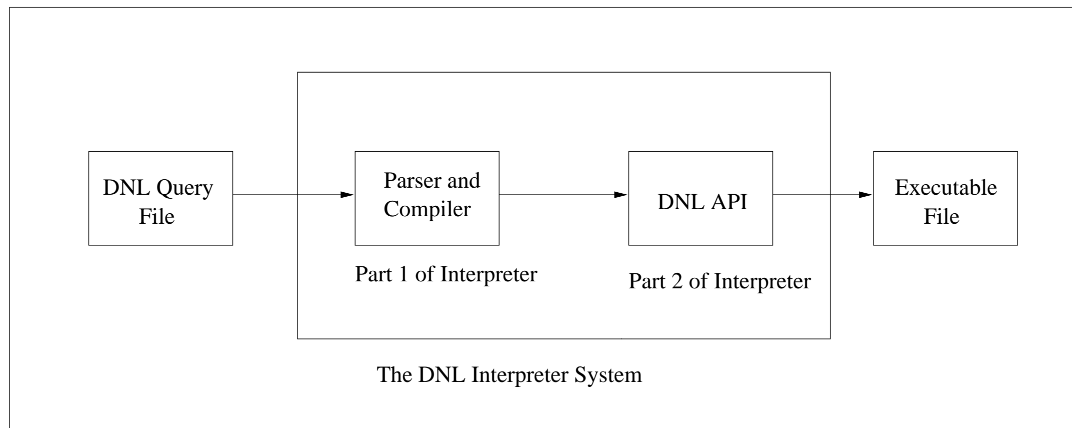


Figure 5.1: Requirement of Data Network Language Interpreter System

5.2.1 User Interface

Like the system requirements, the user interface is also divided into two parts. Both of the interfaces are command line interfaces.

User Interface for Part 1

The input of the Data Network Interpreter system is a DNL text file. The first part of the user interface provides the information to check the DNL text file's syntax and grammar. To parse and compile a Data Network Language query file, the user types:

parser *filename*

where *filename* is the name of Data Network Language query file.

The interpreter system then parses the file and generates C++ source code. If there are some errors on file, for example at line 4, the statement as following and there is a ")" missing,

```
R1 <- Domain(R01;
```

the parser will give report it on the console:

```
Line 4 column 16 error, before or at ';', ')' is expected.
```

If the file is correct, the parser will give the following report:

```
*****DNL parser has recognized this program!****
```

and three output files will be generated by the compiler. They are a C++ variable and function declaration file (*decl.txt*), a C++ function file (*func.txt*) which includes the API functions to be called by the main program file, and a main program file (*main.txt*). All the three files are C++ source file in step 2.

User Interface for Part 2

In order to get the final results, the user needs to compile and execute the three output C++ source files. The user interface of this part is consistent with the ANSI C++ user interface. For convenience, we provided a default makefile to compile the output C++ files. A user can either choose to write his/her own makefile to compile the C++ source files[1] or use the default makefile. The simplest way is to use the default makefile. The user only needs to type
make
on a Unix console. A default executable file "DNL" will be generated. If some users like to write their own makefile to generate the executable file, they can refer to the "Unix Programmer's Guide" and Appendix A.

5.2.2 DNL Interpreter Application Programming Interface

The DNL Interpreter system provides an Application Programming Interface (API) for all the functions defined in Chapter 4. Anyone can call these set and binary relation functions directly by using the interface. The way to use this interface is like

[1] See Appendix A for how to write a makefile for DNL Interpreter

using any other C/C++ library and API. The details about these function interfaces are described in the module guide and module interface section (Section 5.4).

5.2.3 Anticipated Changes

It is expected that the following parts are likely to change within the useful life of DNL Interpreter System.

- The Definition of Data Network Language
- The Interpreter System Design
- The User Interface
- The Input and Output File Format

The software has been designed to make these changes.

5.3 Module Decomposition

Data Network Language Interpreter System is composed of a set of modules. This section gives an overview of the top level module "uses" relations in the DNL Application Programming Interface. To make the design easy to understand and change, the DNL interpretation system has been decomposed into a set of modules using the "information hiding" principle.

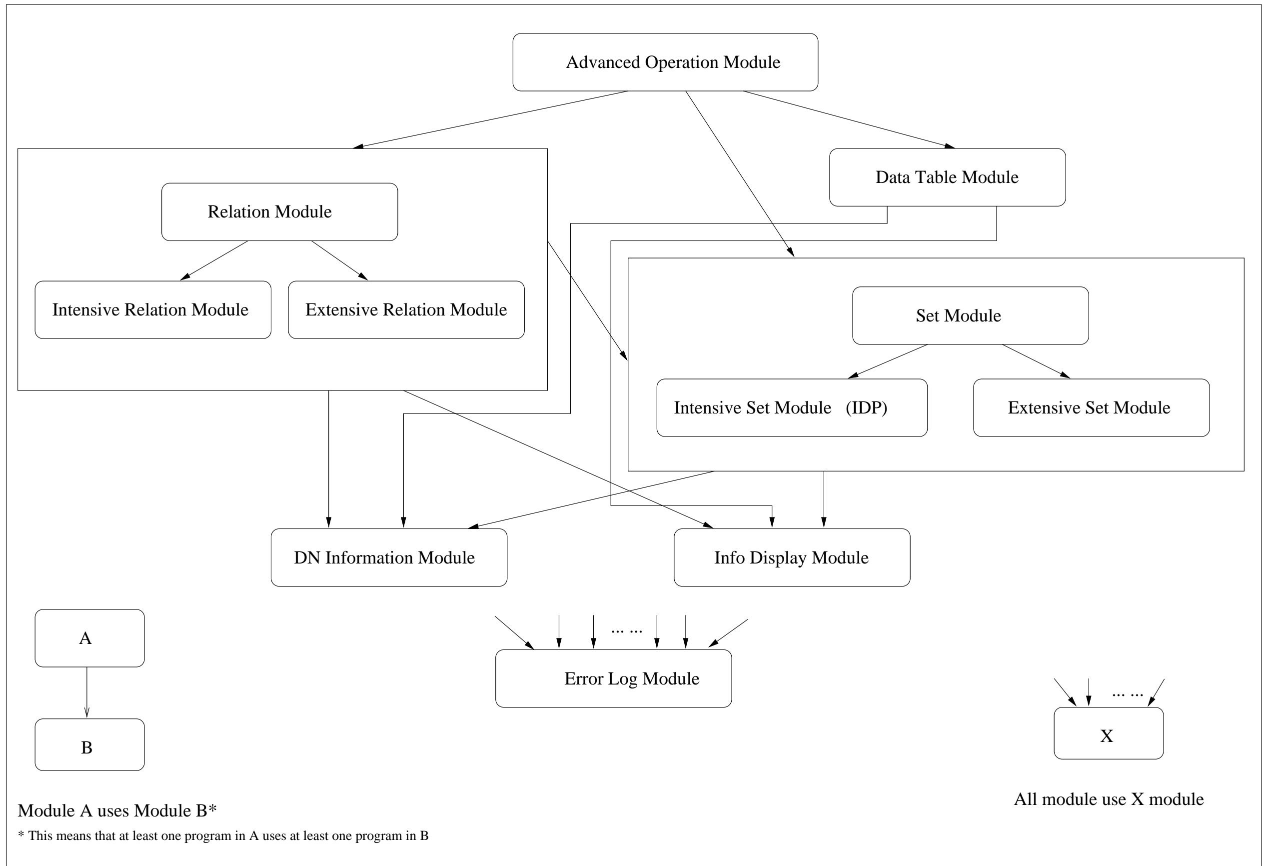


Figure 5.2: DNL Top Level Module and Uses Relation

5.4 Module Guide and Module Interface

This section we describe each module's functionality and the access program interfaces.

5.4.1 Error Module (DN_Error.c)

The Error module is responsible for handling errors and exceptions. This module hides the secrets of how to handle errors and exceptions.

Table 5.1: Access Program Interface for Error Module

Access Program	Return Type	Description
Error(int errorType)	char*	Give an error type and return error message to user and also write to log file.

5.4.2 Display Module (Display.c)

The Display module is responsible for displaying set, relation, and data table. This module hides the secrets how to display different kinds of data structure.

Table 5.2: Access Program Interface for Display Module

Access Program	Return Type	Description
Display(Set &s)	void	Display the set structure and data.
Display(Relation &r)	void	Display the relation structure and data.
Display(DTable &dt)	void	Display the data table structure and data.

Part 1 Run Time Module

The run time modules belong to the DNL API Part.

5.4.3 Information Module (DN_Info.c)

The Information module is responsible for basic information storage. This module hides the data structures for linked lists, attributes, data tables, and the algorithms to retrieve this information. This module is further divided into four submodules as follows.

Attribute (class Field)

The Attribute module hides the data structure to represent attributes. We call one attribute a Field of the DNL database. A field contains a field name, the field data type, and the field size. Field name is the identifier of this field and field size is used for allocating memory for the attribute.

Table 5.3: Access Program Interface for class Field

Access Program	Return Type	Description
Field()	Field	Create an empty field.
Field(Name, Type, Size)	Field	Create a new field with name, type and size.
Field(const Field&)	Field	Copy a new field with the same information as the existing field.
~Field()	void	Delete the field structure.
GetName()	Name	Get this field's name[2].

continued on next page

[2]In C++, non-static functions are invoked by a intance of the class. In the following function description "this field" and "the field" refer to the instance of the field class that invokes the function. The same terminology applies to the instances in other classes

continued from previous page

Access Program	Return Type	Description
GetType()	Type	Get this field's data type.
GetSize()	Size	Get this field's size.
SetName(Name)	void	Update this field's name.
SetType(Type)	void	Update this field's type.
SetSize(Size)	void	Update this field's size.
Operator == (Field &fd)	Bool	Compare whether this field has the same information as field fd.
Operator = (Field &fd)	Field	Assign field fd's information to this field. This function is same as the copy constructor Field(const Field&).

Node (template class Node)

The template class Node hides the data structure to represent a double linked list node. A node contains a data object and two pointers linked to a previous node and a next node.

Table 5.4: Access Program Interface for class Node

Access Program	Return Type	Description
Node(const T& item, Node* next, Node* pre)	Node<T>	Create a new node.
Node* getNext()	Node*	Returns the next node.
Node* getPre()	Node*	Returns the previous node.
setNext (Node *ptr)	void	Sets the next node as ptr.
setPre (Node *ptr)	void	Sets the previous node as ptr.

Linked List (template class List)

The template class List hides the data structure to represent a double linked list and algorithms to manipulate it. A double linked list has a node pointer that points to the list head, another pointer points to current position, and a integer to represent the size of the list.

Table 5.5: Access Program Interface for class List

Access Program	Return Type	Description
List()	List	Create a new empty list.
List(const List&)	List	Copy a list from the existing list l.
~List()	void	Destroy list and free resource allocated.
Insert(T& item)	bool	Insert the object item into the list. Returns false if not successful.
Delete (T& item = CurItem)	bool	Delete the item from the list. Default item will be the current one. Return false if not successful.
NextItem()	T&	Return the next object in the list, return null if none is there.
PreItem()	T&	Return the previous object in the list, return null if none is there.
CurItem()	T&	Return the current object in the list, returns null if the list is empty.
Skip(int i=1)	bool	Move the current pointer to next i positions. Default move one position. If cannot move, return false.

continued on next page

continued from previous page

Access Program	Return Type	Description
IsMember(T &item)	bool	Return true if item is a member in the list, otherwise, return false.
getSize()	Size	Return the number of objects in the list.
IsEmpty()	bool	Return whether the list is empty.
getHead()	Node<T> *	Return the head of linked list.
getCurrent()	Node<T> *	Return the current object of the linked list.
CurrentPos()	int	Return the current object's position.
ClearList()	void	Delete all the objects in the list and release the allocated resource.
Reset(int pos=1)	bool	Reset the current pointer's position at pos, default position is the head.

Data Table File Structure (class FStruct)

The class FStruct hides the data structure to represent the data table and algorithms to manipulate it. A data table has a table name as identifier of the table, a table size, and a field list which contains all the attributes.

Table 5.6: Access Program Interface for class FStruct

Access Program	Return Type	Description
FStruct()	FStruct	Create a new empty data table structure.

continued on next page

continued from previous page		
Access Program	Return Type	Description
FStruct(FStruct &fs)	FStruct	Copy a file structure from the existing file structure fs.
~FStruct()	void	Free all allocated resource.
GetSize()	Size	Return the record length.
GetFieldNum()	int	Return number of fields in the data table structure.
GetFieldName (int ith)	Name	Return the ith field's name.
GetFieldType(int ith)	Type	Return the ith field's data type.
GetFieldSize(int ith)	Size	Return the ith field's length.
GetFieldType (Name nm)	Type	Return the field's type via name.
GetFieldSize (Name nm)	Size	Return the field's size via name.
SetSize(int size)	void	Resize the file structure record size.
SetFieldName (int ith, Name nm)	void	Reset the ith field's name to nm.
SetFieldType (int ith, int tp)	void	Reset the ith field's type to be tp.
SetFieldSize (int ith, Size sz)	void	Reset the ith field's size to be sz.
SetFieldType (Name, Type)	void	Reset the field's type via name.
SetFieldSize (Name, Size)	void	Reset the field's size via name.
AddField(Field &fd)	Bool	Add a new field fd into the field list.

5.4.4 Record Module (DSFile.cc)

The class Record hides the data structure used to represent a record. A record is an n-tuple in a set or data table.

Table 5.7: Access Program Interface for class Record

Access Program	Return Type	Description
Record()	Record	Create a new empty record.
Record(char*)	Record	Create a new string record.
Record(int len)	Record	Create a new record with length len.
~Record()	void	Free allocated resource.
operator==	bool	Compare two records, return true if they have the same content, otherwise return false.
operator= (const Record& rd)	Record&	Assign this record with the value from record rd.
setSize(int i)	void	Reallocate record size to be i.
getSize()	int	return the record size.

5.4.5 Set Module (Set.h)

The DNL Set Module hides the data structure for representing DNL sets and algorithms for manipulating them. A set is a collection of objects of a specified type. The secret of this module is how DNL sets are represented in the DNL processing system. There are several ways to represent sets. Sets can be represented by a collection of individual objects or by an expression. For example, $\{1, 2, 3, \ldots 10\}$ and $\{x \mid x \in \mathbb{N} \land x < 11\}$ are equivalent. The former one is called the extension. The later one is called the intension. In this module we can use either of the two ways or a combination of the two. This module uses two sub-modules: the Extensive Set Module (class ExSet<T>) and the Intensive Set Module (class Idp<T>).

Table 5.8: Access Program Interface for class Set

Access Program	Return Type	Description
Set(Name nm)	Set	Create a new set with name nm.
Set(const Set<T> &x)	Set	Copy the existing set x.
~Set()	void	Free allocated resource.
operator =(const Set<T> &s)	Set	Assign this set from the existing set s.
IsRelation()	bool	Check whether this set is a relation. Return true if yes. Otherwise, return false.
IsMember(const T& elt)	bool	Check whether element elt (type T) is a member of the set.
operator== (Set<T>& s)	bool	Compare whether set s is equal to this set, return true when they are equal, otherwise return false.
IsSubSet(Set<T>& s)	bool	Check whether set s is a subset of this set.
Union(Set &x)	Set	Union set x with this set.
Difference(Set &x)	Set	Delete set x's member from this set.
Intersection(Set &x)	Set	Intersection set x with this set.
Insert(T& elt)	void	Insert a new element elt (type T) into this set.
Delete(const T& elt)	void	Delete the element elt (Type T) from this set.
AssignExSet (ExSet<T> *s)	void	Assign extensive set s as this set's extensive representation.
AssignInSet(Idp<T> *s)	void	Assign intensive set s as this set's intensive representation.

continued on next page

continued from previous page

Access Program	Return Type	Description
getExSet()	ExSet<T> *	Return this set's extensive representation.
getInSet()	Idp<T> *	Return this set's intensive representation.

5.4.6 Extensive Set Module (ExSet.h)

The Extensive Set Module hides the data structures representing a set as a collection of objects of the certain type, and algorithms to manipulate them. This module uses some data structures defined in DN Information module.

Table 5.9: Access Program Interface for class Extension Set

Access Program	Return Type	Description
ExSet()	ExSet	Create a new empty extensive set.
ExSet(const ExSet<T> &x)	ExSet	Copy the existing set x.
~ExSet()	void	Free allocated resource.
operator =(const ExSet<T> &x)	ExSet<T>	Assign a extensive set from the existing extensive set.
IsMember(const T& elt)	bool	Check whether element elt is a member of the set.
operator== (ExSet<T>& s)	bool	Compare whether extensive set s is equal to this extensive set, return true when they are equal, otherwise return false.
Union(ExSet<T> &x)	ExSet	Union extensive set x with this extensive set.

continued on next page

continued from previous page

Access Program	Return Type	Description
Difference (ExSet<T> &x)	ExSet	Delete extensive set x's member from this extensive set.
Intersection (ExSet<T> &x)	ExSet	Compute the intersection set of set x and this set.
Insert(T& elt)	void	Insert a new element elt (type T) into this extensive set.
Delete(const T& elt)	void	Delete the element elt (Type T) from this extensive set.

5.4.7 Intensive Set Module (DN_Idp.h)

The Intensive Set Module hides the data structures representing a set as an Inductively Defined Predicate (IDP) and set algorithms to manipulate the IDP. For example, this module stores sets like $\{x \mid x > 3 \land x < 2000, x \in \mathbb{N}\}$. This module will use IDP data structure defined in TTS.

Table 5.10: Access Program Interface for class IDP

Access Program	Return Type	Description
Idp()	Idp	Create an empty indeuctively defined predicate.
Idp(List<T> ∗ lst, int len, T (∗fp)(T), bool (∗pq)(T))	Idp	Create an inductively defined predicate with an initial list lst, length len, function P fp, Predicate Q pq.
Idp(const Idp<T>& idp)	Idp	Create a new inductively defined predicate from the existing inductively defined predicate idp.

continued on next page

continued from previous page		
Access Program	Return Type	Description
~Idp()	void	Free allocated resource from this inductively defined predicte structure.
first()	T	Return the first element from the inductively defined predicate set.
next()	T	Return the next element from the inductively defined predicate set.
operator()(T elt)	bool	Check whether element elt is a member of the inductively defined predicate. If yes, return true, otherwise return false.

5.4.8 Relation Module

The Relation Module hides the data structure and algorithms for representing and manipulating DNL relations[3]. A DNL relation is a set but the elements of a relation must be pairs. The same idea is used in the relation implementation. A set is a relation only if all elements in this set have a uniform pair structure. If any element of the set has a different structure from the others, it is not a relation but a set. The relation module inherits all the features of the Set Module and plus operations that are only defined for the elements of the set that are pairs.

Like other sets, a relation represented in this module has two representations, an intensive representation and an extensive representation. The relation module has three submodules: Relation Structure Module, Extension Relation Module, and Intension Relation Module.

Figure 5.3 illustrates the secrets in the relation module.

[3]DNL relations are binary relations. The definition of binary relation is in Chapter 4. See [16] for details.

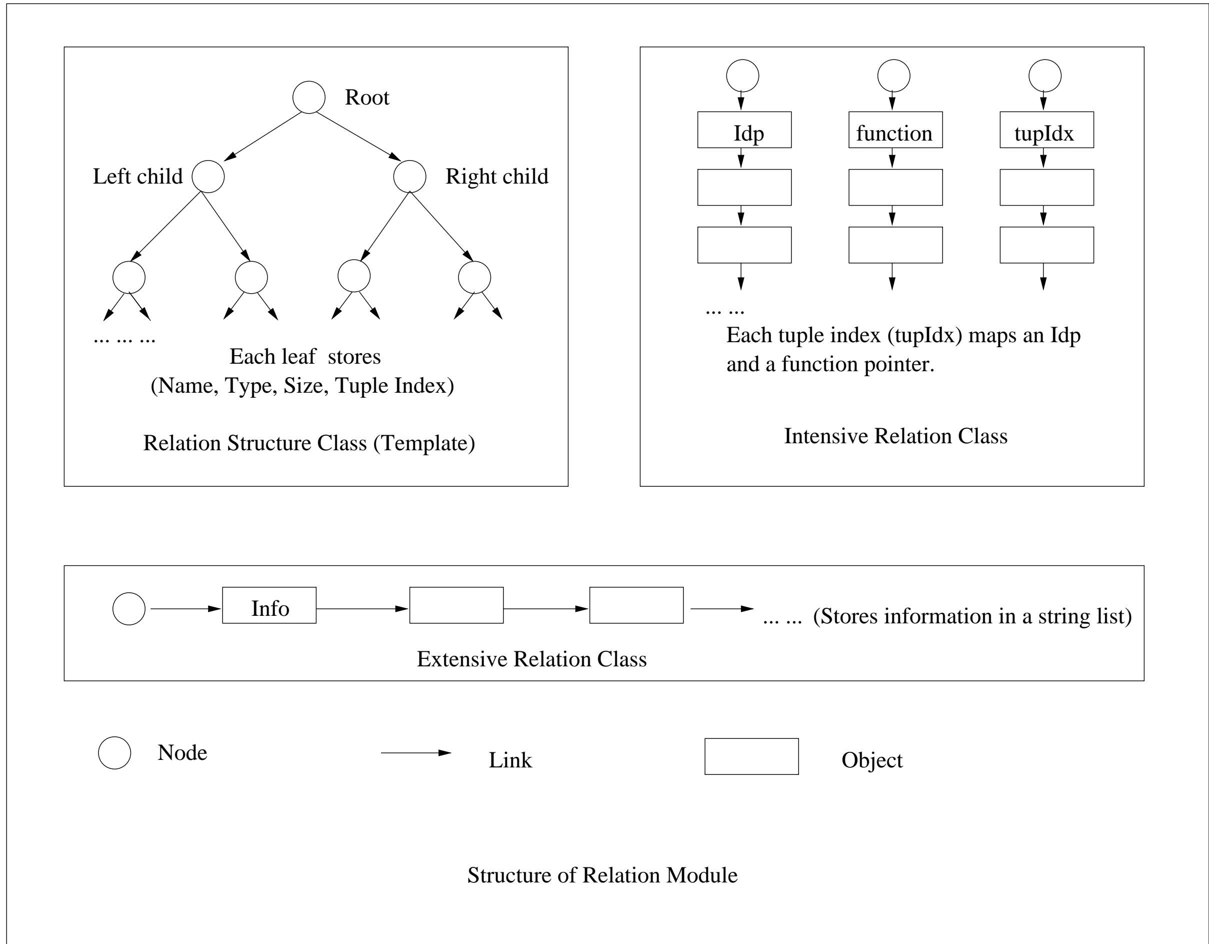


Figure 5.3: Secret of Relation Module

Relation Data Structure Module

The Relation Data Structure module hides the data structure and algorithms used to represent the structure of a relation, also called a template. A binary relation template is a binary tree structure. The leaves of the tree are the tuple node structures. The structure of the domain is described by the structure's left subtree. The structure of the range is described by the structure's right subtree.

A relation data structure is composed by a set of linked tuple node data structures in a binary tree structure. Before we present the interface of relation data structure,

we need to give the data structure of tuple node.

Tuple Data Structure (class TupleDS)

A tuple node structure represents the data structure of a tuple node. To construct this struture, we can reuse the attribute structure (class Field) which we already defined in the information module. The tuple structure is an attribute structure plus a tuple index and three links: left son node, right son node, and parent node respectively. The tuple data structure module hides the data structure and algorithms for representing and manipulating a tuple data structure.

Table 5.11: Access Program Interface for class TupleDS

Access Program	Return Type	Description
TupleDS()	TupleDS	Create a new empty tuple data structure.
TupleDS(TupIdx ti, Name nm, Type tp, Size sz)	TupleDS	Create a new tuple data structure with tuple index ti, name nm, data type tp and size sz.
TupleDS(const TupleDS &tds)	TupleDS	Copy from the existing tuple data structure tds to current tuple data structure.
~TupleDS()	void	Free allocated resources and destroy the tuple data structure.
setIndex(TupIdx ti)	void	Replace the tuple index of current leaf with ti.
getIndex()	TupleIdx	Return the tuple data structure's tuple index.
Left()	TupleDS*	Return the tuple data structure's left child node. Return null if none.

continued on next page

continued from previous page

Access Program	Return Type	Description
Right()	TupleDS*	Return the tuple data structure's right child node. Return null if none.
Parent()	TupleDS*	Return the tuple data structure's parent node.
AssignLeft (TupleDS* tds)	void	Assign node tds to be left child node.
AssignRight (TupleDS* tds)	void	Assign node tds to be right child node.
AssignParent (TupleDS* tds)	void	Assign node tds to be parent node.

Relation Data Structure Module (class RelaDS)

This section we present the relation data structure module which uses the tuple data structure module.

Table 5.12: Access Program Interface for class RelaDS

Access Program	Return Type	Description
RelaDS()	RelaDS	Create an empty relation data structure.
RelaDS(TupleDS *rt)	RelaDS	Create a relation data structure with root rt.
RelaDS (const RelaDS &rds)	RelaDS	Copy from existing relation data structure rds to the current relation structure.
~RelaDS()	void	Free allocated resource, destroy the relation data structure.

continued on next page

continued from previous page

Access Program	Return Type	Description
Insert(TupleDS *tds)	bool	Insert a tuple data structure tds into this relation data structure. Return true if successful, otherwise return false.
Delete(TupleDS *tds)	void	Delete the tuple data structure tds from this relation data structure.
IsExist (TupleDS *tds, Name nm, TupleDS *result)	bool	Check whether the tuple named as nm existed as son node in tuple data structure tds, if yes, assign that node's information to result and return true. Otherwise return false. This function can also be used as one of the get-information functions.
getRoot()	TupleDS*	Return the root node of this tuple data structure.
setRoot (TupleDS *tds)	void	Set the root node of the data structure to be tds.
NodeCopyAddIndex (TupleDS *scr, TupIdx ti, ExRelaDS *dst)	void	Merge all the child nodes of scr to relation data structure dst, add tuple index ti as their tuple index.
NodeCopyDelIndex (TupleDS *scr, TupIdx ti, ExRelaDS *dst)	void	Copy all the child nodes of scr to relation data structure dst, delete tuple index ti from the dst tuple index.

continued on next page

continued from previous page

Access Program	Return Type	Description
TreeToList (TupleDS *scr, List<TupDSL> *dst)	void	Copy the relation data structure scr (Tree Structure) to a list structure dst. This is useful when a user wants to transform a relation structure to be a data table structure using a depth first algorithm.

Intensive Relation Module (class InRelation)

The intensive relation module[4] hides the data structure and algorithms used to represent an intensive relation. An attribute of a intensive relation is represented by a structure of IDP, function (including predicate function) pointer, and a tuple index. An intensive relation is a list of such attribute structures (See Figure 5.3).

Table 5.13: Access Program Interface for class InRelation

Access Program	Return Type	Description
InRelation()	InRelation	Create a new empty intensive relation.
InRelation(const &InRelation<T> inrl)	InRelation	Copy a new intensive reltion from the existing intensive relation inrl.
~InRelation()	void	Free allocated resources and destroy the relation.

continued on next page

[4]The functions in this module are used for implementing class Relation and not directly used by users.

continued from previous page

Access Program	Return Type	Description
Insert(Idp<T>& idp, T (∗function)(T), TupIdx idx)	bool	Insert an Idp, a function, and a tuple index which associates with the Idp and function in the intensive relation. Return true if the tuple index was consistent, false otherwise.
Delete(TupIdx idx)	bool	Delete the Idp and function associated with the tuple index idx. Return ture if tuple index exists, false otherwise.
first()	char∗	Return the first element[5] from the intensive relation. If the intensive relation is empty, return null.
next()	char∗	Return the next element from the intensive relation. Return null if the intensive relation is empty or there is no element that was not returned by the last "next".
operator()(char∗ elt)	bool	Check whether element elt is a member of the inductively defined predicate. If yes, return true, otherwise, return false.

continued on next page

[5]The order is arbitary.

continued from previous page		
Access Program	Return Type	Description
CheckConsistent()	bool	Check if the intensive relation is consistent[6], return true if yes, otherwise return false.

Extensive Relation Module (class ExRelation)

The Extensive Relation Module hides the data structure and algorithms for representing and manipulating the extension representation for a relation. This module inherits from the extensive set module and has some additional functionalities. A user can get the attribute's info and update the attribute's info.

Figure 5.4 illustrates the secrets of an extensive relation element.

[6] An intensive relation is consistent if the three conditions are satisfied.

- No duplicated tuple index existing and no unlinked tuple index.
- If there exists a domain, the range must exist to match the domain.
- The range of the function within the range of Idp.

A	null	null	B	C	null	null

0 3

For example, if we have a binary relation R, the first attribute of Ris (name = String, Type = String, Size = 3, Tuple Index = "1") the second attribute of R is (name = String, Type = String, Size = 4, Tuple Index = "2"). The offset for the first attribute is 0 and the offset for the second attribute is 3.

Data Structure of an Extensive Relation Element

Figure 5.4: Data Structure of an Extensive Relation Element

Table 5.14: Access Program Interface for ExRelation

Access Program	Return Type	Description
ExRelation()	ExRelation	Create an empty extension relation.
ExRelation(const ExRelation &exr)	ExRelation	Create a new extension relation from existing extension relation r. Copy the extension relation.
~ExRelation()		Free allocated resource.
operator = (const ExRelation &exr)	ExRelation	Assign this extension relation from extension relation r.
Insert(char * info)	bool	Insert an element info into this extension relation. Return true if inserted successfully[7], otherwise return false.
Delete(char * info)	bool	Delete the element info from this extension relation. Return true if deleted successfully[8], otherwise return false.
getInfo()	char*	Return the information associated with the element.
first(int i=1)	bool	Reset the set to indicate that no element has been selected. If the relation is empty, return false.
next(int i=1)	bool	To indicate there are i elements have been selected. If there are less than i elements in the set, return false.

Relation Module (class Relation)

[7] Before inserting an element in a relation, we will check the element is not duplicated.

[8] Before deleting an element, we will make sure the element is a member of the relation set.

Table 5.15: Access Program Interface for class Relation

Access Program	Return Type	Description
Relation()	Relation	Create an empty relation.
Relation(const Relation &r)	Relation	Create a new relation from existing relation r.
~Relation()		Free allocated resource.
AssignStructure(RelaDS *rlds)	void	Assign the relation structure rlds to this relation.
operator = (const Relation &r)	Relation	Assign this relation from relation r.
Insert(char * info)	bool	Insert an element info into this relation. Return true if insert successfully, otherwise return false.
Delete(char * info)	bool	Delete the element info from this relation. Return true if delete successfully, otherwise return false.
AssignInRelation(InRelation<T>& inrl)	void	Assign the intensive relation inrl to this relation.
GetInRelation()	InRelation <T>	Return the intensive relation from this relation.
CheckConsistence()	bool	Return true if the relation is consistent[9]. Otherwise, return false.
getOffset(Name nm)	int	Return tuple named nm's offset[10]. The offset of the tuple is start position of the attribute in the tuple.
getType(Name nm)	Type	Return attribute nm's data type.
		continued on next page

[9]Defined on page 53.
[10]Defined in Figure 5.4

continued from previous page

Access Program	Return Type	Description
getSize(Name nm)	Size	Return attribute nm's size.
getTotalSize()	int	Return the total size of the tuple (Sum of all attribute size in the tuple).
getInfo()	char*	Return the information of the tuple.
getInfo(Name nm)	char*	Return the information of attribute named nm.
update(char* info)	void	Replace this tuple information with info.
update(Name nm, char* info)	void	Replace attribute named nm's content with info.
first(int i=1)	bool	Reset the relation set to indicate that no element has been selected. If the relation is empty, return false.
next(int i=1)	bool	Set the relation set to indicate i elements have been selected. If there is less than i elements in the relation set, return false.

Data Table Module (class DTable)

Table 5.16: Access Program Interface for class DTable

Access Program	Return Type	Description
DTable()	DTable	Create an empty data table.
DTable(const DTable &dt)	DTable	Create a new data table from existing data table dt to the current one.

continued on next page

continued from previous page

Access Program	Return Type	Description
~DTable()		Free allocated resources.
operator = (const DTable &dt)	DTable	Assign this data table from data table dt.
AssignStructure(List<Field> lst)	void	Assign the data table's structure.
Insert(char * info)	void	Insert an element info into this data table.
Delete(char * info)	bool	Delete the element info from this data table. Return true if the info exists, otherwise return false.
getOffset(Name nm)	int	Return tuple named nm's offset. The offset of the tuple is start position of the attribute in the tuple.
getType(Name nm)	Type	Return attribute nm's data type.
getSize(Name nm)	Size	Return attribute nm's size.
getTotalSize()	int	Return the total size of the tuple (Sum of all attribute size in the tuple).
getInfo()	char*	Return the information of the tuple.
getInfo(Name nm)	char*	Return the information of attribute named nm.
update(char* info)	void	Replace this tuple information with info.
update(Name nm, char* info)	void	Replace attribute named nm's content with info.

continued on next page

continued from previous page

Access Program	Return Type	Description
first(int i=1)	bool	Reset the pointer at the ith element in the data table. The default position is the first one. If the data table is empty, return false.
next(int i=1)	bool	Advanced the pointer to hte at next element in the data table. If cannot advance, return false.

5.4.9 Operator Module (Operator.c)

The Operator Module hides the algorithms to implement the built-in functions we defined in Chapter 4. Each of the access programs has mapped a built-in function.

Table 5.17: Access Program Interface for class Operator

Access Program	Return Type	Description
Domain(Set *dst, Relation &scr)	Set*	Compute domain set of relation scr and store all results in the set dst.
Range(Set *dst, Relation &scr)	Set*	Compute range set of relation scr and store all results in set dst.
Join(Relation *dst, Relation &r1, Relation &r2)	Relation	Compute join relation of relation r1 and relation r2. Result is stored in dst.
Compose(Relation *dst, Relation &r1, Relation &r2)	Relation	Compute composition relation of relation r1 and relation r2. Result is stored in dst.

continued on next page

continued from previous page

Access Program	Return Type	Description
Rearrange(Relation &dst, TupIdx ti1, Relation &rsc, TupIdx ti2)	bool	Rearrange a tuple from relation rsc's position ti2 to dst's position ti1.
Image(Set *dst, Relation &rsc, Set &s)	Set	Compute the image[11] of relation rsc restricted by integer set s, save result in the set dst.
PreImage(Set *dst, Relation &rsc, Set &s)	Set	Compute the preimage[12] of relation rsc restricted by set s, save result in the set dst.
Product(Relation *result, Set &s1, Set &s2)	Relation	Compute the product relation of set s1 and set s2, save result in relation result.
Identity(Relation *result, Set &s)	ExRelation	Compute the identity relation of integer set s, save result in relation result.
RangeMerge(Relation *result, Relation &rsc, TupIdx tidx, Op op)	ExRelation*	Computer RangeMerge relation on rsc restrict by tuple index tidx, operator op, save result in relation result[13].
RangeDivide(Relation *result, Relation &rsc)	Relation*	Computer RangeDivide relation on rsc, save result in relation result[14].

continued on next page

[11] See chapter 4 for the definition.
[12] See chapter4 for the definition.
[13] See chapter 4 for the definition of RangeMerge
[14] Also see chapter 4 for definition.

continued from previous page

Access Program	Return Type	Description
ArithmeticComp(Relation *result, Relation rsc, TupIdx tidx, Op op, int x)	Relation*	Compute ArithmeticComp relation restricted by relation rsc, tuple index tidx, operator op and integer x, save result in relation result.
Restriction(Set *result, Set &rsc, char *(*Info)(), bool (*cond)())	Set*	Compute restricted set from set rsc according to condtion function cond. Function Info will provide condtion function the data that should be check. Results are stored in relation result.
OperatorOnFunction(char op, Relation & r)	int	Compute operator op on relation r. Return integer value.

Part 2 Compile Time Module

The previous modules belong to the DNL API part. The Place Holder Module is designed for the DNL compiler. The functions in this module are called by the Yacc, and will eventually generate C++ source codes. We call the function in the API modules the run time functions, and the functions in the DNL Compiler module the compile time functions.

5.4.10 Place Holder (Holder.cc)

The Place Holder module hides the data structures representing a holder to hold a function. The functionality of this module is like a stack that can hold all information

about a function. It pushes the function name, parameters, and return type when it reads the inputs. It pops all the information about this function when it recognizes the function.

Table 5.18: Access Program Interface for class Holder

Access Program	Return Type	Description
Init()	void	Initialize the place holder module.
CreateVar()	char*	Create a new variable name, and return it as a string.
Insert_Id(Id_Tab* id)	void	Insert the variable id into idenfier table.
LookUp(char* id)	bool	Look up the identifier id from the identifier table. If id already existed, return true, otherwise return false.
GetType(char* id)	IdType	Return identifier id's data type.
Push_Id(Id_Info *id)	void	Push identifier id's information into place holder stack.
Pop_Id()	Id_Info*	Pop out top identifier object, return all its information.
Top_Id()	Id_Info*	Return the top identifier object's information. Do not pop it.
Push_Op(Op_Info* op)	void	Push operator op into operator stack.
Pop_Op()	Op_Info*	Pop out a operator op object.
Top_Op()	Op_Info*	Return the top operator's information.
Done()	void	When all jobs are finished, close all used files.

5.4.11 Parser Module(p.y and main.c)

The Parser Module hides the Data Network Language grammar specification, and the actions to be invoked when a grammar rule is recognized. This module does not have access functions used by any outside module. The program in this module will be called by main.c. The functionality of main.c has been stated in [16] and [2].

5.5 Algorithms for Compiling DNL

Compiling DNL text file is the work of the first part of DNL Interpreter System. The compiler design is based on [1] and [14]. In this section, we provide an overview of the algorithm of this part. The input of DNL Interpreter System is a DNL text file. The output of the compiler part of DNL Interpreter is C++ source code including a header file, a function file, and a variable declaration file. The compiler part of DNL Interpreter uses the following algorithms.

1. Check the keywords and variables by Lex checker (details see [16]).

2. Check the grammar rules by Yacc and grammar specifications.

3. Push variables in variable stack and decide variable type. Compare the variable type with the variables in the variable register table. If the variable is new, record it in the variable register table.

4. Once a function is recognized,

   - Create a new variable to save the result. Decide its type and record it in the variable table.
   - Pop up all the arguments from variable stack.
   - Decide whether there is a need to create associate functions. If yes, create associate functions and output them in the buffer.
   - Output the main C++ function statement in the main output buffer.

5. Repeat for all DNL statements. If finished,

   - Write all related include files in a header file.

- Output all variable declaration in a header file.
- Output associate functions from the function buffer to the function file.
- Output main C++ statements from the main buffer to the main file.
- Close all files and release resources.

Chapter 6

Illustrative Examples

In this chapter we use two examples to illustrate the use of Data Network Language Interpreter System. We will show how to create DNL relations and how to do some simple queries by using these relations.

6.1 Town-Hotel Example

Going back to the example in Chapter 1, a Dog-lover's club wants to have a meeting in New York City. The coordinator of the club wants to determine whether hotels in New York have enough available rooms for all members and allow them to bring dogs.

To illustrate this example, we build three relations as follows,

Relation $R_1 = \{(Town, Hotel) \mid Town \in Town_Set \wedge Hotel \in Hotel_Set\}$

Relation $R_2 = \{(DogAllowed, Hotel) \mid DogAllowed \in Boolean \wedge Hotel \in Hotel_Set\}$

Relation $R_3 = \{(Hotel, (Day, AvailableRooms)) \mid Hotel \in Hotel_Set \wedge Day \in Day_Set \wedge AvailableRooms \in \mathbb{N}\}$

We look for a new relation $R = \{((Town, Day), NumberOfDogeeRooms) \mid$

$Town \in Town_Set \wedge Day \in Day_Set \wedge NumberOfDogeeRooms \in \mathbb{N}\}$
Steps:

1) Create all the three relations $R_1, R_2, and R_3$ as above.

```
Create(R1, (1, Town, char, 10), (1, Hotel, char, 25));
Create(R2, (1, DogAllowed, int, 1),(1.2, Hotel, char, 25));
Create(R3, (1, Hotel, char, 25), (2.1, Date, char, 8), (2.2, Room, int, 4));
```

We can get the three relations.
Relation $R_1 = \{(Town, Hotel) \mid Town \in Town_Set \wedge Hotel \in Hotel_Set\}$
Relation $R_2 = \{(DogAllowed, Hotel) \mid DogAllowed \in Boolean \wedge Hotel \in Hotel_Set\}$
Relation $R_3 = \{(Hotel, (Day, AvailableRooms)) \mid Hotel \in Hotel_Set \wedge Day \in Day_Set \wedge AvailableRooms \in \mathbb{N}\}$

2) Insert some data into relations for illustration.

```
Insert(R1, ("New York", "New York DaysInn Hotel"));
Insert(R1, ("New York", "New York Sheraton Hotel"));
Insert(R1, ("New York", "New York EconoLodge Hotel"));
Insert(R1, ("Toronto", "Toronto Sheraton Hotel"));
Insert(R1, ("Toronto", "Toronto HolidayInn Hotel"));
Insert(R2, (true, "New York EconoLodge Hotel"));
Insert(R2, (true, "New York DaysInn Hotel"));
Insert(R2, (true, "New York Sheraton Hotel"));
Insert(R2, (true, "Toronto Sheraton Hotel"));
Insert(R2, (false, "Toronto HolidayInn Hotel"));
Insert(R3, ("New York Sheraton Hotel", ("07/04/00", 50)));
Insert(R3, ("New York DaysInn Hotel", ("07/04/00", 100)));
Insert(R3, ("New York EconoLodge Hotel", ("07/04/00", 80)));
Insert(R3, ("Toronto Sheraton Hotel", ("07/04/00", 110)));
Insert(R3, ("Toronto HolidayInn Hotel", ("07/04/00", 120)));
```

Therefore,
R_1 ={ (New York, New York DaysInn Hotel),
(New York, New York Sheraton Hotel),
(New York, New York Econolodge Hotel),
(Toronto, Toronto Sheraton Hotel),
(Toronto, Toronto HolidayInn Hotel) }
R_2 ={ (true, New York EconoLodge Hotel),
(true, New York DaysInn Hotel),
(true, New York Sheraton Hotel),
(true, Toronto Sheration Hotel),
(false, Toronto HolidayInn Hotel) }
R_3 ={ (New York Sheraton Hotel, (07/04/00, 50)),
(New York DaysInn Hotel, (07/04/00, 100)),
(New York EconoLodge Hotel, (07/04/00, 80)),
(Toronto HolidayInn Hotel, (07/04/00, 120)),
(Toronto Sheraton Hotel, (07/04/00, 110))}

3) Compute the target relation. The following steps are performed.

```
R4 < − Restriction(R2, GetAttribute(R2, 1) = true);
R5 < − Join(R4, R3);
R6 < − Rearrange(R5, ((2.1, (2.2.1, 2.2.2)), 1));
R7 < − Domain(R6);
R8 < − Compose(R1, R7);
R9 < − Rearrange(R8, ((1, 2.1), 2.2));
R < − RangeMerge(R9, 2, Sum);
```

From these steps, we will get
R_4 ={ (true, New York EconoLodge Hotel), (true, New York DaysInn Hotel),
(true, New York Sheraton Hotel), (true, Toronto Sheraton Hotel) }
R_5 ={ (true, (New York EconoLodge Hotel, (07/04/00, 80))),
(true, (New York DaysInn Hotel, (07/04/00, 100))),
(true, (New York Sheraton Hotel, (07/04/00, 50)))

(true, (Toronto Sheraton Hotel, (07/04/00, 110))) }
R_6 ={ ((New York EconoLodge Hotel, (07/04/00, 80)), true),
((New York DaysInn Hotel, (07/04/00, 100)), true),
((New York Sheraton Hotel, (07/04/00, 50)), true),
((Toronto Sheraton Hotel, (07/04/00, 110)), true) }
R_7 ={ (New York EconoLodge Hotel, (07/04/00, 80)), (New York DaysInn Hotel, (07/04/00, 100)),
(New York Sheraton Hotel, (07/04/00, 50)), (Toronto Sheraton Hotel, (07/04/00, 110)) }
R_8 ={ (New York, (07/04/00, 80)), (New York, (07/04/00, 100)),
(New York, (07/04/00, 50)), (Toronto, (07/04/00, 110)) }
R_9 ={ ((New York, 07/04/00), 80), ((New York, 07/04/00), 100),
((New York, 07/04/00), 50), ((Toronto, 07/04/00), 110) }
R ={ ((New York, 07/04/00), 230), ((Toronto, 07/04/00), 110) }

We can see from the result. On July 4, 2000, the dog lover's club can get 230 available hotel rooms in New York City and 110 rooms in Toronto.

4) Compose a DNL query file. From now, we already see how to write a simple DNL query. To compose a DNL query file, simply add step 1), 2), and 3) altogether into a text file, "Town-hotel.dnl".

5) Parse the DNL file by entering the command line:

   Parser Town-hotel.dnl

   The parser will give the following message, if grammar of the file is correct.

   *****DNL parser has recognized this program!****

   Otherwise, the parser will give error information and the user needs to correct it and parse again.

After the file is parsed, the DNL file is translated into three C++ files, a main program file, a variable declaration file, and a function file.

6) Make the C++ file and run it to get the final results. Simply enter in the command line:

```
make
```

An executable file named DNL will generated. If we type the file name in the command line, the final result R will be generated.

6.2 Shopper's Guide Example

Suppose many supermarkets have their price list on-line to attract customers. This example shows how to use DNL to build an application to help customers decide which supermarket can provide them the lowest price.
Suppose we have a relation,

$R_1 = \{(Store, (Item, (Price, Quantity))) \mid Store \in Store_Set \wedge Item \in Item_Set \wedge (Price, Quantity) \in (R^+ \times \mathbb{N})\}$
and given any shopping list,

$$S = \{(Item, Quantity) \mid Item \in Item_Set \wedge Quantity \in \mathbb{N}\}$$

We would like to have the relation,

$$R = \{(Store, Cost) \mid Store \in Store_Set \wedge Cost \in R^+\}.$$

1) Create relation R_1 and R, and insert some data.

```
Create(R1, (1, Shop, char, 15), (2.1, Item, char, 10),
      (2.2.1, Price, float, 6), (2.2.2, Quantity, int, 5));
Create(R, (1, Shop, char, 15), (2, Cost, float, 6));
Insert(R1, ("Nofrill Toronto", ("Fuji Apple", (1.99, 2500))));
Insert(R1, ("Nofrill Toronto", ("RoyalApple", (1.59, 4000))));
Insert(R1, ("Fortino Toronto", ("Banana", (0.49, 8000))));
... ...
```

2) Search for the supermarkets which can provide the required item and quantity. Suppose we want to buy 3 pounds of Royal apple and 2 pounds of banana.

```
R2 < − Restriction(R1, GetAttribute(R1, 2.1)= "RoyalApple"
      && GetAttribute(R1, 2.2.2)>3);
R3 < − Restriction(R1, GetAttribute(R1, 2.1)= "Banana"
     && GetAttribute(R1, 2.2.2)>2);
```

Search for supermarkets which have both Royal apple and banana.

```
R4 < − Restriction(R2, GetAttribute(R2, 1) member Domain(R3));
R5 < − Restriction(R3, GetAttribute(R3, 1) member Domain(R2));
```

3) Compute results and insert them into the new relation R.

```
R6 < − Restriction(R4, GetAttribute(R4, 1) = "Nofrill Toronto");
R7 < − Restriction(R5, GetAttribute(R5, 1) = "Nofrill Toronto");
I1 < − ArithmeticComp(R6, 2.2.1, ∗, 3);
I2 < − ArithmeticComp(R7, 2.2.1, ∗, 2);
Insert(R, ("Nofrill Toronto", I1+I2));
... ...
```

4) Compose the DNL query file from step 1, 2, and 3. Save the file in a text file, for example, named "ShopperGuide.dnl".

5) Parse the “ShopperGuide.dnl” by entering

```
Parser ShopperGuide.dnl
```

When the parser give the following information, it means the “ShopperGuide.dnl” passed grammer and generated the C++ source file.

```
*****DNL parser has recognized this program!****
```

6) Make the C++ file and run it to get the result.

```
make
```

A executable file named DNL will generated. Type the file name in command line, the final result R will be generated.

Chapter 7

Results and Conclusions

This chapter summarizes the results, discusses limitations and future work, and draws the conclusions of this thesis.

7.1 Results

A Data Network Language Interpreter System has been developed. This thesis presents the design of the Data Network Language Interpreter System and the use of interpreter system by examples. The Interpreter System interprets the Data Network Language based on the the definition in Chapter 4. This interpreter is the kernel of Data Network Language. It translates Data Network Language into executable files and provides the final results. From the examples we have shown that the system achieves our objectives. It provides a base for further research on the new language. The Interpreter System also provides interfaces to use the existing tool TTS. Expressions such as inductively defined predicates can be used to represent sets.

In the design and implementation of the Interpreter System we focus on Software Engineering methodologies. Each of the modules in the Interpreter System hides the secrets (data structure and algorithms) of that module. Modules communicate by defined interface and access functions. The Interpreter System was written in C++ and is executable on a Unix platform. The structure will facilitate future changes.

7.2 Limitations

The purpose of this research was to investigate the basic idea behind DNL, not to produce a production tool. Consequently, this work has the following limitations.

7.2.1 Performance Optimization

Part of this Interpreter System is a compiler. To get the better performance, commercial compilers perform code optimization. Since this is not a compiler project, we kept the compiler part as simple as possible. This interpreter system, therefore, does not include code optimization for the target C++ codes. However, based on the example codes we inspected, the C++ codes generated by this interpreter system is good enough for our present purpose.

7.2.2 Data Type in Data Network Language

We only use the integer and string as the default data types in this interpreter system. This is because the data type has not yet been defined in Data Network Language and type checking is not part of this interpreter system. The current system can not distinguish integer number and real number. Arithmetic operations involving both integer and real values are possible with this interpreter. In this case, the system will convert real number to be the next lowest integer. For other datatypes, arithmetical operation will not be allowed if they can not be converted to integer.

The reason that type checking does not include in this tool is because type checking itself is a complex work in the compiler. As the same reason we mentioned above that we did not focus on compiler part. Without type checking it has greatly deducted the complexity of the compiler design.

7.3 Future Work

The following future work is based on the the limitations above.

1. Add data type definition in Data Network Language and type checking in semantic analysis.

2. A graphical user interface and other tools could be added to let DNL be used more easily.

3. Add performance optimization in the interpreter system to get better performance.

7.4 Conclusions

From the use of DNL Interpreter System, we can see that binary relations can be applied on general structured data querying. Using binary relations can improve the consistency and accuracy on data querying. It also appears realistic to apply binary relation operations across networks.

DNL will also contribute to the Internet technologies. If we set up a class of public binary relational data, it will be relatively easy to query information by using DNL. From the properties of DNL we can see that not only markup languages like HTML and XML, but also query languages, can be used in computer networks. Based on DNL, we can see that formal mathematics, particularly relational algebra, can provide a firm foundation for network software applications.

Appendix A

Data Network Language Makefile Reference

This appendix shows the design of the two default makefiles for Data Network Language Interpreter. A system like DNL Interpreter System, that uses many program modules, has to cope with an assortment of individual files. There are some generation procedures needed to turn the assortment of individual files into the final executable file.

Make provides a method for maintaining file dependencies and the exact sequence of operations needed to generate a new version of the program[2]. *Make* keeps a description file which keeps the track of the commands that create files and the relationship between files. It also trace the impacts that have made on other files if some of the files were modified. In short, Makefile helps to keep an up-to-date version of programs[1][2]

A.1 Parser Makefile

This section gives a brief description of the procedure to generate the DNL compiler. Since the functionality of this compiler was already stated in Chapter 5 we will not

[1] For the details of makefile, user can refer to the reference

repeat it. We explain the compiler generation procedure and file dependencies as follows.

Table A.1: File Dependencies and Compiler Generation Procedure

Statements	Description
LEX = flex YACC = yacc -dv CC = g++ hline parser: y.tab.o lex.yy.o Holder.o main.o $(CC) -o parser y.tab.o Holder.o main.o lex.yy.o -ly -ll	Lex Analyzer command Yacc command C++ compiler command Generate object file *parser* by linking all the object files generated below with the Lex and Yacc.
main.o: main.c y.tab.h Holder.h $(CC) -c main.c	Generate object file for parser module.
Holder.o: Holder.h Holder.cc $(CC) -c Holder.cc	Generate object file for Place Holder module.
lex.yy.o: lex.yy.c y.tab.h Holder.h $(CC) -c lex.yy.c	Generate object file for Lex Analyzer.
y.tab.o: y.tab.c y.tab.h Holder.h $(CC) -c y.tab.c	Generate object file for Yacc specification.
y.tab.c: p.y Holder.h $(YACC) p.y	Generate a parser from specification[2].
lex.yy.c: lexer.l $(LEX) lexer.l	Parse the lex specification and produce the lexical analyzer lex.yy.c.

[2] See [2] for details.

A.2 Makefile for Linking with API

This section gives the file dependencies and generation procedure of DNL API.

Table A.2: File Dependencies and Compiler Generation Procedure

Statements	Description
CC = g++	define C++ compiler
DNL: DN_Info.o idp.o ExSet.o Set.o DSFile.o ExRelation.o Operator.o scr.o $(CC) DN_Info.o Idp.o Set.o DSFile.o ExRelation.o Operator.o scr.o -o DNL	Generate the executable file DNL
scr.o: Operator.h scr.c $(CC) -c scr.c	Generate the object file scr.o from the scr.c.
scr.c : decl.txt main.txt func.txt cat decl.txt defu.txt main.txt func.txt > tmp.c	Generate the C++ source file from the three result files of part 1.
Operator.o: Operator.h Operator.c $(CC) -c Operator.c	Generate the object file for the advanced operator module.
ExRelation.o: ExRelation.h ExRelation.c $(CC) -c ExRelation.c	Generate the object file for the relation module.
Set.o: Set.h ExSet.h Set.c $(CC) -c Set.c	Generate the object file for the set module.
Idp.o: Idp.h Idp.cc $(CC) -c Idp.cc	Generate the object file for the intensive set module.
DSFile.o: DSFile.cc DSFile.h $(CC) -c DSFile.cc	Generate the object file for the record module.
DN_Info.o: DN_Info.h DN_Info.cc $(CC) -c DN_Info.cc	Generate the object file for the information module.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffery D. Ullman. *Compilers - Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, December 1985.

[2] AT&T. *UNIX System V Programmer's Guide.* Prentice Hall, 1989.

[3] J. Bradley. *Combinatorics & Discrete Probability.* Addison-Wesley Pub. Ltd., 1988.

[4] Soumen Chakrabarti, Byron E. Dom, S. Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, Andrew Tomkins, and David Gibson andJon Kleinberg. Mining the web's link structure. *Computer*, 32(8):60 – 67, August 1999.

[5] D. Gries and F. Schneider. *A Logical Approach to Discrete Math.* Spring-Verlag New York Inc., 1993.

[6] Software Engineering Research Group. Table tool system developer's guide. Technical report, CRL Report 229, January 1997.

[7] Paul R. Halmos. *Naive Set Theory.* D. Van Nostrand Company, Inc., 24 West 40 Street,New York, 1960.

[8] D. C. Ince. *An Introduction to Discrete Mathematics and Formal System Specification.* Clarendon Press, Oxford, 1988.

[9] Ryszard Janicki, David Lorge Parnas, and Jeffery Zucker. Tabular representations in relational documents. In C. Brink, W. Kahl, and G. Schmidt, editors, *Relational Methods in Computer Science*, Advances in Computing Science, chapter 12, pages 184–196. Springer Wien NewYork, 1997.

[10] P. Li. Networking design of data network language. Master's thesis, McMaster University, 2000.

[11] D. L. Parnas. Tabular representation of relations. Technical report, CRL Report 260, October 1992.

[12] D. L. Parnas. Predicate logical for software engineering. *IEEE Transactions on Software Engineering*, 19(9):856–862, September 1993.

[13] D. K. Peters. Generating a test oracle from program documentation. Crl report, McMaster University, April 1995.

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

[15] W. M. Thski and T. S. E. Maibaum. *The Specification of Computer Programs.* Addison-Wesley Publishing Company, 1987.

[16] X. Yan. Data network language design. Master's thesis, McMaster University, September 2000.