User’s Guide to

Froglingo, An Alternative to DBMS, Programming Language, Web Server, and File System

Release 1.0, January 14th, 2010

Kevin Xu, Jingsong Zhang
Bigravity Business Software LLC
2306 Johnson Circle
Bridgewater, New Jersey 08807

Table of Content

Chapter 1: INTRODUCTION ... 4
1.1 What is Froglingo? ... 4
1.2 Who should use Froglingo? ... 5
1.3 Why is Froglingo? ... 5
1.4 Prerequisites ... 5
1.5 System Setup ... 6
1.6 Sample Tasks ... 6
1.6.1 A DBMS and a programming language ... 7
1.6.2 An Information Community ... 8
1.6.3 Access over Internet ... 8
1.7 Where Froglingo is ... 9
1.7.1 Language ... 9
1.7.2 Programming Language ... 10
1.7.3 Data Model ... 11
1.7.4 Hybrid ... 11
1.7.5 Unification ... 12
1.8 Document Organization ... 12
Chapter 2: DATA CONSTRUCTION ... 14
2.1 Constant ... 14
2.2 Identifier ... 14
2.3 Term ... 15
2.4 Assignment and Database ... 16
2.5 Dependent Relationships ... 17
2.5.1 Functional Dependency ... 18
2.5.2 Argumentative Dependency ... 18
2.5.3 Recursively Functional Dependency ... 18
2.5.4 Recursively Argumentative Dependency ... 19
2.6 Database Update ... 20
2.6.1 Create Operation ... 21
2.6.2 Assignment Update ... 21
2.6.3 Delete Operation ... 21

2.6.4 Record Operation ... 22
Chapter 3: DATA QUERY ... 23
3.1 Normal Form and Evaluation ... 23
3.2 Arithmetical and Boolean Operators ... 24
3.2.1 Arithmetical ... 24
3.2.2 Boolean on Numbers ... 25
3.2.3 Complex Boolean ... 25
3.3 Set-Oriented Query ... 25
3.3.1 Variable ... 25
3.3.2 Select Operator ... 25
3.3.3 Index Attribute ... 27
3.3.4 Sort Clause ... 27
3.3.5 Summary Clause ... 27
3.4 Derivative Relationships ... 28
3.4.2 Functional Derivative ... 29
3.4.3 Argumentative Derivative ... 30
3.4.4 Recursively Functional Derivative ... 30
3.4.5 Recursively Argumentative Derivative ... 31
3.4.6 Properties of Derivative Relationships ... 31
Chapter 4: BUSINESS LOGIC ... 33
4.1 Infinite Data ... 33
4.1.1 Variable and Database ... 33
4.1.2 Variable Range ... 34
4.1.4 Query on Infinite Data ... 35
4.1.3 Function Recursion ... 36
4.1.5 Update Operation in Assignment ... 37
4.1.6 Variable and Database Update ... 37
4.1.7 Evaluation Rules for Variables ... 38
4.2 Sequential Terms ... 39
Chapter 5: INFORMATION COMMUNITY ... 41
5.1 User Account ... 41
5.2 Session ... 42
5.2.1 Establishment ... 42
5.2.2 Signature ... 43
5.2.3 Stand ... 43
5.3 Traveling and Naming ... 44
5.3.1 Moving Stand ... 44
5.3.2 Current and Upper Stand ... 45
5.3.3 User Home ... 45
5.3.4 Application Home ... 45
5.3.5 Non-Application Home ... 46
5.3.6 Absolute Name ... 46
5.4 Privilege ... 47
5.4.1 Administration ... 48
5.4.2 Access ... 48
5.4.3 Privilege Removal ... 49

5.4.4 Interface 50
5.4.5 Partnership 50
Chapter 6: FILE MANAGEMENT 53
6.1 Upload 54
6.2 Download 55
6.3 OS Path 56
Chapter 7: ACCESS OVER INTERNET 58
Disable favicon support in Firefox 58
7.1 Web Server Setup 58
7.2 URI 59
7.3 HTML/XML Files 61
7.3.1 Document 61
7.3.2 Tag <frog> 62
7.3.3 Attribute Proceeded with “frog” 66
7.3.4 Attribute “frog:if” 66
7.3.5 Attribute “frog:while” 67
7.3.6 File Argument 69
7.4 Request via HTML Form 70
7.4.1 Extended URIs 70
7.4.2 HTML Form 71
7.4.3 Upload via Web Browser 72
Chapter 8: End User, Developer, and Administrator 73
8.1 Simple Data Type and Membership Operator 74
8.2 Void for Nothing 75
8.3 Null for Undefined 75
8.4 Error for Failure 76
8.5 Day and Time 76
8.6 Database File 77
8.6 Log Files 78
Appendix A: Release Notes 78
Appendix B: Grammar 78
Acknowledgement 80
Reference 81

Chapter 1: INTRODUCTION

Once upon a time, the only resident of the world was a turtle. Every day, this turtle swam across a river and walked to a cropland for food. One day, however, a fish appeared—the world's second resident—and said to turtle, "Can you bring me food if I give you a ride across the river?"

"Sure!" said the turtle.

Years later, a bird appeared—the world's third resident—and said to turtle and fish, "I will fly to the cropland for food."

The turtle and fish were puzzled. "How?" they asked.

Many database applications were written in programming language in 1960s and 1970s and they are currently still in operation. Database management system (DBMS) came to the field of database application software around 1970s. It significantly improved the productivity in the development and maintenance of database applications. Due to its limited expressive power, however, a DBMS has to be employed together with a programming language for a database application.

A typical database application system in a corporate environment currently requires DBMSs (such as Oracle and MySQL), programming languages (such as C, Java, and C#), and middleware components including web servers (such as Websphere and WebLogic), and data exchange tools (such as Hibernate and LINQ). In addition, an application-based data access control mechanism has to be constructed.

This document is to introduce Froglingo, a new language and database management system aimed to have both the expressive power of programming language and the productivity of DBMS.

In this Chapter, we briefly introduce Froglingo from Section 1.1 to Section 1.6. For the readers who are interested in the related work of Froglingo in computer science, we recall the history of programming language and database management in Section 1.7. We point out that the easiness in programming language and database management is measurable mathematically; and conclud that a data model, e.g., EP data model of Froglingo, reaches the limit in easiness when it is semantically equivalent to a class of total recursive functions.

1.1 What is Froglingo?

Froglingo is a system consolidating the multi-component system architecture of the traditional technologies into a single component. It is a unified solution for information management, and an alternative to having to combine a programming language, DBMS, a file system, and a web server. It is a "database management system" (DBMS) that stores and queries business data; a "programming language" that supports business logic; a "file system" that stores and shares files; and a "web server" that interacts with users across networks.

It does more than combine existing technologies; it is a single language that uniformly expresses both data and application logic. It is a system supporting integrated

applications without using application-based data exchange components and data access control mechanism.

Froglingo system is a computer system that implements the Froglingo. It has a database uniformly managing data and application logic. Therefore, we call a Froglingo system a database application management system (DAMS).

1.2 Who should use Froglingo?

Froglingo is a generic tool for database applications. It can be used to construct any kinds of information management systems involving data, files, and business logic. One can Froglingo for the following systems:

- Database application,
- Content management system,
- Data warehouse,
- Web sites,
- Collaborative computing environment across multiple organizations.

1.3 Why is Froglingo?

The following facts make database application development easier:

- Froglingo is a single language to manipulate files, business data, and business logic.
- Database applications can communicate without data exchange agent.
- Froglingo offers more powerful query expressions than SQL does.
- Froglingo has a single storage uniformly storing files, data, and business logic.
- User accounts and access privileges, as built-in facilities, can be specified by users to perform data access controls between business units, and (or) between users in a business unit.
- Froglingo has its own built-in web server that communicates with web browser across network.

More significantly, using Froglingo reduces maintenance cost. A Froglingo system is a single piece of software program on the top of operating system. The administrative work traditionally for database, middle-ware, and application can be consolidated to Froglingo only. More efficiency on software development and maintenance implies more software assets for businesses with limited resources.

The authors in the article “Data Model and Total Recursive Functions” have formally proved that the EP data model, Froglingo without variables, is equivalent to the class of total recursive functions”. This led to the conclusion in the article that Froglingo is the easiest to use in software development and maintenance.

1.4 Prerequisites

Since Froglingo is a new technology, it doesn’t require readers to have knowledge on the

traditional technologies such as programming languages and database management systems. The exceptions are the HTML language and Java Scripts (Ajax) for web page construction.

1.5 System Setup

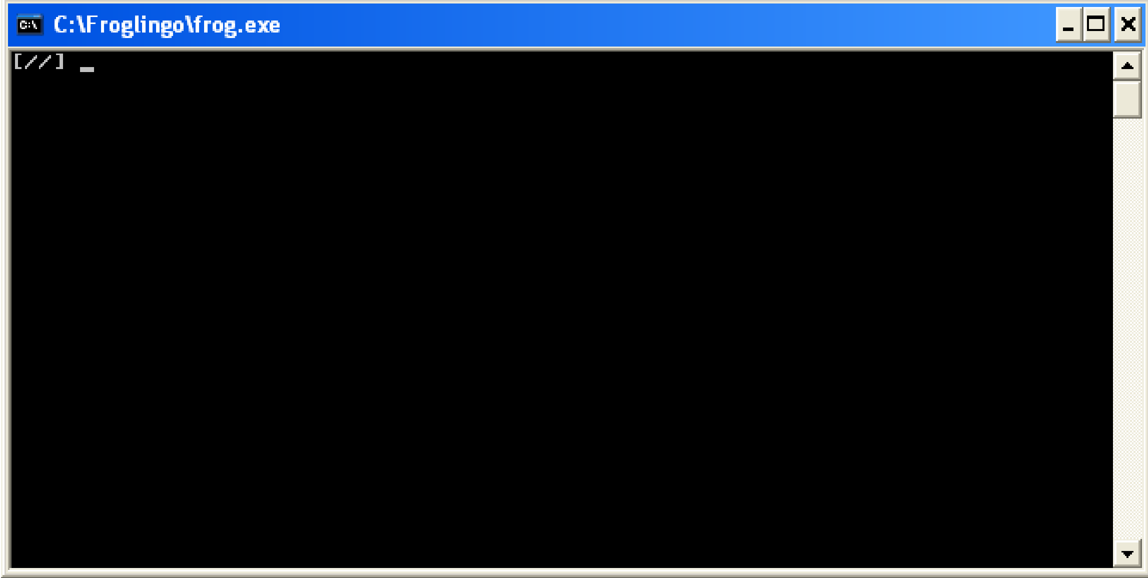
To run Froglingo, a computer needs to be installed with:

- Windows XP system,
- Web browser Internet Explorer 7.0 and above, or FireFox.

Here are the steps of installing a Froglingo system:

- Create a folder in your Windows XP system, say: `C:\Froglingo`.
- Download the application file `frog.exe` to the folder. The file is available on the website: http://www.froglingo.com

Now the system is ready for use. To run the system, you may simply double click the application icon `frog.exe`. A command prompt (CMD) window will appear as the following:



You are now at the root of the system and you are ready to give commands as you desire. Here are examples:

```
[//] "Hello World";
"Hello World";
[//] 3 + 5;
8;
```

1.6 Sample Tasks

This section provides a few sample Froglingo expressions. It gives readers an overview on Froglingo.

1.6.1 A DBMS and a programming language

A string or a number is simply echoed:

```
[//] "Hello World";
"Hello World";
```

Regular arithmetic calculations are supported:

```
[//] 3 + 5;
8;
```

Data is constructed by using a built-in operator `create`:

```
[//] create Mike salary = 1000;
[//] create Dave salary = 2000;
```

Queries on entered data are available immediately:

```
[//] Mike salary;
1000;
[//] Mike salary + Dave salary;
3000;
[//] select $person, $person salary where $person salary >=1000;
Mike, 1000
Dave, 2000;
```

Business logic (or infinite data) is stored as data too:

```
[//] create tax $money = ($money * 0.3);
[//] select $person, $person salary, tax ($person salary) where
$person salary >= 100;
Mike, 1000, 300
Dave, 2000, 600;
```

Another example of business logic - a factorial function:

```
[//] create fac 0 = 1;
[//] create fac $n:[$n > 0] = ($n * (fac ($n - 1)));
[//] fac 4;
24;
```

A set of built-in operators are applicable to Froglingo data. They are more powerful than those existing in the traditional DBMSs such as SQL. Queries against a directed graph is a typical example:

```
[//] create A; /* define a vertex 'A' */
[//] create B; /* define a vertex 'B' */
[//] create C; /* define a vertex 'C' */
[//] create A B = B; /* define a directed connection A -> B */
[//] create B C = C; /* define a directed connection B -> C */
```

Query: Is there a path from vertices A to C?

```
[//] A >=+ C;
true;
```

Froglingo manages files and a HTML file can embed Froglingo expressions. Suppose you have a file `myprofile.html` and the content is:

```
<frog> $name </frog>
<html>
     <body background="photo.jpg">
          My name is
```

```
            <frog> $name </frog>
            and my salary is
            <frog> $name salary </frog>
      </body>
</html>
```

The file is loaded by using the `load` command:

```
[//] load myprofile.html;
```

1.6.2 An Information Community

The data you entered into Froglingo database is not shared with anyone else unless you explicitly granted permission to some one on specific data. The first step is to setup a multi-user environment by giving a password to youself as the root user:

```
[//] passwd;
New Passwd: ******
Confirm Passwd: ******
```

Now you are acting as the most privileged user `root`; and you are ready to create additional user accounts:

```
[//] addusr greg;
The passwd is: un@Ik8l2
[//] addusr www.myclienta.com;
The passwd is: kkjkadsf
[//] quit;
```

The last command `quit` terminates the Froglingo process. Now you may login again with a different user account.

```
C:\Froglingo\frog.exe
User Id: www.myclienta.com
Passwd: ********
Confirm passwd: ********
[//www.myclienta.com]
```

1.6.3 Access over Internet

To demonstrate how Froglingo supports web browsers, let's continue with a few more tasks:

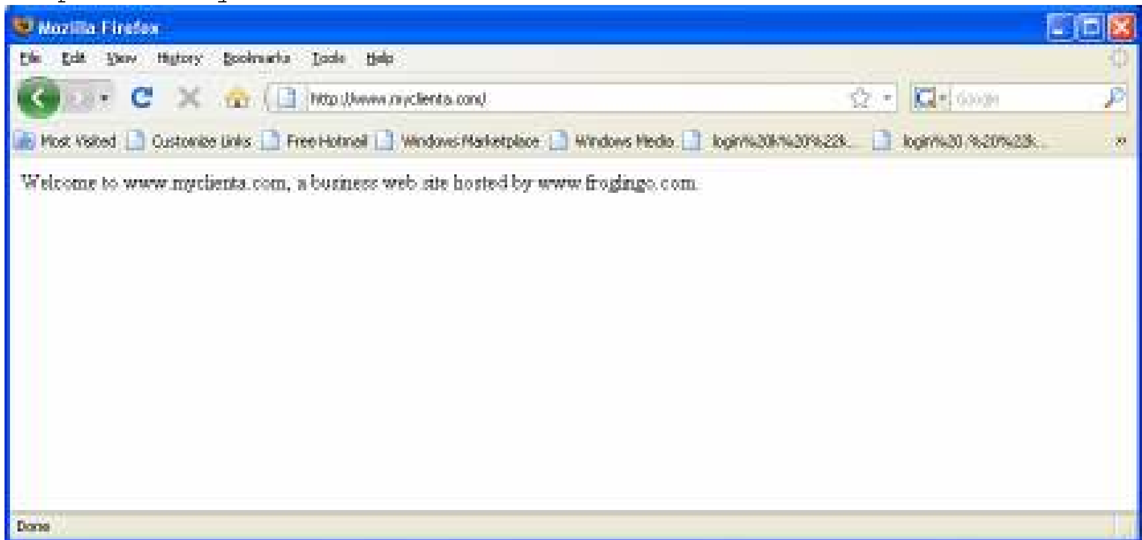
```
[//www.myclienta.com] load photo.jpg;
[//www.myclienta.com] load index.html;
[//www.myclienta.com] print index.html;
index.html = <html>
<body>
      Welcome to www.myclienta.com, a business web site hosted by
www.froglingo.com.
</body>
</html>;
[//www.myclienta.com] grtacc index.html anyone;
[//www.myclienta.com] quit;
```

The command `grtacc` above assigned the built-in user account `anyone` to have access permission on `index.html`. Now any user can view the web page via a web browser across network as soon as a Froglingo web server is started via a CMD window:

```
C:\\Froglingo\frog.exe -p 80
```

Assume that the user account `www.myclienta.com` also has been registered as a domain name via a domain name registration agent, you are ready to use a web browser to interact with the Froglingo web server by entering URIs embedding Froglingo expressions. Here is an example:

```
http://www.myclienta.com/index.html
```



1.7 Where Froglingo is

Expressive power has been well-established as a dimension of measuring the quality of a computer language. Easiness is another dimension. It is the main stream of development in programming language and database management. Assessing the easiness of a language appears to be subjective and there hasn't been a formal method in measuring it. The introduction of Froglingo, however, sheds a light on how to view easiness more objectively. We start with the two assumptions: (1) A data model is easier than a programming language in the development and maintenance of those applications expressible by the data model; (2) If a data model is more expressive than another data model, the former is easier than the latter in the development and maintenance of database applications where a programming language is involved.

Sections 1.7.1 to 1.7.5 recall the history of the progress in the fields programming language and database management, and point out that data model is about total recursive functions while programming language is about partial recursive functions. A system having both a data model and a programming language is called a hybrid, and it becomes a unification and therefore the easiest when the semantics of the data model is equivalent to the class of total recursive function.

1.7.1 Language

A language is a set of expressions used by a set of users. An expression is a sequence of words. Once an expression is presented to a user, the user reacts and produces another expression. Users use a language to communicate with each other. In other words, a user

feeds an expression to another user and waits for a reply.

Talking about a language, people must be the ultimate users while computers and other devices are possible users too. A person uses a language because the expressions of the language carry his/her messages. The messages are actually the person's view of some entities in the world. Therefore, when a person is composing an expression, he/she maps a set of entities in the world to the expression; when a person receives an expression, he/she maps the expression to another set of entities. The whole set of the entities expressible by a language is called the semantics of the language.

The semantics is a way of measuring the quality of a language. We call this measurement the expressive power. The larger the semantics is, i.e., the larger size of entities a language can express, the more expressive power the language has.

The quality of a language is also measurable by using another criterion – easiness. When two languages have the same expressive power, one may prefer to use one rather than another. There is no need at this point to analyze what contribute to easiness, but we raise it because it does factor in the computer language evolution.

Two languages can be comparable in easiness only if they are equivalent in terms of expressive power. It is meaningless to compare two languages in easiness if they are two exclusive sets of semantics. But if the semantics of a first language is a superset of a second one's, the first one can be compared with the second one in easiness when the first one is used to express the semantics of the second one.

1.7.2 Programming Language

A programming language is used by both computers and human beings. Its semantics is a subset of the entities in the world, mathematically measured by a class of computable (or partial recursive) functions, the upper-limit a computer can do.

Programming languages have been evolved from machine/assembly languages to high level languages. The unchanged part of programming languages along with the evolution is their expressive power -- Turing machine, i.e., expressing computable functions.

What has been changed along the evolution? It has to do with the easiness of a programming language. We have been continuously working on producing an easier programming language such that we can be more productive in software development and maintenance.

A key difference of a programming language from a DBMS is its ability of representing computable functions and therefore infinite data (normally called business logic or queries) in finite expressions.

The semantics of programming language inevitably falls into the class of partial recursive functions. However we strive hard to design and to use a programming language such that all the software applications coded in the language fall into a narrower class -- total recursive functions. The class of total recursive functions is all useful in representing software applications with the given limitation of computer.

1.7.3 Data Model

A data model is also a computer language. Its semantics is a subset of the class of total recursive functions. It is the mathematical abstraction of a database management system (DBMS) for database applications.

A data model is defined to have a data structure storing a set of data and to have a set of built-in operators under which the set is closed. By closed, we meant that an operation always terminates and returns members from the set. To emphasize the dominating role of data structure, we redefine a data model to have a data structure storing a set of objects and offering dependencies among the objects. The second definition is aimed to be equivalent to the first one while the dependency attributes the built-in operators.

By dependency, we meant that if one object in the set depends on another, then the second must be in the set as well; and reversely if the second is not in set, neither the first one. For example, an attribute in a relational table depends on its row; a child object depends on its parent object in hierarchy; and the birth to an infant depends on both mother and father. The dependency as a restriction avoids exceptions in programming language.

Dependency is also required to be decidable, i.e., data model is always able to tell if two arbitrary objects in set are dependable or not. This restriction disallows an object depending on itself (i.e., data is organized to have a cyclical loop) in the managed sets, and therefore disqualifying a computer language as a data model if there is a program in the language that doesn't terminate on an input.

With the definition above, we say that queue and stack in programming language are the examples of data models. So are the relational data model and the hierarchical data model with the containment relationship. The traditionally called "network data model" that allow cyclical data and not clearly defining the dependencies among cyclical data is excluded from being a data model in this paper. Obviously, both Datalog and a programming language are not data models.

Practice tells us that a data model is easier than a programming language in expressing a finite set of objects (normally called business data). The dependencies that lead the managed set closed on built-in operators are the supporting factors.

1.7.4 Hybrid

Programming language defines functions by coding algorithms. Data model defines functions by enumerating properties. Although data model is preferable, programming language is inevitable.

First of all, many business data, falling into a class of total recursive functions, is desired but not expressible in a traditional data model. By not expressive, we meant that some dependencies would be lost even if they were placed (decomposed) into the data structure of a data model. Hierarchical data, as a typical example, can be folded into table, but its containment relationships cannot be captured by the relational data model. As another example, dependency among the vertices in a directed graph cannot be captured in both relational data model and hierarchical data model.

Secondly, constructing arbitrary queries (business logic) on the top of managed data set

needs programming language. Although built-in operators can construct a class of useful queries, they don't exhaust all the queries that are practically needed. For example, a query in the relational data mode cannot simply return a single attribute or a sequence of attributes out of a relational database. This has no exception even for a unification as to be discussed in the next section.

A system having both a programming language and a data model is called a hybrid.

The most popular hybrid today is the combination of the relational data model and a programming language. There are two other kinds of hybrids today: XML/XQuery and Object-Oriented approaches, partially originated from the research effort of "database programming language" in 1970s to the early 1990s. The XML/XQuery approach is based on the hierarchical data model. The Object Oriented approach is based on the network structure by adding other data structure including hierarchical.

A hybrid is easier than a stand-alone programming language in database application. A hybrid is easier because a data model is used for a part of database application. A hybrid is easier than another hybrid if the data model of the first hybrid semantically is a super set of the data model of the second one. Comparing the easiness of those hybrids based on relational and hierarchical data models is not meaningful because their semantics are overlapped and not inclusive.

1.7.5 Unification

A hybrid becomes a unification if an arbitrary total recursive function can be expressed in the data structure. In other words, the data model can be used to express all the intended business data while programming language is for business logic or queries only. It mathematically means that a unification could enumerate arbitrary software applications, as long as they are total recursive, without programming language if space was unlimited.

Given the context of easiness provided in this article, we conclude that a unification is the easiest. The articles [2] and [1] demonstrated and proved that Froglingo is such a unification.

The upper bound of Froglingo in easiness is translated to the following benefits in the software development and maintenance:

- There is no need to decompose business data to a traditional database; and to reassemble the business data by using programming languages.
- The built-in operators available in Frolingo are more expressible than those in a traditional data model.
- There is no need to do data exchange between DBMS and programming language and between database applications hosted in one Froglingo system.
- There is no need to construct the logic of access control at application level.

1.8 Document Organization

Chapter 2 and 3 introduce the core concepts. It is for managing business data roughly in correspondence to that in a traditional DBMS. The Chapter 4 introduces variables and sequential terms. It is roughly in the correspondence of programming language for business logic. Nevertheless, both business data and business logic are stored as data in

Froglingo.

Chapter 5 covers the concepts that enable a Froglingo system being an information community. By saying a community, we meant that Froglingo offers an environment in which a user can construct his/her own data in a private space; selectively specify data to be shared with authorized users; and collaborate with other users to construct business partnerships. By users, we mean business organizations, employees in an organization, and individual users. This community is available across network, mainly the Internet via web browsers as discussed in Chapter 7.

As a part of information, file is managed in Froglingo and is discussed in Chapter 6. Chapter 8 discussed miscellaneous issues that system administrators and developers should concern. To better assist readers in understanding Froglingo, Chapter 9 provided a few applications as case studies

In the document, we use the font `Courier New` for those texts appeared as the inputs or outputs of Froglingo system; and the bolded font **`Courier New`** for those texts as meta expressions. When a Froglingo expression appears between **`[`** and **`]`**(a pair of bolded square brackets), it means that the expression is optional.

In the document, a sample expression is given in the context of a database that accumulates data from earlier sample expressions. In other words, the value evaluated from a given expression may be different under a different sample database.

Chapter 2: DATA CONSTRUCTION

This chapter introduces the concepts that allow users to construct business data. There is not a precise definition about what is business data. But here we say that business data is those computer presentations that represent or model a finite set of entities and their relationships in the world.

2.1 Constant

Like in other languages, integers, real numbers, and strings are constants (basic data types). Froglingo recognizes constants by default. When a constant is entered, it is simply echoed back. For examples:

```
[//] "Hello World";
"Hello World";
[//] 3.14;
3.14;
```

The basic mathematical operations plus (+), minus (-), multiplication (*), division (/) and modulus (%) among numbers are supported. For example:

```
[//] 3 - 5.5;
-2.5;
```

The operator plus (+) can be applied among numbers and strings, which concatenates two operants. For example:

```
[//] "The pie is " + 3.14 + ".";
"The pie is "3.14"."
```

There are a few Froglingo specific constants: `null`, `true`, and `false`. The constant `null` is a special one used to represent "not defined". It can be a return value from a query expression when the expression yields with no value. The constants `true` and `false` are the two boolean constants.

A string surrounded by the single quote, such as `'2/20/2009'`, representing date and time, is also a constant. It is stored as an integer in Froglingo; but can be displayed in a format users desired. Please see the detailed discussion in Chapter 8.

2.2 Identifier

An identifier is a sequence of ASCII characters including letters, digits, and the special characters '_', and '.'. When the sequence forms an integer or a real number, it is not an identifier. When the first character of the sequence is the character '.', it is not an identifier. The examples of identifiers are `Salary`, `Mike`, `Word123_`, `_basic`, `www.mycompany.com`.

While a constant is automatically recognized, an identifier must be "created" before a Froglingo database recognizes it. One way of declaring identifiers is to explicitly create it by using the built-in operator "`create`". For examples:

```
[//] John;
null;    /* John is not recognized */
[//] create John;
```

```
[//] John;
John;  /* John is recognized now*/
```

There are built-in identifiers `void`, `error`, `timestamp`, and `signature`. They have their special meanings as we will see along the document.

Note that identifiers are different from strings even if one identifier and one string share the same sequence of ASCII characters, e.g., John and "John". Users should be careful on the differences during software development.

2.3 Term

A term is a constant, an identifier, or a pair of parenthesized terms. In other words:
1. If **T** is a constant, then **T** is a term,
2. If **T** is an identifier, then **T** is a term,
3. If **T1** and **T2** are terms, then (**T1 T2**) is a term.

The examples are `3.14`, `Mike`, `(Mike Salary)`, `((country state) county)`, `(tax (Mike salary))`, and `(3.14 (salary "Hello World"))`.

A term that consists of an ordered pair of two terms, is called a combinatory term, spelled as comb-term. The first term of a comb-term is called the plus-term; and the second term the minus-term. For example, the comb-term `(Mike salary)` has `Mike` as the plus-term and `salary` as the minus-term.

If the minus-term of a term is not a comb-term, the parentheses surrounding the term don't have to be written. For example, `((country state) county)` is equivalent to `country state county`; and `((a b) (c d))` is equivalent to `a b (c d)`.

Like an identifier, a comb-term must be explicitly created before it is recognized. For example:

```
[//] country state county;
null;
[//] create country state county;
[//] country state county;
country state county;
```

A term in a comb-term is called an inner-most or a left-most if there is no more term at the left of it. Given term `a b c d`, for example, `a` is an inner-most term; so are the terms `a b`, `a b c`, and `a b c d` itself. As another example, the term `(e (f (g h)))` has `e` as its only inner-most term.

A term in a comb-term is called an outer-most or a right-most term if there is no more term at the right of it. Given term `a b c d`, for example, `d` is the only outer-most term. As another example, the term `(e (f (g h)))` has `h`, `g h`, `f (g h)`, and `(e (f (g h)))` as its outer-most terms.

A term appearing within a second term is called a sub-term of the second term. All the comb-terms, constants, and identifiers appearing in a term are sub-terms. For example, `a b c d` has sub-terms `a`, `b`, `c`, `d`, `a b`, `a b c`, and `a b c d`. Note that `b c` are not `a` sub-term of `a b c d` because b c is not a sub-term in `(((a b) c) d)`.

2.4 Assignment and Database

An assignment is a state that a term takes another term as its value. For example, `Mike salary = 2000, 2 = 3,` and `a = b`. Not all the assignments are valid and allowed to be stored in a database. To be recognized by a database, an assignment needs to be declared first. Here are examples:

```
[//] create a_number = 1;
[//] create a b (c d) = 6;
[//] a b (c d);
6;
[//] create y u = a b;
[//] y u;
a b;
[//] create Mike salary = 1000;
[//] create Dave salary = 2000;
[//] create Bob = Dave;
[//] create income = salary;
[//] create January 1 = "New Year";
```

Given an assignment, the term at the left side of the symbol '=' is the assignee (also called entity or enterprise); and the term at the right side the assigner (also called value).

As discussed in Section 1.3, a term without explicit assigner will be allowed to be in a database. To give the definition of database, we call a term without assigner an assignee too. We do so because a term without assigner has a derived value semantically. To show a term without assigner in a database meaningful, we echo the term itself back to users. Here are examples:

```
[//] a b;
a b  /* given that "create a b (c d) = 6;" has been entered earlier*/
[//] country state county;
country state country /* given that it has been entered earlier in Section 4*/
```

A database is a finite set of assignees, each of which has the following restrictions:
1. A constant cannot have an assigner by itself, and cannot be a plus-term,
2. The minus-term of a comb-term appeared in the assignee must not have an assigner; and
3. Assignments cannot form a circle: If there is a sequence of assignments: $M_0 = M_1, M_1 = M_2, \ldots, M_{n-1} = M_n$, M_n must not be M_0.

The first restriction says that a constant defines itself and it is not allowed to explicitly assign another value for it. The second restriction is not necessary theoretically, but it helps to improve system performance. The third restriction is to avoid undesirable loops both practically and theoretically.

Here are the examples of invalid assignments:

```
[//] create 55 = 1;
Creation operation failed. Constant 55 is not allowed to have an assigner.
[//] create 55 john;
Creation operation failed. Constant 55 is not allowed to be a plus-term.
```

```
[//] create B (6 C) = 343;
Creation operation failed. Constant 6 is not allowed to be a
plus-term.
[//] create e f = 66;
[//] create E (e f) F = 88;
Creation operation failed. the term (e f) having an assigner
cannot be a minus-term of a comb-term.
[//] create c3;
[//] create c1 c2 = c3;
[//] update c3 = c1 c2;
There is an assignment loop having node c3. Update operation
failed.
```

From the definition, we can see that a database has the following properties:
1. If a comb-term is in database, its plus-term and minus-term are both in database.
2. If a comb-term is in database, its plus-term doesn't have an (explicit) assigner.
3. If a comb-term is in database, its minus-term is either a constant, or a term without assigner.
4. An assigner must be a constant, or an assignee in database already.
5. An assignment is always identified by its assignee. In other words, an assignee is not only for itself, but also implies its assignment.

The concepts introduced so far is sufficient for constructing business data. Here we construct a sample database for a school administration:

```
[//] create SSD John SSN = 123456789;
[//] create SSD John birth = '6/1/1990';
[//] create College admin (SSD John) enroll
                        = '9/1/2008';
[//] create College CS;
[//] create College admin (SSD John) Major
                        = College CS;
[//] create College CS CS100
            (College admin (SSD John)) grade = "F";
```

The sample database constructed above says: John, born on 6/1/1990, is a resident registered with his SSN = 123456789 in Social Security Department (SSD); he was enrolled in College on 9/1/2008 and majored in Computer Science (CS); and his grade is "F" in course CS101. Though the sample database is small, it is intended to show that it can manage residents, colleges, organizational structures in colleges, activities of students in colleges, and the relationships among the managed objects.

2.5 Dependent Relationships

In the real world, we say that one thing depends on another if the existence of the first depends on the existence of the second. For example, human beings depend on the Earth; a child object depends on its parent object in a hierarchy; and the birth to an infant depends on both mother and father. In mathematics, the process (and therefore the result) of applying a function to an argument depends on the argument and the function. Give a function f(x) = x + 1 and an argument 4, for example, we say that the process of applying function f to the argument 4, i.e., f (4), or (f 4) in Froglino term, is dependent on both the

function f and the argument 4.

There are dependent relationships among the terms in a database. Given a comb-term (**`t1 t2`**) in a database, we say that the comb-term (**`t1 t2`**) functionally depends on **`t1`** and argumentatively depends on **`t2`**. These dependent relationships lead the following operators available in Froglingo.

2.5.1 Functional Dependency

Binary operators: `{+, }+`
Unary operator: `pterm`
Definition: If there is a term (**`M N`**) in database, the following binary operations are evaluated to be true:
`M N` `{+` **`M`**,
`M` `}+` **`M N`**.
Further the unary operation `pterm` (**`M N`**) is evaluated to be **`M`**. Examples:

```
[//]((College CS) CS100) {+ College CS;
true;
[//] pterm (College CS CS100);
College CS;
```

2.5.2 Argumentative Dependency

Operators: `{-, }-`
Unary operator: `mterm`
Definition: If there is a term (**`M N`**) in database, the following binary operations are evaluated to be true:
`M N` `{-` **`N`**,
`N` `}-` **`M N`**.
Further, the unary operation `mterm` (**`M N`**) is evaluated to be **`N`**. Examples:

```
[//] ((College admin) (SSD John)) {- SSD John;
true;
[//] mterm (College admin (SSD John));
SSD John;
```

2.5.3 Recursively Functional Dependency

Binary operators: `{=+, }=+`
Definition: If **`N`** is an inner-most term of term **`M`** in a database, the following operations are evaluated to be true:
`M` `{=+` **`N`**,
Or equivalently:
`N` `}=+` **`M`**.

Examples:

```
[//] College admin (SSD John) {=+ College admin;
true;
[//] College admin {=+ College;
true;
[//] College admin (SSD John) {=+ College;
```

```
ture;
```

It is clear from the definitions and the example aboe that if **N** {+ **M**, then **N** {=+ **M.** The recursively functional dependency is transitive, i.e., if **M** {=+ **N** and **N** {=+ **Q**, then **M** {=+ **Q**.

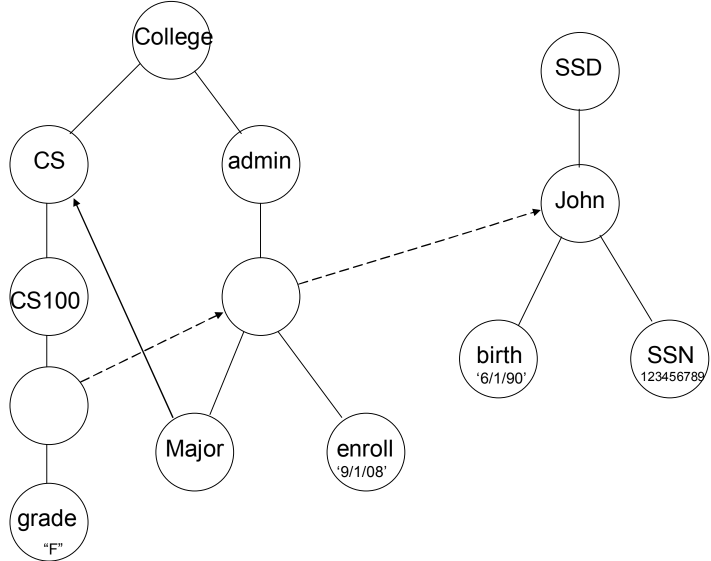
2.5.4 Recursively Argumentative Dependency

Binary operators: `{=-, }=-`
Definition: If **N** is a outer-most term of term **M** in a database, the following operations are evaluated to be true:
M {=- **N**,
Or equivalently:
N }=- **M**.

Examples:
```
[//] College CS CS100 (College admin (SSD John)
       {=- College admin (SSD John);
true;
[//] College admin (SSD John) {=- SSD John;
true;
[//] College CS CS100 (College admin (SSD John)) {=- John;
true;
```

It is clear from the definitions and the examples that if **M** {- **n**, then **M** {=- **N**. The recursively argumentative dependency is transitive, i.e., if **M** {=- **N** and **N** {=- **Q**, then **M** {=- **Q**.

Before moving to the next section, we point out that the dependent relationships align terms in database to hierarchical structures. To show the hierarchical structures, we provide a graphical view of the sample database of the school administration in Section 2.4.



In the graph above, each circle represents an assignee in database. A root node represents an identifier, where the identifier is spelled out in the circle center. A non-root node represents an application, where the left sub-term is spelled out in the circle center. The leaf nodes are the assignments normally having explicit assigners. A solid up-down link connects an application to its left sub-term. A dash arrow connects an application to its right sub-term. A solid arrow connects an assignee to its assigner. For those assignees whose values are constants or other non-assignees their values spelled out in the cycles.

The up-down links and dash arrows represent the relations {+ and {-. Under each of the relations, the graph forms tree(s), and further the leaf nodes in trees depend on the upper nodes.

2.6 Database Update

In addition to the operator `create`, there are three more operators that can be used to keep a database evolving: `update`, `delete`, and `record`. This section gives a comprehensive discussion on them.

Given two terms **`M`** and **`N`**, the syntactical forms are:

(create **M [** = **N]**),

(record **M [** = **N]**),

(update **M** = **N**), and

(delete **M**).

When a pair of braces **`[`** and **`]`** appears in the meta expressions above, the content surrounded by the pair is optional. Therefore the part "= **N**" is optional for the operators

`create`, and `record`.

In the meta expressions above, we added a pair of parentheses (and) for each operation expression because we treated a update operator as a term too. You may drop the parentheses off whenever there is no ambiguity.

2.6.1 Create Operation

A create operation, when it is executed, requires that the assignee is new to database. A sub-term of the assignee can be existed in database. But when it is not in database, it will be automatically created too.

When a create operation has an assigner, each sub-term of the assigner don't have to be in database. But each identifier in the assigner must be in database already. For example:

```
[//www.a_trial.com] create t1 t2 = t3;
The term t3 is not in database. Or you don’t have access to the
term t3.
Create operation is not successful.
```

Preventing assigner from including undefined identifiers is to prevent users from entering unintended expressions. It helps users in debugging.

2.6.2 Assignment Update

An update operation, when it is executed, requires that the assignee exists in database and an assigner must be provided. The assigner is a term, in which all of its identifiers have been in database. For example,

```
[//www.a_trial.com] update Mike salary = Dave salary;
[//www.a_trial.com] update country state county = “Somerset”;
[//www.a_trial.com] update Mike Dave = 3;
The term (Mike Dave) is not an assignee in database.
Update operation is not successful.
[//www.a_trial.com] update Mike salary = t3;
The term t3 is not in database. Or you don’t have access to the
term t3.
Update operation is not successful.
```

2.6.3 Delete Operation

When a term is deleted from database, all the other terms that are functionally and argumentatively dependent on the given term are deleted too.

```
[//www.a_trial.com] print .;
Bob = Dave;
College CS CS100 (College admin (SSD John)) grade = "F";
College admin (SSD John) Major = College CS;
College admin (SSD John) enroll = 1220245200;
Dave salary = 2000;
January 1 = “New Year”;
Mike salary = 1000;
SSD John SSN = 123456789;
SSD John birth = 644216400;
a b (c d) = 6;
a_number = 1;
```

```
country state county = "Somerset";
c1 c2 = c3;
e f = 66;
income = salary;
y u = a b;
[//www.a_trial.com] delete c1;
[//www.a_trial.com] print .;
Bob = Dave;
College CS CS100 (College admin (SSD John)) grade = "F";
College admin (SSD John) Major = College CS;
College admin (SSD John) enroll = 1220245200;
Dave salary = 2000;
January 1 = "New Year";
Mike salary = 1000;
SSD John SSN = 123456789;
SSD John birth = 644216400;
a b (c d) = 6;
a_number = 1;
country state county = "Somerset";
e f = 66;
income = salary;
y u = a b;
```

Here, we used the command "`print .`" to list the assignments stored in database. More discussion about the command print will follow later.

2.6.4 Record Operation

A create operation fails if the assignee exists in database. An assignment update fails if the assignee is not in database. When a user is not sure if an assignment exists or not and wants to commit its construction in database anyway, the user can use the operation `Record`. The record operation is typically useful when one needs to upload a set of assignments to database. More discussion on upload process is presented in Chapter 3. Here are some examples:

```
[//www.a_trial.com] create Mike salary = 7000;
The term Mike salary has existed already. Create operation failed.
[//www.a_trial.com] record Mike salary = 7000;
[//www.a_trial.com]
```

The record operation provides users with more flexibility, but increases the risk of unexpected change on database. Users need to be careful when using it.

Chapter 3: DATA QUERY

Given a database, an arbitrary term that was defined in Chapter 2 can be evaluated to have a unique value (called the normal form of the given term). Terms are themselves query expressions. In addition, Froglingo offers "select" operations that output sets of values.

In this chapter, we discuss how a term is evaluated to its normal form; and how we use the select command to support set-oriented operations. For the readers who are interested in advanced queries, we introduce a set of built-in operators that reflects pre-orderings relationships among managed data. They can be used to manage business data having complex relationships including cyclical relationships.

3.1 Normal Form and Evaluation

Any terms, under the environment of a database, can be evaluated to unique values – normal forms. With a given database, a term is called a normal form if

1. it is a constant, or
2. it is in the database and doesn't have an assigner.

For example, `null, 3.14, College, College CS,` and `salary` are all normal forms under the database that was collected so far in this document.

An arbitrary term can be evaluated to its normal form. Here are the evaluation rules:

1. If an identifier is not in the database, it is reduced to `null`.
2. If a term is in the database and has an assigner, its normal form is the normal form of its assigner.
3. A comb-term having a constant as its plus-term is reduced to `null`.
4. If two terms **`M`** and **`N`** have normal forms **`M′`** and **`N′`**, then the normal form of the comb-term **`(M N)`** is the normal form of the comb-term **`(M′ N′)`**.

Here are examples of evaluation processes under the database we have so far:

```
[//] 44;
44;
[//] 23 John;
null;
[//] a_undefined_id;
null;
[//] a b (c d);
6;
[//] a b;
a b;
[//] Dave salary;
2000;
[//] Bob salary;
2000;
```

```
[//] College CS CS100 (College admin (SSD John)) grade;
"F";
[//] College CS CS100;
College CS CS100;
[//] College admin (SSD John) Major CS100;
College CS CS100;
```

There are infinite terms while finite assignments are defined in database. The rules guarantee that each of the infinite terms can be reduced to its unique normal form at the state of a database at a given time. Since database is evolving, the normal form of a term at a given time may be different from the one at another time.

For some business reasons, however, one may not want a term to be reduced to its normal form but to keep its original canonical form. Here we introduce a built-in operator `canon`, which takes a term in database as input and returns the canonical form of the term as output. For example:

```
[//] Dave salary;
2000;
[//] canon (Dave salary);
Dave salary;
```

3.2 Arithmetical and Boolean Operators

Before starting the discussion of set-oriented operations, we introduce the arithmetical and Boolean operations commonly used in programming languages and DBMSs.

3.2.1 Arithmetical

Binary operators: `+, -, *, /, %`.

The operants of the operators are reduced to their normal forms before the operators are applied. Like in other languages, the operators minus `-`, multiplication `*`, quotient `/`, and remainder `%` only apply to numbers. The symbol + acts as the plus operator when two operants are numbers. Otherwise it syntactically concatenates two operants. Here are examples:

```
[//] 22.2 + Dave salary;
2022.2;
[//] 1000 – Dave salary;
-1000;
[//] Dave salary * 0.3;
600;
[//] Dave salary / 5;
400;
[//] Dave salary % 1999;
1;
[//] "Dave's salary is " + Dave salary;
"Bob's salary is " 2000;
```

3.2.2 Boolean on Numbers

Binary operators: `<, >, <=, >=`.

The operants of the operators are reduced to their normal forms before the operators are applied. Like in other languages, the operators less than <, bigger than >, less than or equal <=, and bigger than or equal >= only apply to numbers. Here are examples:

```
[//] 22.2 <= Dave salary;
true;
```

3.2.3 Complex Boolean

Binary operators: `and, or`.

The operants of the operators are reduced to their normal forms before the operators are applied. Like in other languages, the operators and `and`, and or `or` only apply to Boolean constants `true` and `false`. Here are examples:

```
[//] 22.2 <= Dave salary;
true;
[//] (22.2 <= Dave salary) and true;
true;
```

3.3 Set-Oriented Query

With the Boolean operators discussed in sections 2.5 and 3.2, and more to be introduced in 3.4, we can express set-oriented queries: the outputs of queries are not single terms, but sets of terms. This section formally introduce the expressions consisting of variables, the select operator, and other attribute clauses.

3.3.1 Variable

A variable is an identifier prefixed with the character `$`. A variable is also a term in Froglingo. Therefore, the combinations of a variable with other terms are terms too. For example, `$var`, `$person salary`, and `College CS CS100 $student Grade` are all terms.

3.3.2 Select Operator

The basic select expression is defined with the following syntactical form:

(select $\mathbf{M_1}$, $\mathbf{M_2}$, …, $\mathbf{M_n}$ where **condition_clause**)

Where,

1. **condition_clause** is a boolean expression, in which there is at least one variable.
2. $\mathbf{M_1}$, $\mathbf{M_2}$, …, $\mathbf{M_n}$ is a sequence of terms delimited by ','. If a variable appears in the sequence, it must appear in **boolean_exp**.

Here a few examples are provided to intuitively demonstrate how the system reacts on select expressions.

```
[//] select $p, $p salary, ($p salary * 0.3) where $p salary >=
1000;
```

```
Bob, 2000, 600
Dave, 2000, 600
Mike, 1000, 300;
```

For this query expression, the system retrieves all the persons whose salaries are greater than or equal to 1000. Then for each person retrieved, the system calculates and prints out the person's name, salary, and tax.

```
[//] select $x where $x {=+ SSD;
SSD
SSD John
SSD John SSD
SSD John birth;
```

For this query expression, the system retrieves all the entities functionally dependent on SSD, i.e., the residents and their attribute names registered under Social Security Department.

```
[//] select $x where $x {=- John;
John
SSD John
College admin (SSD John)
College CS CS100 (College admin (SSD John));
```

For this query expression, the system retrieves all the entities argumentatively dependent on John, i.e., all the activities John participated in society.

```
[//]select $college, $dept, $student where $college $dept $class
($college admin $student) grade == "F" and $college admin
$student enroll > '1/1/2006' and $class == CS100 and ($college ==
College or $college == ABC_Univeristy);
College, CS, SSD John;
```

For this query expression, the system retrieves the students who were enrolled after 1/1/2006 and whose grades were "F" in class CS100 in college `College` or `ABC_University` (an identifier is not in database). Then for each student selected, print out the college and the department that offered the classes CS100; and the student.

For the readers who are interested in a more formal introduction of the select operator, we discuss more on how the system reacts to a select expression. Given a set of variables appeared in **condition_clause**, v_1, v_2, …, v_p, the system searches database and find a relation:

V_{01}, V_{02}, …, V_{0p}

V_{11}, V_{12}, …, V_{1p}

…

V_{q1}, V_{q2}, …, V_{qp}

such that the substitution:

condition_clause:[v_1:= V_{i1},…, v_p:= V_{ip}]

is evaluated to be true for each i between 0 and q. Note that each element V_{ij}, here i is between 1 and p, and j is between 0 and q, is either a term in database. Futher, for each row V_{i1}, V_{i2}, …, V_{ip}, the system calculates and prints out the normal form of the substitution:

$M_1, M_2, \ldots, M_n$:[$v_1 := V_{i1}, \ldots, v_p := V_{ip}$]

Please reference Section 4.17 for the definition of substitution.

3.3.3 Index Attribute

In the syntax of select operations: (select $M_1, M_2, \ldots, M_n$ where **boolean_operation**), an element of the output list $M_1, M_2, \ldots, M_n$ can be the built-in operator `index`. For the ith row of the generated output, `index` is replaced with number i. For example:

```
[//] select  index, $p, $p salary, ($p salary * 0.3) where $p
salary >= 1000;
1, Bob, 2000, 600
2, Dave, 2000, 600
3, Mike, 1000, 300;
```

3.3.4 Sort Clause

The order of output rows can be specified by using a sort clause. The syntax is:

```
(select M1, M2, …, Mn
      where  boolean_operation
      [sort n1 order, …, nm order]
)
```

Here $n_1, \ldots, n_m$ are a sequence of numbers, each of which represents an element position of the output list $M_1, M_2, \ldots, M_n$. The parameter **order** can be either `ascent` or `descent`. To sort salaries (the column 2) in the ascent order, for example, we give the following expression:

```
[//] select $p, $p salary where $p salary >= 1000 sort 2 ascent;
Mike, 1000
Bob, 2000
Dave, 2000;
```

3.3.5 Summary Clause

The aggregation functions maximum, minimum, average, sum, and count can be applied to the selected set and provide a summary for a select operation. Here is the enhanced syntax:

```
(select M1, M2, …, Mn
      where  boolean_operation
      [sort n1 order, …, nm order]
      [summary t1, …, tj]
)
```

Here $t_1, \ldots, t_j$ are a sequence of terms, each of which may be a constant, the parameter `count`, or a regular term that may or may not include a sub term:

aggregate n.

Here **aggregate** is one of the following: `max`, `min`, `ave`, and `sum`; and **n** is a number representing a position of the output list $M_1, M_2, \ldots, M_n$. Here is an example:

```
[//] select $p, $p salary where $p salary >= 1000
      sort 1 ascent
      summary "Summary:", count, ave 2,sum 2, (0.3 * (sum 2));
Bob, 2000
```

```
Dave, 2000
Mike, 1000
"Summary:", 3, 1666, 5000, 1500;
```

3.4 Derivative Relationships

For the readers who are not interested, this section can be skipped.

In Section 2.5, we said that a comb-term depends on its plus-term functionally and its minus-term argumentatively because if the existence of the comb-term in a database implies the existences of its plus-term and its minus-term in the database too. In this section, we define:

1. Two terms are equal if they have the same normal form.
2. If a com term **`(m n)`** is equal to another term **`q`**, and **`(m n)`** and **`q`** are in a database, then we say that **`q`** is functionally derivative from **`m`**; and argumentatively derivative from **`n`**.

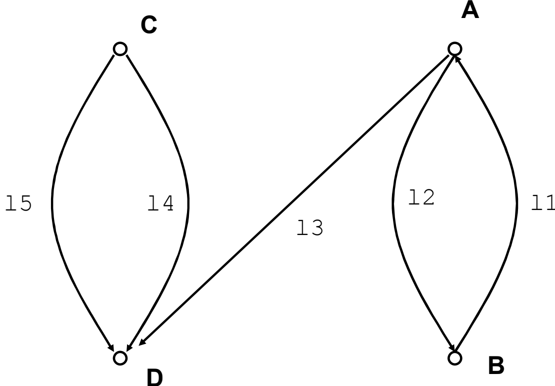
When a term is derivative from another term, it may no longer be dependent on the second term. Given the function f (x) = x + 1 and the argument 4, for example, the process, i.e., f (4) or (f 4), of applying f to 4 is ended with the value 5. Therefore both the process (f 4) and the value 5 are functionally derivative from the function f; but 5 is not dependent on f. This section gives the binary operators stemming from the derivative relationships.

To show that Froglingo is able to manage directed graphs possibly including circles in the coming sub-sections, we give the following examples:

```
[//] create A l1 = B;
[//] create B l2 = A;
[//] create A l3 = D;
[//] create C l4 = D;
[//] create C l5 = D;
```

Here, the directed graph consists of 4 vertices: `A`, `B`, `C`, and `D`; and 5 directed links: `l1` from `A` to `B`, `l2` from `B` to `A`, `l3` from `A` to `D`, `l4` from `C` to `D`, and `l5` from `C` to `D`. With the Froglingo expressions above, we can have the following interesting queries:

```
[//] A l1
B;   /* Starting from A, one can reach B by following link l1 */
[//] A l1 l2 l3;
D;  /* Starting from A, one can reach D by following links l1, l2, and l3 */
[//] A l1 l2 l1 … l2;
A;  /* Starting from A, one can "walk" along the circle as many times as desired by following links l1 and l2*/
```



A Sample Directed Graph

3.4.1 Equation

Binary operators: `==, !=`
Unary operator: `not`
Definition: If two terms **m** and **n** have the same normal form, then **m** is equal to **n**, i.e., **m** == **n.** If two terms **m** and **n** have two different normal forms, then **m** is not equal to **n**, i.e., **m** != **n**, or `not` (**m** == **n**)**.** Examples:

```
[//] Bob salary == 2000;
true;
[//] Dave income == Bob salary;
true;
[//] Bob salary != 2000;
false;
[//] Bob != salary;
true;
```

To retrieve all the terms that are equal to 2000, we have the following select expression:

```
[//] select $x where $x == 2000;
2000
Dave salary
Bob salary;
```

Two assignees not having explicit assigners have themselves as normal forms, therefore they are never equal.

Note that the operator `!=` is not correctly functioning in this release in set-oriented operations discussed in Section 3.3.

3.4.2 Functional Derivative

Binary operators: `<+, >+`
Definition: Given terms **m n** and **q** in a database, if **m n** == **q**, then **q** is a functional derivable from **m**, i.e., **q** <+ **m**; or equivalently **m** >+ **q** . Examples:

```
[//] 2000 <+ Bob;
```

```
true; /* 2000 is a derivative (salary) of Bob */
[//] College CS <+ College admin (SSD John);
true; /* College CS is a derivative (Major) of College admin (SSD John)*/
[//] D <+ C;
true;
[//] C <+ D;
false;
[//] A <+ B;
true;
[//] B <+ A;
True;
```

In the directed graph constructed with the vertices A, B, C, and D earlier, we have the following expression to find all the vertices that have directed links to D:

```
[//] select $v where $v >+ D;
C
A
```

3.4.3 Argumentative Derivative

Operators: `<-, >-`
Definition: Given terms **`m n`** and **`q`** in a database, if **`m n`** `==` **`q`**, then **`q`** is an argumentative derivative from **`n`**, i.e., **`q`** `<-` **`n`**; or equivalently **`n`** `>-` **`q`**.
Examples:

```
[//] 2000 <- salary;
true; /* 2000 is a derivative of salary, i.e., someone's salary is 2000 */
[//] 2000 <- income;
true;
[//] College CS <- Major;
true; /*College CS is a derivative of Major, i.e., someone's major is College CS */
```

In the directed graph constructed with the vertices A, B, C, and D earlier, we have the following expression to find the directed links that terminate at D:

```
[//] select $v where $v >- D;
L3
L4
L5;
```

3.4.4 Recursively Functional Derivative

Binary operators: `<=+, >=+`
Definition: If **`m`** is an inner-most term of **`p`** and **`p`** is equal to another term **`q`**, here **`m`**, **`p`**, and **`q`** are in a database, then **`q`** is a recursively functional derivative from **`m`**, i.e., **`q`** <=+ **`m`**, or equivalently **`m`** >=+ **`q`**.

Examples:

```
[//] 2000 <=+ Bob;
true;
[//] 2000 <=+ Robert;
true;
[//] "F" <=+ College;
true;
```

```
[//] College <=+ College;
true;
```

In the directed graph constructed with the vertices A, B, C, and D earlier4, we have the following expression to find if there is a path from B to D:

```
[//] B >=+ D;
true;
```

The query: "Is there a circle having vertices `A` and `B`?" is represented as:

```
[//] A <=+ B and B <=+ A;
true;
```

To find all the vertices that are reachable from A is expressed as:

```
[//] select $x where $x <=+ A;
A
B
D;
```

3.4.5 Recursively Argumentative Derivative

Binary operators: `<=-, >=-`
Definition: If **n** is an outer-most term of term **p** and **p** is equal to another term **q**, here **n**, **p**, and **q** are in a database, then **q** is a recursively argumentative derivative from **n**, i.e., **q** `<=-` **n**, or equivalently **n** `>=-` **q**.

Examples:

```
[//] 2000 <=- salary;
true;
[//] "F" <=- grade;
True;
[//] College
true;
[//] College <=- College;
ture;
```

3.4.6 Properties of Derivative Relationships

We introduced quite a lot of relationships in Section 2.5 and this section 3.4. Due to the nature of their definitions, they are related. To better understand them, we summarize how they are related with each others. Given a database, the following properties hold:

1. if **p** `{+` **q**, then **p** `<+` **q**,

2. if **p** `<+` **q**, then **p** `<=+` **q**,

3. if **p** `{=+` **q**, then **p** `<=+` **q**.

Similarly,

4. if **p** `{-` **q**, then **p** `<-` **q**,

5. if **p** `<-` **q**, then **p** `<=-` **q**,

6. if **p** `{=-` **q**, then **p** `<=-` **q**.

In addition, the operators <=+ and <=- are pre-ordering relations, i.e., reflexive and transitive while the operators {=+ and {=- are partial ordering relations, i.e., reflexive, anti-symmetric, and transitive.

Chapter 4: BUSINESS LOGIC

Managing business data is the first, but not the final step in database application development. Business logic, though not a formally defined concept, is the computer presentation for business work flows and user-server interaction flows. Mathematically, a language supporting business logic must be able to deal with infinite data. To represent the logic of traffic light, for example, we have to consider the exception that a light is neither red, green, nor yellow. This exception implies an infinite number of the colors other than the 3 colors.

Another requirement of business logic is multiple actions trigged by a single event. A typical example is trade. A trade wouldn't happen unless objects are exchanged between two parties simultaneously.

By adding variables and sequential terms, Froglingo becomes a unification for business data and business logic, and is sufficient for arbitrary database applications..

4.1 Infinite Data

We have introduced variables in Section 3.3. A variable in a select operation is a placeholder for a finite set of terms satisfying the boolean condition given in the select operation. The variable is bounded to the select operation itself, and not visible to the rest of database.

In this section, we introduce variables in the scope of a database. A variable in database allow a database with limited space to manage infinite data.

4.1.1 Variable and Database

From Section 3.3, we know that a variable is formed by an identifier prefixed by '`$`', and the definition of term has been extended with variable. Therefore, `$x person, tax $money, A $x B $y` are terms too.

Variables can appear as sub-terms of assignees in database. But there are two restrictions:

1. If a variable appears in an assigner, it must appear in assignee, and
2. A variable cannot be a plus-term in assignee.

Here are some examples of valid assignments:

```
[//] create tax $money = ($money * 0.3);
[//] create Grade $x $y = College CS $y $x grade;
[//] create fun $x 1 $y = ($x + $y);
[//] create fun $x 2 $y = ($x * $y);
```

When a variable is to be declared, what an identifier is chosen for the variable is not important. The importance is that the variable in assigner matches the intended variable declared in assignee.

Variable serves as a placeholder for the domain (i.e., the complete set of arguments) of a function. Variables virtually bring infinite number of entities into database. For example, the assignment

```
tax $money = ($money * 0.3)
```

is mathematically equivalent to the following Froglingo assignments without a variable:

```
tax 0 = 0;
tax 1 = 0.3;
tax 2 = 0.6;
…
tax n = (n * 0.3);
```

Here we only considered a non negative integer to be arguments and `n` is a positive number as large as user wants.

4.1.2 Variable Range

Business needs may require a range of values that a variable can take. For example employee salaries are always non-negative. In Froglingo, we allow a variable in an assignee optionally to have a range. The syntax is:

```
$ID: [bool_exp]
```

Here, **`bool_exp`** is a boolean expression embedding variable `$ID`. For example, we can re-define the function tax:

```
[//] create tax $money:[$money >= 0] = ($money * 0.3);
```

Note that when a variable is defined without a range, it takes all the constants and all the assigners in database as its range.

Sometime, a function may need to behave differently depending on what an argument is applied. For example, we want the tax rate is 0.2 for the people whose salary are less than 20,000; 0.5 for the people whose salary is larger than 250,000; and 0.3 for the rest of the people. This requirement can be expressed in Froglingo:

```
[//] delete tax $money; /* remove the assignment tax $money */
[//] create tax $r1:[$r1 >= 0 and $r1 < 20000] = ($r1 * 0.2);
[//] create tax $r3:[$r3 >= 20000 and $r3 < 250000] = ($r3 * 0.3);
[//] create tax $r2:[$r2 >= 250000] = ($r2 * 0.5);
```

In this case, Froglingo accepted multiple variables with the corresponding range for the domain of the function `tax`. When multiple variables are entered for a single function as we did above, Froglingo system stores them in the alphanumerical order of the variables under the function. Here is the sequence of the 3 assignments in database for the sample function `tax` above:

```
[//] print tax;
tax $r1:[$r1 >= 0 and $r1 < 20000] = ($r1 * 0.2);
tax $r2:[$r2 >= 250000] = ($r2 * 0.5);
tax $r3:[$r3 >= 20000 and $3 < 250000] = ($r3 * 0.3);
```

When the system is trying to find which range a user-entered instance falls into, the

system sequentially checks the multiple ranges starting from the one whose variable identifier has the lowest alphanumeric order; and pick up and stop at the first one that has a match. Given an instance, such as `21000` as an example, the system attempts to see if `21000` is in the range of the variable `$r1` first. When the system found that it is not the case, it moves to the next variable `$r2`. When the system moved to the last variable `$r3`, and determined that `21000` is in the range of `$r3`, the assigner of the third assignments above is used to calculate the value, i.e., `6300`, for `tax 21000`.

When multiple variables are entered for the domain of a function, the system doesn't check if the multiple ranges have an overlap. For example, the tax function can be re-defined as:

```
[//] update tax $r1:[$r1 >= 0 and $r1 < 20000] = ($r1 * 0.2);
[//] update tax $r2 = ($r3 * 0.3);
[//] update tax $r3:[$r3 >= 250000] = ($r3 * 0.5);
```

Here, there is no range specified for the variable `$r2`. When a user specifies an instance such as `300000`, for example, the system will stop at `$r2`, and will not reach the expected variable `$r2`. It's user's responsibility to ensure that an overlap among the multiple ranges doesn't cause any conflicts with business needs.

With variables in database, one function may be defined with a domain consisting of the ranges of variables and individual argument instances. To interpret traffic lights, for example, we can define the following assignments:

```
[//] create light red = "to stop";
[//] create light green = "to go";
[//] create light yellow = "be cautious";
[//] create light $x = "The color " + $x + " is not a color for traffic lights";
```

When an argument instance is applied to a function with the domain of having both individual instances and variables, the system always tries to match one of individual instances first.

4.1.4 Query on Infinite Data

Variables bring functions with infinite data. We can query the properties of the functions. In this section, we intuitively discuss the query processes of Froglingo system when variables are involved. For those readers who are interested in a precise understanding on evaluation rules, please reference Section 4.1.6 for more information.

```
[//] tax (Bob salary);
400;
```

Bob salary is reduced to `2000`, and `2000` is replacing the variable `$money in tax $money = ($money * 0.3)`.

```
[//] Grade (College admin (SSD John)) CS100;
"F";
```

The facts leading to the result are:

```
Grade $x $y = College CS $y $x grade;
College CS CS100 (College Admin (SSD John)) = "F";
```

```
[//] fun 2 1 3;
```

```
5;
```
The fact leading to the result is:
```
fun $x 1 $y = ($x + $y);
```
```
[//] light green;
"To go";
```
The fact led the result is:
```
Light green = "To go";
```
```
[//] select $p, $p salary, tax ($p salary) where $p salary >=
1000;
Bob, 2000, 600
Dave, 2000, 600
Mike, 1000, 300;
```

4.1.3 Function Recursion

With variable introduced, we can define a function by directly or indirectly referencing itself. For example, the factorial function can be defined in Froglingo:
```
[//] create fac 0 = 1;
[//] create fac $n = ($n * (fac ($n - 1)));
```
Here the function `fac` appeared in the body (the assigner) of defining the function `fac` itself. Then we can express a query:
```
[//] fac 4;
24;
```
The facts leading to the result are:
```
fac $x = ($x * (fac ($x - 1)));
fac 4 = (4 * (fac 3));
fac 3 = (3 * (fac 2));
fac 2 = (2 * (fac 1));
fac 1 = (1 * (fac 0));
fac 0 = 1;
```
Given two vertices X and Y in a directed graph, between which there is not a circular directed links, we can print out all the paths from X to Y by using the following expression:
```
create path $a $b:[$b <+ $a or $b == $a] = ($a + $b);
create path $a $c  = select ($a + path $z $c) where $z <+ $a and
$c <=+ $z;
```
To avoid the circle between `A` and `B` in the sample directed graph given in Section 3.4 and to demonstrate the effect of the function `path`, we add one more link:
```
[//] create D l6 = E;
```
Then,
```
[//] path C E;
CDE;
```
Function recursion brings the full power of a programming language into Froglingo. The power is the Turing-equivalent programs or partial recursive functions expressible in finite algorithms. However, it also inevitably brings the possibility that a query on a recursive function may never terminate. With the expression,

```
[//] fac 1.1;
```
for example, the evaluation process will never terminate and the system will exit without a result after running out of memory. The cause was that the argument `1.1` provided was not in the expected domain non-negative integers of the factorial function. Similarly the system on the expression:
```
[//] path A B;
```
will never terminate because mathematically there are a infinite number of paths between `A` and `B`: `A B, A B A, A B A B, ….`

Non termination is naturally with the Turing-machine, and it cannot be avoided unless users paid attentions in software development to keep the Froglingo-defined domain of a recursive function consistent with its expected domain. For example, the function `fac` in Froglingo should have been defined to have the range of its variable that accepts non-negative integers only. The function `path`, as another example, should never been used for a directed graph with circular links. To take arbitrary directed graphs as input, the function `path` itself should be redefined, say "the shortest path between X and Y", rather than "all the paths between X and Y".

4.1.5 Update Operation in Assignment

Froglingo allows update operations as assigners in database. For example:
```
[//] create add_a_value  = (create a_value = 6);
```
Allowing update operations in assignments become particularly useful when variables and sequential terms, as to be discussed in 4.2, are part of the assignments. For example:
```
[//] create index = 1;
[//] create objects;
[//] create add_an_element $val  = (create objects index = $val),
                                   (update index = index + 1);
```
Since an update operation becomes an assigner, it is required to return a value. When an update operation is executed successfully, it returns a special identifier `void`. While users are able to express the identifier `void`, it is never displayed as an output. When an update operation fails to terminate successfully, it returns a special identifier `error`. Similarly the identifier `error` can be expressed by users as input. However, it is not displayed as it is, but the textual messages from the system that reflects the error condition. Reference Chapter 9 for more information about the identifiers `void` and `error`.

4.1.6 Variable and Database Update

Once a variable is declared in database, it cannot be reference directly. For example:
```
[//] update fac $x;
The variable $x cannot be referenced directly.
```
To be able to modify assignments with variables, variables can be referenced in update operations. For example:

```
[//] update fac $x = ($x + (fac ($x - 1)));
[//] delete fun $x 1 $y;
[//] print fun;
fun $x 2 $y = ($x * $y);
```

When an existing assignee having variables is to be updated via `update` or `delete`, the assignee is expressed without spelling out the range.

The range of a variable cannot be updated by using `update`. To do so, we need to delete variable first and create it again with the desired new range.

4.1.7 Evaluation Rules for Variables

Now let's start with a more formal discussion on how exactly system react to query expressions against functions with infinite properties. For the readers who are not interested, the remaining part of this section can be skipped.

Given a list of variables v_0, …, v_n, and a list of terms as values V_0, …, V_n, here $n >= 0$, we call the form:

$[v_0 := V_0, …, v_n := V_n]$

an environment.

Given a term **P** and an environment $[v_0 := V_0, …, v_n := V_n]$, we can obtain another term **P′** such that each instance of v_i appeared in **P** is substituted with V_i. The substitution is done for each **i** that is ranged from 0 to n. We use the form:

$P: [v_0 := V_0, …, v_n := V_n]$

to represent **P′** and call it the substitution of **P** under the environment: $[v_0 := V_0, …, v_n := V_n]$.

Given an assignee **M** without an assigner, in which there are variables $V_0, …, v_n$, here $n >= 0$, as sub-terms, and given a list of values V_0, …, V_n from the corresponding ranges of the variables, then we define the substitution:

M: [v_0:= V_0,…, v_n:= V_n]
to be the normal form of M under the environment [v_0:= V_0,…, v_n:= V_n].

Given a substitution M: [v_0:= V_0,…, v_n:= V_n], here none of the variables v_0,…, v_n appears in M, then the normal form of the substitution is the normal form of M itself.

Now let's extend the reduction rules in Section 9 and give a comprehensive algorithm of reducing a term M to its normal form under a database that may have variables defined. Given a term M, here are the steps of reducing it to its normal form:
1. If M is a constant, M itself is its normal form.
2. If M is an identifier,
   - 2.1 If M is not defined in database, its normal form is `null`;
   - 2.2 If M has no assigner, M itself is its normal form;
   - 2.3 If M has an assigner, its normal form is the normal form of its assigner.

3. If M is a comb-term $P\ Q$, Evaluate P and Q separately to obtain their normal forms P' and Q', where P' is the substitution of P under an environment **ENV**, and **ENV** may have 0 or at least one pair of variable and value,
   - 3.1 If P' is a constant, then the normal form of M is `null`;
   - 3.2 If P' is a term in database, find the list of non-variable terms N_0, …, N_k, here k is 0 or a positive integer, such that $P\ N_0$, …, $P\ N_k$ are in database,
     - 3.2.1 If there is a N_i such that N_i == Q', here i is a number between 0 and k, then the normal form of M is the normal form of $P\ N_i$:**ENV'**.
     - 3.2.2 If there is not a N_i such that N_i == Q', here i is a number between 0 and k, find the list of variable terms v_0, …, v_l, here l is 0 or a positive integer, such that $P'\ v_0$, …, $P'\ v_l$ are in database,
       - 3.2.2.1 If there is a v_j, here j is a number between 0 and l, such that Q' falls into its range, then the new environment **ENV'** is the union of **ENV** and [v_j:=Q'],
         - 3.2.2.1.1 If $P'\ v_j$ has an assigner U, then the normal form of M is the normal form of the <u>substitution</u> U:**ENV'**.
         - 3.2.2.1.2 If $P'\ v_j$ doesn't have an assigner, then M:**ENV'** is the normal form of M.
       - 3.2.2.2 If there is not a v_j, here j is a number between 0 and l, such that Q' falls into its range, then M has the normal form `null`.

When M ends up with an environment as a part of its normal form, then the system reports it as an error. For examples:
```
[//] fun 2;
There are not sufficient arguments provided to complete
evaluation.
```

4.2 Sequential Terms

Allowing a sequence of terms as an assigner is to express multiple actions that are triggered by a single event. For example, when a purchase order is to be closed, multiple operations have to be executed: reduce storage volume, generate shipping report, verify credit card, and deposit money.

A <u>sequential term</u> is a sequence of terms separated by commas ‘,’. Sequential terms can only serve as assigners. For example,

```
[//] create account1 = 100;
[//] create account2 = 300;
[//] create transfer $money =
 (update account2 = (account2 - $money)),
 (update account1 = (account1 + $money));
[//] transfer 10;
[//] account1;
110;
[//] account2;
290;
[//] create ack_after_action $m = transfer $m,
                "The amount ",
                $m,
                " has been transferred";
[//] ack_after_action 30;
"The amount "30" has been transferred";
[//] account1;
140;
[//] account2;
260;
```

Before ending this section, we give a formal discussion on how the system reacts on sequential terms. For those who are not interested, the remaining material of this section can be skipped.

Given an assignment $\mathbf{M} = \mathbf{N}_1, \mathbf{N}_2, \ldots, \mathbf{N}_n$, its substitution under an environment $[\mathbf{v}_0 := \mathbf{V}_0;\ \ldots,\ \mathbf{v}_n := \mathbf{V}_n]$, syntactically $\mathbf{N}_1, \mathbf{N}_2, \ldots, \mathbf{N}_n$: $[\mathbf{v}_0 := \mathbf{V}_0;\ \ldots,\ \mathbf{v}_n := \mathbf{V}_n]$, is the sequence of the following sub substitutions:

$\mathbf{N}_1$: $[\mathbf{v}_0 := \mathbf{V}_0;\ \ldots,\ \mathbf{v}_n := \mathbf{V}_n]$,

…

$\mathbf{N}_n$: $[\mathbf{v}_0 := \mathbf{V}_0;\ \ldots,\ \mathbf{v}_n := \mathbf{V}_n]$.

The normal form of the substitution ($\mathbf{N}_1, \mathbf{N}_2, \ldots, \mathbf{N}_n$): $\mathbf{v}_0 := \mathbf{V}_0;\ \ldots,\ \mathbf{v}_n := \mathbf{V}_n$ is the sequence of the normal forms of the sub substitutions.

Chapter 5: INFORMATION COMMUNITY

A community in our daily life is a geographical area in which a group of people live together. In the community, each person has a family and each family has a private space. A community has a set of rules followed by the citizens such that they collaborate to enrich the community. By information community, we mean that Froglingo provides a computing environment for multiple business owners and their customers. They construct and share information inside and outside of their organizations. They collaborate and construct partnerships among them.

Froglingo offers private spaces for multiple business owners and individual users. It offers data access control mechanism such that people can share information and collaborate. By setting Froglingo as a website, people can use web browsers to manage and share their data including files across the Internet. Users can change their stands (called directory or folders in file management systems) when they are constructing or viewing data. It allows a user to manage multiple applications separately. More importantly, one user can construct his/her own data on the base of other users' data.

5.1 User Account

A user account is actually an assignee in database. More than an assignee, it has additional associated information such as a password, the created date, and more.

There are two built-in user accounts: `root` and `anyone`. The account `root` (also expressed as `//`) is the root of an entire database. In other words, the root is functionally determining all the assignees in a database. Given an assignee **A**, it is always true that **A** `{=+ //`. The account `anyone` is a term under the root, that is, `anyone {+ //`.

We have being seen that all the sample command lines appeared in the earlier sections were headed with:

```
[//]
```

It hinted that a database is always started with the account `root`. To add more user accounts, the account `root` must be provisioned with a password first:

```
[//] passwd;
User Id: //
New password: *******
Confirm password: ********
[//]
```

Now Froglingo is ready to create additional user accounts, which is done by using the command `addusr`. The form is:

```
addusr userID;
```

Here **`userID`** is an identifier uniquely representing a user. For example:

```
[//] addusr jason;
The password is: QiRW$5N8
[//] addusr www.myclienta.com;
The password is: i8u9i1o@
```

Now the user accounts are ready for users to use. To quite from Froglingo environment:
```
[//] quit;
Thank you for using Froglingo!
```

Then come back to Froglingo again as the user account `www.myclienta.com` through CMD window:
```
C:\Froglingo\frog.exe
User id: www.myclienta.com
Passwd: ********
```

Change the password for `www.myclienta.com`:
```
[//www.myclienta.com] passwd;
```

By following the instruction after the command `passwd`, the password can be changed.

Now the user `www.myclienta.com` can create its own user accounts:
```
[//www.myclienta.com] addusr jone.dow;
The password is: oI45E0@a
```

To facilitate the management of user accounts, we introduce additional commands in the rest of the section. The readers who are not interested in the subject at this moment can move to next section.

A user account can be suspended or deleted by super users: The formats are:
```
sususr userID
delusr userID
```

When a user account is suspended, no one can use the user account to logon to system. But all the data related to the user account still exists in database. To activate a suspended account, a super user needs to use the command:
```
actiusr userID
```

When a user account is deleted from database, all the data related to the user account are removed. The command `delusr` is equivalent to the command `delete`.

Earlier, we have already demonstrated how a user changes its own password by the command `passwd` alone. At that time, the user had to provide the old password before a new password was requested. In the case that a user forgot its password, the super user can reset the password without providing the old password. Here is the syntax:
```
passwd useraccount;
```

For example:
```
[//www.myclienta.com] passwd jone.dow;
```

5.2 Session

A session is the period between the time when a user logins via a user account and the time when the user logouts.

5.2.1 Establishment

There are many ways to start a session. The first one is when a new database starts. By invoking the executable `frog.exe` at Operating System level, a user enters into a session even without providing a password. We have done so in the section 1.6.

The second method is to start a session after a user exits from a previous session. The user is prompted to provide a user id and a password after invoking the executable frog.exe in a CMD window. After successfully logging into the database by providing the correct password for a user account, a session starts. An example has been provided in Section 5.1.

The third method is to enter the following command in an earlier session:

```
login term "passwd";
```

Here **passwd** is displayed in plain text and must be surrounded by a pair of double quotes; and **term** is a user account. For example:

```
[//] login www.myclienta.com "i8u9i1o@";
[//www.myclienta.com]
```

When the new session starts, the earlier one will be terminated automatically. This method is typically useful when a user attempts to login through a web browser. We will discuss more about it in Chapter 7.

5.2.2 Signature

A session is always associated with two parameters. The first one is its user account, representing the user who is interacting with the system. Through the user account, the system knows the privileges the user was assigned with; and will either grant or reject the requests from the user depending on the privileges. The topic of privileges will be discussed in a later section. Whenever a session is started, the user account is echoed by using the special identifier `signature`. For example, a user via the user account `//` in a session can have the following interactions:

```
[//] signature;
//;
[//] signature John salary;
2000;
```

Since the term `signature` reflects a system value, and not editable by anybody, it serves as the <u>signature</u> or footprint of the user when the user is interacting with other users. We will talk about it in a later section.

To logout from the current session, users call the command `quit`. We have used it earlier.

5.2.3 Stand

The second parameter associated with a session is its <u>stand</u>, a "position" where the user stands at in a database at a given time. It actually is an assignee that is associated with the session. When a session starts, the stand is always the user account itself. When a user logins with the user account `www.myclienta.com`, as an example, the user of the session is `www.myclienta.com`; and the stand is `www.myclienta.com` too.

In a session, the stand changes when the user navigates data in database via dependent relationships by using the command `cd`. The stand is echoed by using the command `whereami`. At a different stand, a user will view data differently and is required to express the data differently as we will see it in a later section. We will dedicate the next section to discuss how users navigate database.

5.3 Traveling and Naming

A Froglingo database is intended to host multiple business owners, where each business owner may have multiple applications. To effectively manage data and share data with others, a user may want to "travel" through database. When user travels, entities appear differently to user, and user is required to express entities differently. The key to accomplish traversal through database is on how to name entities. This section gives the command `cd` that changes session stands, a set of common stands users normally go, and the naming scheme that makes traversal possible.

When a mom speaks to her own kids and says "Michelle", the name Michelle is always unambiguously referencing a specific kid. But when she speaks to a group outside of her family, she may have to say "my Michelle" to reference her own daughter Michelle. In another scenario, a residential community manager may have to say "Mike on 22 High Street" to reference a specific person in her community. On the other hands, she only needs to say "Mike" when she is visiting the house on 22 High Street. The scenarios tell us two common senses in our daily life:

1. The expression referencing an entity in the world may have to be changed depending on the circumstance in which the speaker talks.
2. The expression referencing an entity may not have to be changed at a given circumstance no matter who is the speaker.

We apply the two common senses to Froglingo. Similar to a location or circumstance in the life scenarios given above, the stand in a session provides the user a datum from which expressions are given to reference entities in database. An expression being uniquely referencing a database entity is called a name of the entity. An entity may have many names. The assignee (entity) itself is a name. All the terms that can be reduced to the same normal form can serve as names for the entity too.

5.3.1 Moving Stand

Users can use the command `cd` to move user's stand from one to another. The syntax is:

```
[stand] cd term;
```

Here **`stand`** is the current user's stand; and **`term`** is the new stand the user moves to. The new stand can be an assignee in database as long as the user has a privilege to do so (we will discuss it in next section).

To demonstrate this, we assume that a user wants to construct two independent applications and to store a set of files (to be discussed in Chapter 6). Then the user creates three terms serving as sub folders[1]:

```
[//www.myclienta.com] create appl1;
[//www.myclienta.com] create appl2;
[//www.myclienta.com] create files;
```

Then the user can change her stand to one of the subordinates; and do some work:

```
[//www.myclienta.com] cd appl2;
[//www.myclienta.com appl2] create a b (c d) = 8;
[//www.myclienta.com appl2] print .;
```

[1] Here terms appl1, appl2, and files are similar to folders in operating systems.

```
a b (c d) = 8;
```

The stand is always displayed as the header of each command line.

Constructing an application under a separated stand is important because it isolates one application from others, and makes the development and maintenance easier. Also an independent application can be migrated from one place to another by using the download and upload commands `print` and `load` discussed in Chapter 6.

In addition to travel inside its own space, a user is allowed to travel outside as long as the permission is set to allow doing so. For example, any user can travel to the user account `anyone`:

```
[//www.myclienta.com] cd // anyone;
[//anyone]
```

5.3.2 Current and Upper Stand

The stand of a session at a given time is called the current stand, and we use the special character '.' to represent it. Continuing the examples in Section 18.1, here are some examples:

```
[//www.myclienta.com appl2] .;
www.myclienta.com appl2;
[//www.myclienta.com appl2] . a b (c d);
8;
```

The assignee above the current stand is called the upper stand. In other words, if the current stand is **`M`**, then the upper stand is `pterm` **`M`**. We use the special string "`..`" to represent it:

```
[//www.myclienta.com appl2] ..;
www.myclienta.com;
[//www.myclienta.com appl2] .. appl2;
www.myclienta.com appl2;
```

5.3.3 User Home

The user home of a session is the user account itself. We use the special symbol '~' to represent it. For example:

```
[//www.myclienta.com appl2] ~;
www.myclienta.com
[//www.myclienta.com appl2] cd ~;
[//www.myclienta.com]
```

5.3.4 Application Home

When a user created data on a stand, the stand became an application home. The examples are `//`, `anyone`, `www.myclienta.com`, and `www.myclienta.com appl2`. The assignees `a` and `a b` were created under the stand `www.myclienta.com appl2`, and they are not application homes because no user yet took them as stands to create data. The sample terms such as `college`, `college admin`, and `Mike` given in Chapter 2 are not application home either. A user home must be an application home.

Assume that a user created an assignee $\mathbf{A}_1$… $\mathbf{A}_n$ under an application home **`S D`**, here **`S`** is

also an application home. When the user changes the stand to **S**, then the $A_1 \ldots A_n$ under the stand **S** appears to be the expression $D\ A_1 \ldots A_n$, or equivalently $(\ldots(D\ A_1) \ldots A_n)$. For examples:

```
[//www.myclienta.com appl2] print .;
a b (c d) = 9;
[//www.myclienta.com appl2] cd ..;
[//www.myclienta.com] print app2;
appl2 a b (c d) = appl2 9;
[//www.myclienta.com] app2 a b (c d);
appl2 9;
```

Keeping an assignee a consistent view when a user moves from one application home to another is important in data construction, in data sharing, and data collaborating.

5.3.5 Non-Application Home

If a user created an entity $A_1 \ldots A_n$ at the stand **S**, then the terms $S\ A_1$, $S\ A_1\ A_2$, …, $S\ A_1\ A_2 \ldots A_n$ are not application homes. For example `www.myclienta.com appl2 a` and `college admin` under `//` are non-application homes. Users are allowed to step into non-application homes by using `cd`. However Froglingo prevents users from creating new data at non-application homes. Therefore a non-application home cannot become an application home.

5.3.6 Absolute Name

The names we discussed so far are relative. An entity in database has to be named differently when user changes stand from one to another. Here we utilize the root "//" when referencing an entity. It leads absolute names, which do not change when users travel.

A term having "//" as the inner-most term is an absolute name. Therefore, one without "//" as the inner-most term is a relative name.

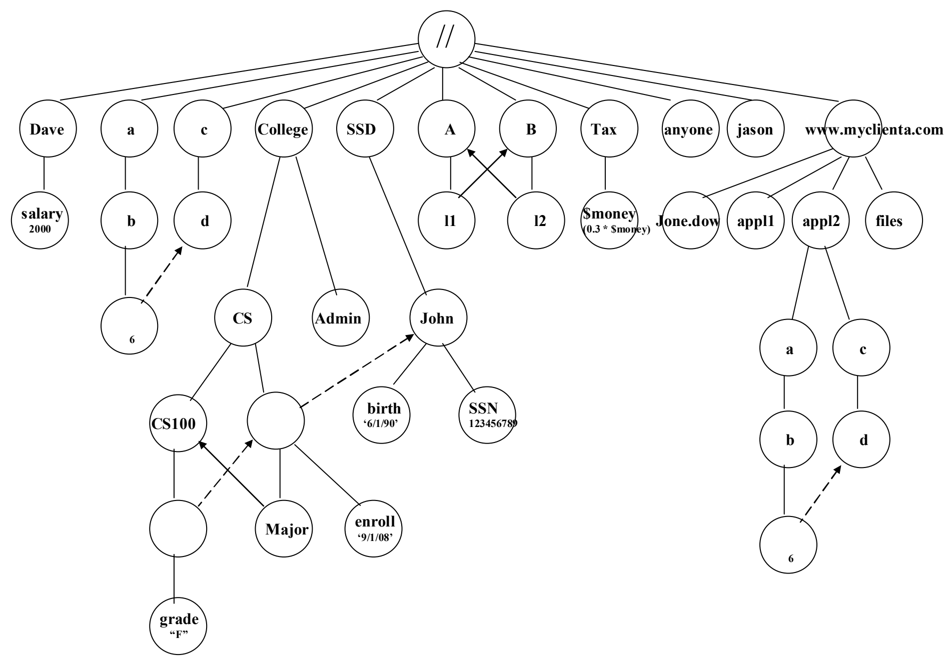
Given an assignee $A_1 \ldots A_j$ under an application home $S_0 \ldots S_i$ as the stand, the absolute name of the assignee is $//S_0 \ldots S_i\ A_1 \ldots A_j$, or equivalently $(\ldots\ ((\ldots\ (//S_0)\ \ldots\ S_i)A_1) \ldots A_j)$. Here are a few examples:

```
[//] www.myclient.a.com appl2 a b (c d);
www.myclienta.com appl2 9;
[//] // www.myclient.a.com appl2 a b (c d);
www.myclienta.com appl2 9;
[//www.myclienta.com appl2] a b (c d);
9;
[//www.myclienta.com appl2] // www.myclient.a.com appl2 a b (c d);
9;
[//www.myclienta.com appl2] // SSD John SSN;
// 123456789;
```

Stands are always displayed as absolute names.

To assist a better understanding on database traveling and naming, we give a graphical view of some frequently used sample data we created through this document so far. The graphical view also demonstrates how Froglingo system uniformally manage data, user

accounts, business logic, and files.



5.4 Privilege

In Froglingo, there are only two kinds of privileges: administration and access. The owner and administrators in an organization, in the real world, manage the entire organization, i.e., define what the organization is; and offer methods through which other people are able to access the organization either entirely or partially. Accessing an organization, in the real world, means not only obtaining knowledge about the organization, but also enriching the organization according to the access methods offered by the administrators. The administration privilege and the access privilege in Froglingo adopt the common sense in the real world.

Assume that a user creates the following data in his/her own space:

```
[//www.myclienta.com] create AA BB CC = 5;
[//www.myclienta.com] create AA BB DD = 6;
[//www.myclienta.com] create AA private CC = 7;
```

Now the user wants to make the terms below `AA BB` readable to anyone; and to make her employee `jone.dow` to have the full control over `A B` and below. Then the user can make the following specifications:

```
[//www.myclienta.com] grtacc (AA BB) anyone;
[//www.myclienta.com] grtadm (AA BB) jone.dow;
```

To test the results, she can login as an `anyone` user (note that the password for the `anyone` user is "anyone" always):

```
User Id: anyone
Password: ******
[//anyone] //www.myclienta.com AA BB CC;
```

```
//www.myclienta.com 5;
[//anyone] //www.myclienta.com AA private CC;
The term may not be accessible or not exists.
```

The output above demonstrates the effectiveness of the command `grtacc`. The rest of this section formally discusses the two types of access privileges: administration and access privileges.

5.4.1 Administration

When a user has the administration privilege on an entity, the user has full control over the entity and the entities below. In other words, if the entity is **M**, then all the terms **N**, such that **N** {=+ **M**, are subject to the administration privilege of the user. The administration privilege is meant that a user can perform all the operations available in Froglingo. These operations include create, record, update, delete, query operations and the user account management operations discussed in this Chapter.

By default, the user possessing a user account has the administration privilege over the entities below the user account. The user account is called the owner of the entities below it. In other words, If **U** is a user account, and **M** is an entity such that **U** {=+ **M** and **U** != **M** , then **U** is the owner of **M** and **U** has the administration privilege over **M** .

By default, if **U** is a user account, and the entity **M** is not one below **U**, that is, **U** {=+ **M** and **U** != **M** doesn't hold true, then **U** has no privilege at all over **M**. To assign **U** to have the administration privilege over **M**, the operation `grtadm` has to be used. The form is:
`grtadm` **M U**

Here **U** must be a single user account. This operation can be performed only by an owner of **M**.

5.4.2 Access

When a user has the access privilege on an entity, the user can do every thing except for create, record, update, and delete operations and the user account administration operations over the entity and the entities below. In other words, if the entity is **M**, then all the terms **N** such that **N** {=+ **M** are subject to the access privilege from the user.

When a user has the administration privilege over an entity, it automatically has the access privilege over the entity. Therefore, a user has the access privilege over the entities below his/her user account.

As stated in section 5.4.1, if **U** is a user account, and the entity **M** is not one below **U**, that is, **U** {=+ **M** and **U** != **M** doesn't hold true, then **U** has no privilege at all over **M**. To assign **U** to have the access only privilege over **M**, the operation `grtacc` has to be used. The form is:
`grtacc` **M users**

Here **users** is **U** itself, or a variable definition with a range such that **U** falls into the range of the variable. If **users** is a variable, then the command allows all the users that fall into the range of the variable to have the access privilege against **M**. For example:

```
[//www.myclienta.com] create AA Associate data = 32;
[//www.myclienta.com] grtacc (AA Associate)
```

```
$x:[$x fname == $x "Jone"];
```

The user accounts under the account `www.myclienta.com` whose first names are "Jone" were granted with the access privilege on `AA Associate`. The example assumed that that each user account under `[//www.myclienta.com]` had its first name as its property. For example:

```
[//www.myclienta.com jone.dow] print .;
fname = "Jone";
lname = "Dow";
```

With the permission set, `jone.dow` is able to view the data owned by its super user:

```
[//www.myclienta.com jone.dow] .. AA Associate data;
//www.myclienta.com 32;
```

The entities accessible to other users due to `grtacc` actually become interfaces. Not only for the purpose of viewing data, the interfaces allow users to trigger update operations. Sections 5.4.4 and 5.4.5 cover the detailed information on the effect of the command `grtacc`.

When an entity is not a user account and falls into the range of the variable definition **`users`**, the entity is not affected by the command `grtacc`.

5.4.3 Privilege Removal

There are two other commands in managing privileges. The first one is to display privileges. Given an entity **`M`**, the command:

```
echpriv M
```

displays the privileges assigned to all the assignees **`N`** such that **`N`** `{=+` **`M`** or **`N`** `}=+` **`M`**. Only the owner of **`M`** is able to perform this command. For example,

```
[//www.myclienta.com] echpriv AA;
//grpRoot   (//www.myclienta.com   AA   (//www.myclienta.com   BB))
$any:[($any {=+ //)) = false;
//grpRoot    (www.myclienta.com    AA    (www.myclienta.com    BB))
(//www.myclienta.com jone.dow) = true;
```

In the example, we see that the value `true` of an entry represents the administration privilege, and the value `false` the access privilege; and that the `anyone` user assigned with access permission was automatically converted to a variable having all the user accounts of database in the range. At this release, the report of privilege summary is rough, and may be improved in the future. But the key point here is that users can have a view of the privileges.

The second command is to remove privileges. The removable privileges are those added by using `grtadm` and `grtacc`. The privileges that are assigned as default cannot be removed. The format for privilege removal is:

```
rempriv entity users;
```

Here **`entity`** is the entity against which an administration or access privilege was assigned before; and **`users`** is the user or the variable representing a group of users who obtained the privilege. To ensure a successful removal, the parameters **`entity`** and **`users`** must be provided correctly to match an entry in the privilege list. When an

attempt is succeeded, an entire entry will be removed. Otherwise, no action will be taken and an error message will be returned. Again only the owner of **entity** is able to perform this command.

5.4.4 Interface

When an entity is granted a user with the access privilege, we call the entity an interface. An interface not only offers a way of retrieving information; and it can also cause data to be updated. The update can occur even if the user who triggered the action has no administrative privilege on the data. To demonstrate this, we give a sample application in simulating a lottery game: A privately held number would not be increased by 1 unless an `anyone` user provided the correct lottery number (e.g., 9242922).

```
[//www.myclienta.com] create number = 1;
[//www.myclienta.com] create analysis true =
                 (update number = (number + 1)),
                 "Congratulations! You won the lottery.";
[//www.myclienta.com] create analysis false =
                 "Sorry, you didn't win the lottery";
[//www.myclienta.com] create receive_request $num =
                 analysis ($num == 9242922);
[//www.myclienta.com] grtacc receive_request anyone;
```

Now an `anyone` user will be able to verify his/her lottery number by the expression:

```
[//anyone] //www.myclienta.com receive_request 2339999;
//www.myclienta.com "Sorry, you didn't win the lottery.";
[//anyone] //www.myclienta.com receive_request 9242922;
//www.myclienta.com "Congratulations! You won the lottery.";
```

In the commands above, the term `number` was kept not accessible at all to `anyone` user, but it was updated. The Froglingo system did it by switching the requester from `anyone` user to the owner `www.myclienta.com` of `number` when the assigner of the assignee `receive_request` is being evaluated. It was permitted to be evaluated because the assignee `receive_request` was granted `anyone` user with the access privilege (and therefore was called an interface).

Switching users during execution is the built-in mechanism allowing users to share information and to collaborate.

5.4.5 Partnership

Through an interface, a user can trigger a comb-term to be created such that the plus-term and the minus-term are not owned by the same user account. We call such a comb-term a partnership. Precisely, given two entities M_1, owned by O_1, and M_2, owned by O_2 in a database, we call the third entity M_1 M_2 in the database a partnership between O_1 and O_2. Further we call that O_1 is the plus-owner and O_2 the minus-owner of the partnership.

A partnership is owned by the owner of the plus-term. Therefore the owner has the full control over the partnership. A partnership is fully private except that the minus-owner automatically obtains the access privilege on the partnership.

As an example, the business owner `www.myclienta.com` allows anyone to create a

piece of data by providing the following interface:

```
[//www.myclienta.com] create add_trash $value = (create trash_can
$value date = timestamp);
[//www.myclienta.com] grtacc add_trash anyone;
```

Then a user, via the `anyone` user account, can call the method `trash_can`:

```
[//anyone] //www.myclienta.com add_trash .;
```

The database is added with the following entities:

```
[//www.myclienta.com] print trash_can;
trash_can (// anyone) date = 1243877000;
```

It is not accessible to anyone else except for the plus-owner and the minus-owner:

```
[//anyone] //www.myclienta.com trash_can . date;
//www.myclienta.com 1243877000;
```

The interface offered above is one of the simplest forms. The plus-owner even doesn't know who came and created the data. To impose a stronger responsibility between the two owners of a partnership, the plus-owner may require the minus-owner to acknowledge the establishment of the partnership by passing its signature to the interface. A club, as an example, accepts new memberships. The only requirement for a person to become a member is to accept an agreement and to provide his/her signature. Assume the owner `www.myclienta.com` establishes this club:

```
[//www.myclienta.com] create club;
[//www.myclienta.com] cd club;
[//www.myclienta.com club] create accept_mem
      $who:[$who isa signature] true =
      (create members $who enroll_date = timestamp);
[//www.myclienta.com club] create accept_mem $who false =
      "You are not allowed to join the club because you didn't
accept the agreement";
[//www.myclienta.com club] grtacc accept_mem anyone;
```

The user `jason` joins the club by calling the function `accept_mem` with his agreement (via the value `true`) and his signature:

```
[//jason]//www.myclienta.com club accept_mem signature true;
```

Now, the new entry is viewable to both the plus-owner and the minus-owner:

```
[//www.myclienta.com club] print members;
. (// dave) enroll_date = 1243878771;
[//dave]print // www.myclienta.com club members .;
enroll_date = 1243878771;
```

In the real life, a partnership needs to be temporarily suspended. A member in the sample club above, i.e., `//www.myclienta.com club members (// jason)`, needs to be suspended for some reasons. To support this scenario, Froglingo provides two facilities:

`susparti` **`partnership`**

It can be performed by the plus-owner only. When a partnership is suspended, the minus-owner will no longer be able to access the partnership. To re-activate the participation, the facility `susparti` can be called by the plus-owner:

`actiparti` **`partnership`**

To permanently remove a partnership, either the plus-owner or the minus-owner can use

the `delete` command.

Chapter 6: FILE MANAGEMENT

When we talk about information today, file is always a part of it. Froglingo manages theree types of files:

1. HTML/XML files, possibly embedding the expressions to retrieve data from database. We need to compose them at operating system level with an editor; upload them to Froglingo such that they can be rendered to web pages in conjunction with Froglingo database.
2. Froglingo data files, containing a list of Froglingo expressions (mostly assignments). We may compose them at operating system level with an editor; upload them to Froglingo.
3. Binary files, all the files except for HTML/XML files and Froglingo data files. They could be image, audio, and other textual files. Froglingo treats them all as binary files.

Users can use the commands introduced in this Chapter to transfer files between Froglingo, operating system, and web browser.

A file has a name and its content. The name is an identifier. The examples are: `textfile`, `myresume.doc`, `myphoto.jpg`, `store.html`, `data.xml`, `backupfile.frog`. The content is a stream of ASCII codes. Note that a file name at operating system level may contains special characters space ‘ ‘ and dash ‘-‘. User needs to convert those file names at OS level to Froglingo identifiers before uploading.

A HTML file having the suffix `.html` has its content structured according to the HyperText Marker Language. It is rendered as a web page by a web browser. Many of HTML files today contain errors and do not exactly follow the Hyper Text Marker Language. Web browsers, on the other hand, accept those files with errors by rendering them as much as possible.

A XML file normally having the suffix `.xml` is one with its content structured according to the Extensible Marker Language. XML language is a more generic than HTML language. In other words, a HTML file should be a XML file. In practice, XML parsers place more restricted grammar rules such that a XML file cannot be accepted unless it more closely follows the XML language grammars. Therefore, a HTML file is more likely not being accepted as a XML file.

Froglingo treats both XML and HTML files equally. Given a HTML/XML file, Froglingo checks and reports possible errors. At the same time, it stores the file content anyway regardless if it contains error or not.

In a Frogingo system, a XML/HTML file can embed Froglingo expressions, and to accept parameters in Froglingo. We will discuss this topic in Chapter 7.

A Froglingo data file, normally having the suffix `.frog`, is one with its content structured according to Froglingo expressions. It is a list of assignments. When a Froglingo data file is uploaded, the list of assignments is constructed in database.

6.1 Upload

A file or an entire folder at operating system level can be uploaded to database. The command format of uploading files is:

```
(load path [in format])
```

The parameter **`path`** is the path of a file or a folder at operating system level. The optional parameter **`format`** tells the system how the file or a folder should be parsed. It takes one of the following values and acts as specified:

Format	Description
`stream`	A binary stream in US ASCII codes. The system will load the file as it is.
`frog`	A list of Froglingo assignments. The system will use the command `create` to load each assignment.
`backup`	A list of Froglingo assignments. The system will use the command `record` to load each assignment.
`Xml`	A XML document. The system will parse it as a XML file, report any syntactical errors. If it doesn't embed any Froglingo expressions, the system stores it as it is no matter if it has error or not. If it does contain Froglingo expression, the system will reject it if a syntactical error is found.
`Html`	A HTML document. The system treats it as a XML document.
`folder`	A folder at Windows level. The system walks the folder recursively and loads the entire folder.

If the parameter **`format`** is not specified, the system detects the format from the path **`path`**. If the path references a file, the type of the file is derived from the suffix of the file name:

```
.htlm - html
.xml - xml
.frog - frog
```

For the files with other suffixes or without suffix, the system takes `stream` as the default format.

Assume that the folder `C:\\Froglingo` has the following files and a sub folder:

```
frog.exe
test1.txt
test2.frog
```

The file `test1.txt` has the content:

```
This is a short text file.
```

The file `test2.frog` has the content:

```
loaded_term x = 3;
loaded_term y = 4;
```

One can load files in Froglingo environment:

```
[//] load test1.txt;
```

```
[//] load test2.frog;
```

The operation of uploading a `stream` file is a term too, and semantically equivalent to a create operation that takes the file name as the assignee and the content of the file as the assigner. After the upload operation of the file `text.txt` above, you can view the data is available under `www.myclienta.com`:

```
[//] print .;
Loaded_term x = 3;
Loaded_term y = 4;
test1.txt = This is a short text file.
```

If the parameter **`path`** is a folder, the system walks through the files and the sub folders under **`path`** and loads the entire folder. Assume that the folder `C:\\Froglingo` has a sub folder `images`. Under the folder `images`, there are files `image1.jpg` and `file.pdf`. One can load the entire sub folder `images`:

```
[//] load images;
[//] ls .;
images file.pdf;
images image1.jpg;
loaded_term x;
loaded_term y;
test1.txt;
```

In the previous examples, we used two utilizes: `print` and `ls`. The command `print` prints all the assignments under the given entity. The command `ls` is used to list assignees only. When there are files with a large volume of contents under a Froglingo entity, we should use `ls` only to browse the dependent entities under the given entity; and use `print` when a file needs to be transferred to a web browser or downloaded to operating system level. More discussion about the two utilizes is given in Section 6.2.

More about uploading XML/HTML files is to be discussed in Chapter 4. Uploading files via web browser is done via the same operator `load`, also discussed in Chapter 4.

6.2 Download

Files uploaded to database will eventually be shared with other people. A file may be downloaded again back to operating system level; displayed as an image; or rendered as a web page. To manage database, a set of assignments in database can be backed up to a file at operating system level. All of these operations are performed by using the following syntax:

```
print term [to path] [in format]
```

Given an assignee **`term`**, all the assignees that are functionally dependent on **`term`** will be downloaded from database. When the optional parameter (`to` **`path`**) is provided, it is the path of a file to be generated at operating system level. If the parameter (`to` **`path`**) is not provided, the data related to **`term`** will be displayed on CMD windows or on web browser depending on where requests come from. When requests come from a web browser, this parameter (`to` **`path`**) should never be provided. Otherwise, the system will give users an error message.

The optional parameter **`format`** specifies the format of the output, and tells the system

how the output should be generated. It takes one of the following values and reacts as specified:

Format	Description
`stream`	It is only for a term **`term`** which is the file name of an ASCII stream file. The system will simply echo back the ASCII stream file. Otherwise, the system will return an error message.
`frog`	The system will report all the assignments that have assigners, and the assignees are functionally dependent on **`term`**. All the strings as sub terms in the assignments are printed out without double-quotes surrounding them.
`backup`	It works as if it was the format `frog`. The only difference is that all the strings as sub terms in the assignments are printed out with double-quotes surrounding them.
`xml`	The system will intend to generate a XML document according to the parameter **`term`**. When **`term`** is the file name of a file uploaded before and the file has no Froglingo expressions embedded, it will simply echo back the file. When **`term`** is a term against an uploaded file having Froglingo embedment, the system will generate a XML document having contents from database. Unit 4 is dedicated for XML generation.
`html`	The system will act as if the format was html.

When **`term`** is the file name of a file uploaded before, the parameter **`format`** may not have to be specified. In this case, the format is derived from the suffix of the file name:

```
.htlm - html
.xml - xml
.frog - frog
```

For the files with other suffixes or without suffix, the system takes `frog` as the file format.

When there are files with large size, one should avoid using the command "`print .`" that would take a long time to print out the contents of all the files. Printing out one file at a time is more appropriate. To have an overview of the assignments under the current stand, one may use another command "`ls .`" that prints out the list of assignees without the assigners.

6.3 OS Path

In sections 6.1 and 6.2, there was not a precise format defined for the parameter **`path`** at Windows level, but assumed that **`path`** was a file or a folder name appearing as an identifier located in the same folder where the Froglingo process `frog.exe` was launched. Actually, a file or folder not in the same folder of `frog.exe` can also be uploaded from or downloaded to. Recall that a path in Windows can be a file name alone, a sequence of file/folder names delimited by either "/" or "\", or a path proceeded with a hard drive identity. In other words, the syntax of file path at Windows is:

[driver:\]$\mathbf{name_0}$****$\mathbf{name_1}$…****$\mathbf{name_n}$, or
[driver:/]$\mathbf{name_0}$**/**$\mathbf{name_1}$…**/**$\mathbf{name_n}$

Here the number **n** can be 0 or a positive integer. However, the delimiter "\"is not recognized by itself in Froglingo, or is interpreted as the escape character of special ASCII codes in a string in Froglingo. Some of the special ASCII codes are: '\0' (NULL), '\t' (tab), and '\\' (backslash). The delimiter "/" is the arithmetic operator division in Froglingo. Therefore, the acceptable syntax of file path at Froglingo is different from the one at Windows, and defined as the following:

"**[driver:****]name**$_0$**name**$_1$…**name**$_n$", or
"**[driver:**/**]name**$_0$/**name**$_1$…/**name**$_n$"

Also Froglingo restricts that a segment **name**$_i$ of a path appears as a Froglingo identifier (not space or a dash in **name**$_i$). Here are a few valid examples:

```
[www.myclienta.com] load "..\\..\\afile";
[www.myclienta.com] load "C:/folder1/folder2/folder3";
[www.myclienta.com] print test1.txt to
"C:\\folder1\\folder2\\filder3\\test1.txt";
```

When the command is `load`, the last segment **name**$_n$ in the syntax we discussed earlier can be the wildcard "*". It is semantically equivalent to:
"**[driver:**/**]name**$_0$/**name**$_1$…/**name**$_{n-1}$", i.e., all the files or folders under the folder **name**$_{n-1}$ would be uploaded. For example:

```
[www.myclienta.com] load "images/*";
```

Chapter 7: ACCESS OVER INTERNET

Froglingo system can be set to act as web server. When a system is a web server, it is also a database server; and users can communicate with the server via a web browser. There are two aims of setting Froglingo as web server:

1. The end users of applications hosted by a Froglingo server can interact with the applications via web browsers. Web browser is the only graphical user interface so far.
2. Application owners can manage the applications via web browsers. In other words, users are prompted to perform as many tasks via web browsers as those via local CMD windows (consoles). However for security reason, the root user is prohibited from accessing Froglingo server via web browser.

This chapter presents the materials on:

1. How to embed Froglingo expressions in URIs such that Froglingo server understands the requests from web browsers.
2. How to embed Froglingo expressions in HTML files such that the data from Froglingo database can be presented on web pages.

Set up web browser:

Disable favicon support in Firefox

- In the address bar, type `about:config`
- Type "icon" in the search filter on the top
- Change browser.chrome.favicons to false by double clicking on it
- Change browser.chrome.site_icons to false by double clicking on it
- Restart Firefox

7.1 Web Server Setup

To use web browser in interacting with Froglingo, We need to do two things:

1. Establish a network. There are two ways to do it:
    a. If it is only for testing purpose, the network could be the server itself, where both the application and the web browser are located in the same server. In this case, the domain name (or the user account) is required to be added to the system file:

"C:\WINDOWS\system32\drivers\etc\hosts". A sample entry would be::

```
127.0.0.1        www.myclienta.com
```

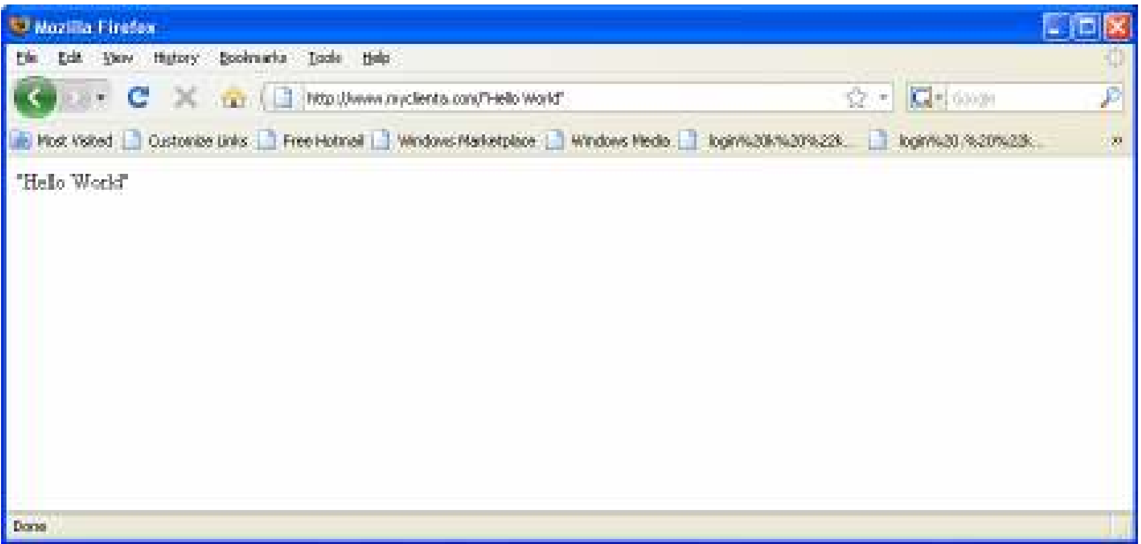
b. To make the application available on the Internet, a physical network is connected to the Internet. It can be done through an internet service provider which offers connections to business locations or to residential homes. Also it can be done by using a dedicated computer hosted by a web hosting company.

2. Set up a Froglingo system installed in a computer as a web server. It is done by entering the command at operating system level, i.e.,

```
C:\Froglingo frog.exe -p 80
The web server has been started at port 80
```

Note that a user account, such as `www.myclienta.com`, is constructed in the Froglingo database of the server. It will also serve as the Internet domain name of the application to be launched on the server. Multiple applications can be launched with a single server.

By opening a web browser, one can enter a URI, e.g., `http://www.myclienta.com/"Hello World"`, in the address field of the web browser. The web browser screen shot below captured a simple expression "Hello World":



7.2 URI

Users via web browser use URI, Universal Resource Identifier, to communicate with web browser. Froglingo parses URIs and converts them to Froglingo expressions. A simple form of the URIs used by Froglingo is:

`http://`**dn[:port][/**t_0`...`**/**t_{n-1}**/**t_n**][/]**

The field **dn,** following the string `http://`, is a registered domain name The optional field **port**, following the character ":", is the port number on which Froglingo server is listening. This field is needed only if the port is not 80. The optional fields /t_0.../t_{n-1}/t_n is a sequence of expressions, and each expression is a sequence of characters, proceeded with the special character '/'. Here each t_i is in the

correspondence of a term (possible a comb-term); and the symbol **id** in the table below is an identifier in Froglingo.

A URI, once received by a Froglingo web server, is converted to Froglingo expression. A URI from a web browser has an equivalent Froglingo term:

URI (Web Browser Request)	Froglingo Expression
http://**dn[:port]**	[//**dn**] print index.html;
http://**dn[:port]**/t_0.../t_{n-1}/t_n/	[//**dn** (t_0)... (t_{n-1}) (t_n)] print index.html;
http://**dn[:port]**/t_0.../t_{n-1}/**id.html**	[//**dn** (t_0)... (t_{n-1})] print **id.html**;
http://**dn[:port]**/t_0.../t_{n-1}/t_n	[//**dn** (t_0)... (t_{n-1})] t_n;

The first and the second two equivalences adapt the behavior of a traditional web server, which feeds the default HTML file index.html when there is no specific request given in a URI. When a URI is ended with an expression, but not '/', the last expression is interpreted as the request while the rest part of the URI proceeding the last expression is interpreted as the stand. Here are sample URIs embedding Froglingo expressions:

```
http://www.myclienta.com
http://www.myclienta.com/appl2/a b (c d)
http://www.myclienta.com/files/webpage.html
http://www.myclienta.com/jone.dow/
```

The expression in the optional field /t_0.../t_{n-1}/t_n can not include a few special characters before URIs arrive at web servers; and each of the special characters has to be replaced with an encoded form called escape. The escaped string of a character is the 2 digits of its hex number proceeded with the character "%". While you may convert other characters in an expression to their corresponding escaped strings, the following special characters must be converted:

Special character	Escaped form
/	%2F
?	%3F
:	%3A
%	%25
^	%5E

For example, the Froglingo expression `$x:[$x > 0]` must be at least converted to `$x%3A[$x > 0]`. The form `$%78%3A[$x >%200]` is acceptable too because the characters 'x' and ' ' have the corresponding escaped forms %78 and %20. When a web server receives URIs, those escaped characters are decoded. When users enter URIs via a web browser address field or a HREF value in HTML file, the users must do the escape character conversion manually. When users use HTML form, the conversion is automatically done by web browser.

Now with web browser alone, a user can communicate with Froglingo web server by typing URIs embedding Froglingo expressions. Remember that one has to login to a user account first before being able to access and to manage the data belonging to the account. The login format is:

//http://**dn**/login . "**password**"

Here are a few sample URIs to communicate with Froglingo web server:

```
//http://www.myclienta.com/"Hello World"
```

```
//http://www.myclienta.com/login . "i8u9i1o@"
//http://www.myclienta.com/create data_via_web = 43
//http://www.myclienta.com/print data_via_web
```

Using URIs alone is not as friendly as using a CMD window, it is sufficient to fully interact with Froglingo server except for using the root account.

7.3 HTML/XML Files

When a user via a web browser sends a request embedding a URI to a web server, the server responds with a HTML document which is further rendered to a web page by web browser. HTML files stored in Froglingo database are the sources of HTML documents. When a file embeds Froglingo expressions, the expressions are replaced with the corresponding normal forms (values) in the HTML documents.

XML (Extensible Markup Language, RFC3470) is a language used to do data communication between computer systems. A special form of XML is HTML (HyperText Marker Language), the language for HTML documents that are rendered for web pages as web pages. Here we treat both HTML and XML equally and call them XML document.

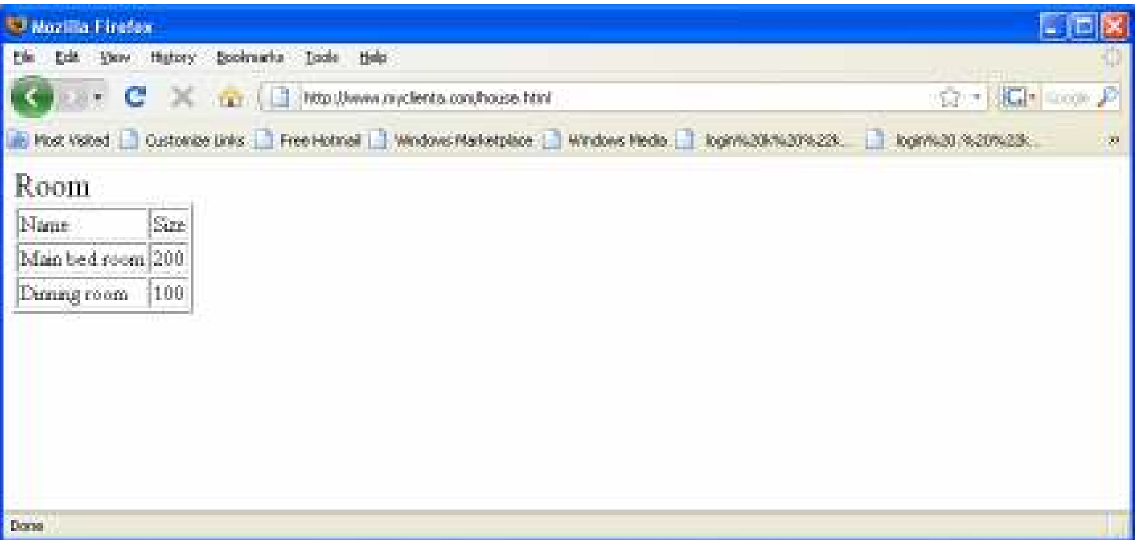
The basic structure of a XML document is a set of elements (or called blocks). Each element starts with an open tag and mostly ends with a close tag. An open tag is made up of a tag name, sometime followed by an optional list of attributes, all of which appears between angle brackets < >. The end tag of an element contains the same name as the on in the open tag, but proceeded by a slash /. An attribute is normally a pair of name and values separated by an equal sign =. Sometime, an attribute is a single value by itself. The tag and the attributes in an open tag are separated by one or more spaces. A text message can appear anywhere before or after a tag. An element can embeds one or more than one nested elements.

7.3.1 Document

If a XML file doesn't embed Froglingo expressions, it is stored as a whole in a binary stream. Assume that there is a file `house.html` in the folder where a Froglingo database is launched:

```
[//www.myclienta.com files] load house.html;
[//www.myclienta.com files] print house.html;
<html>
     <body>
          <font size=5> Room </font>
          <table border = 1>
               <tr>
                    <td>
                         Name
                    </td>
                    <td>
                         Size
                    </td>
               </tr>
               <tr>
```

```
                <td>
                    Main bed room
                </td>
                <td>
                    200
                </td>
            </tr>
            <tr>
                <td>
                    Dinning room
                </td>
                <td>
                    100
                </td>
            </tr>
        </table>
    </body>
</html>
```



There are XML and HTML document editors that help users to generate syntactically correct documents. However, many documents were written manually or generated from software programs. In practice a large volume of such documents contains syntactical errors. However, the web browsers in practice allow such errors and display information as much as they can. Froglingo goes along with the web browsers by storing them as a whole no matter if a document has any syntactical errors or not. In addition, Froglingo reports syntactical errors when documents are stored.

In the rest of this section, we will introduce Froglingo-preserved tags and attributes embedded in XML files. When a file embeds Froglingo expressions, Froglingo does a stronger syntax checking and stores the document only if there is no syntactical error.

7.3.2 Tag <frog>

There is a special block with tag name `frog` defined for Froglingo. When Froglingo

parses the block, it treats the entire content of the block as a term. When the block is downloaded from database, the term is replaced with the normal form of the term. To demonstrate the effects of Froglingo-specific tags and attributes, we assume that the user `www.myclienta.com` collected a database first:

```
[www.myclienta.com] print .;
customer 1000 fname =  "John";
customer 1000 lname = "Smith";
customer 1001 fname =  "Dennis";
customer 1001 lname = "Alexandra";
order (customer 1000) 10000 100000 product = storage apple;
order (customer 1000) 10000 100000 volume = 12;
order (customer 1000) 10000 100001 product = storage milk;
order (customer 1000) 10000 100001 volume = 1;
order (customer 1000) 10001 100003 product = storage apple;
order (customer 1000) 10001 100003 volume = 14;
storage apple price = 1.89;
storage apple volume = 500;
storage milk price = 2.95;
storage milk volume = 0;
```

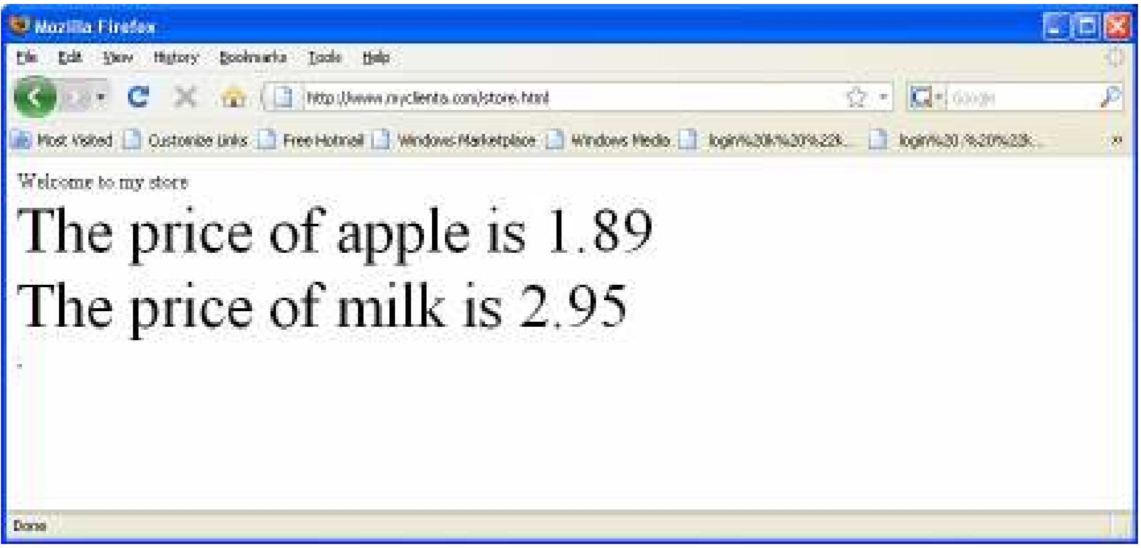
A file `store.html` has the content:

```
<html>
      <body>
            Welcome to my store <br>
            <font size = 8> The price of apple is
                  <frog>storage apple price</frog>
            </font> <br>
            <font size = 8> The price of milk is
                  <frog> storage milk price </frog>
            </font> <br>
      </body>

</html>;
```

It is uploaded to database:

```
[//www.myclienta.com] load store.html;
[//www.myclienta.com] grtacc store.html anyone;
[//www.myclienta.com] print store.html;
<html>
        <body>
                Welcome to my store <br>
                <font size = 8>The price of apple is
                        1.89
                </font> <br>
                <font size = 8>The price of milk is
                        2.95
                </font> <br>
        </body>
</html>;
```

When an `anyone` user access the web server with the URI:
http://www.myclinenta.com/store.html, the web page is displayed as:



One must ensure that the identifiers embedded between the block `<frog>` and `</frog>` have been defined in database before uploading HTML/XML files. Otherwise, the upload request will be rejected.

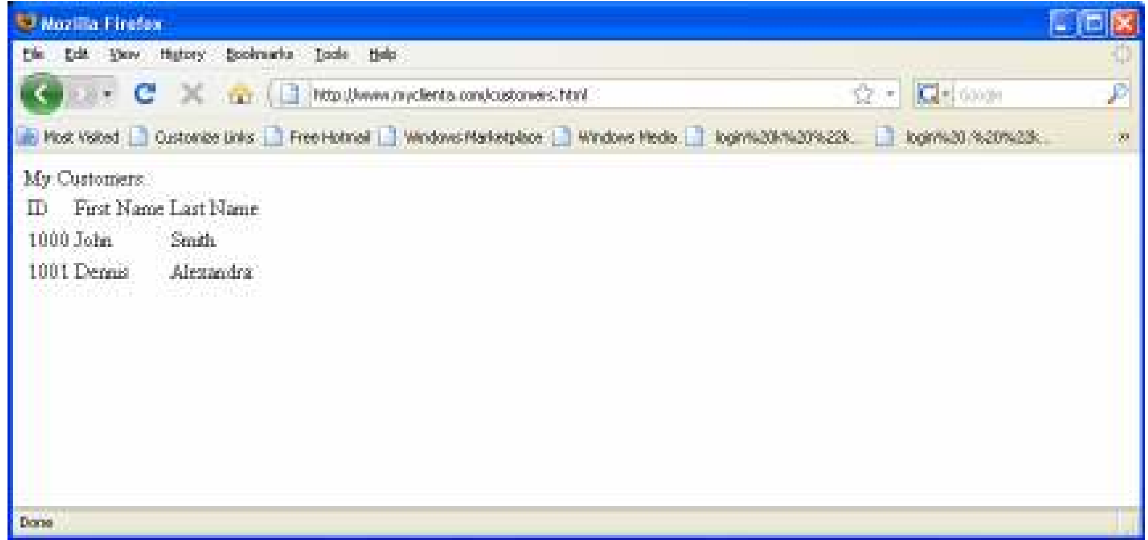
For the readers who are curious on how a XML/HTML file embedding Froglingo expressions is stored, here is a demonstration:

```
[//www.myclienta.com] print store.html in frog;
store.html 1 = <html>
      <body>
            Welcome to my store <br>
            <font size = 8> The price of apple is ;
store.html 2 = storage apple price;
store.html 3 =
            </font> <br>
            <font size = 8> The price of milk is ;
store.html 4 = storage milk price;
store.html 5 =
            </font> <br>
      </body>

</html>;
```

As another example, we upload another file `customers.html` that includes a <frog> block embedding select operation:

```
<html>
      <body>
            My Customers: <br>
            <table>
                  <tr>
                        <td> Customer ID </td>
                        <td> First Name </td>
                        <td> Last Name </td>
                  </tr>
                  <frog>
                  select “<tr><td>”,
                              mterm $cust,
                         “</td><td>”,
                              $cust fname,
```

```
                        "</td><td>",
                             $cust lname,
                        "</td></tr>"
                  where $cust {+ customer
                  </frog>
            </table>
      </body>
</html>;

[//www.myclienta.com] load customers.html;
[//www.myclienta.com] print customers.html;
<html>
      <body>
            My Customers: <br>
            <table>
                  <tr><td>ID</td><td>First Name</td><td>Last
Name</td></tr>
                  <tr><td>1000</td><td>John</td><td>Smith</td></tr>
                  <tr><td>1001</td><td>Dennis</td><td>Alexandra</td><tr
      >
            </table>
      </body>
</html>;
```



7.3.3 Attribute Proceeded with "frog"

Attributes are inside the open tag of a block. The value of an attribute is driven by the state of database. Froglingo recognizes the form of an attribute name and value pair in a tag: <**tagname** … frog:**attr**=**qtermq** …>

Here **tagname** is the tag name of a block, **attr** is an attribute name, i.e., an identifier in Froglingo, and qtermq is a sequence of characters. Froglingo server treats **qtermq** as a term. If **qtermq** is a string surrounded by a pair of the double or single quotes, the quotes are stripped off before Froglingo parses it. If there is spaces, or special characters in **qtermq**, **qtermq** must be a string surrounded by a pair of double or single quotes, or by `(` and `)`.

When the block is downloaded from database, **qtermq** is replaced by its normal form. As an example, we assume that we add the following data into database:

```
[www.myclienta.com] create image width = 183;
```

A html file has a tag:

```
<IMG src="text6.jpg" frog:width=(image width) hight= "200">
```

It will be replaced with the following tag when the file is downloaded:

```
<IMG src="text6.jpg" width="183" hight= "200">
```

7.3.4 Attribute "frog:if"

A block in a document may or many not need to be displayed depending on the state of database. For this purpose, we introduce a special attribute "if" in a block:
<**tagname** … frog:if= **qtermq** …>

The value **qtermq** is a boolean expression in Froglingo. When the block is to be downloaded, the value **qtermq** is evaluated. If it is equal to `true`, then the block is displayed. Otherwise, the block is not displayed. For example, if the file `store.html` is modified as the following:

```
<html>
      <body>
            Welcome to my store <br>
            <font size = 8 frog:if=(storage apple volume > 0) > The
price of apple is <frog>storage apple price</frog> </font> <br>
            <font size = 8 frog:if=(storage milk volume > 0) > The
price of milk is <frog> storage milk price </frog> </font> <br>
      </body>
</html>
```

If the volume of milk in the store is 0 or not defined, the greeting message for milk will not be displayed:

```
[www.myclienta.com] print store.html;
<html>
      <body>
            Welcome to my store <br>
            <font size = 8> The price of apple is 1.89 </font><br> <br>
      </body>
</html>
```

7.3.5 Attribute “frog:while”

In Section 22.2, the file `customers.html` embeds a select operation, which enumerates all the customers. This function can be alternatively done by specifying a while-loop for a block. The syntax is:

```
<tagname … frog:while=var …>
…
<frog> frog_expression_may_having_var </frog>
…
</tagname>
```

Here **`var`** is a variable declaration having a range. When the block is to be downloaded, the database is searched to find all the terms falling into the range of the variable. For each term selected, the block is repeated by replacing those Froglingo expressions with their normal forms. During the evaluation of the normal forms, the instances of the variable are substituted with the selected term. For example, we can modify the file `customers.html`:

```
<html>
      <body>
            My Customers: <br>
            <table>
                  <tr frog:while=($id:[customer $id != null])>
                        <td> <frog>$id </frog> </td>
                        <td> <frog>customer $id fname</frog> </td>
                        <td> <frog>customer $id lname</frog> </td>
                  </tr>
            </table>
      </body>
</html>
```

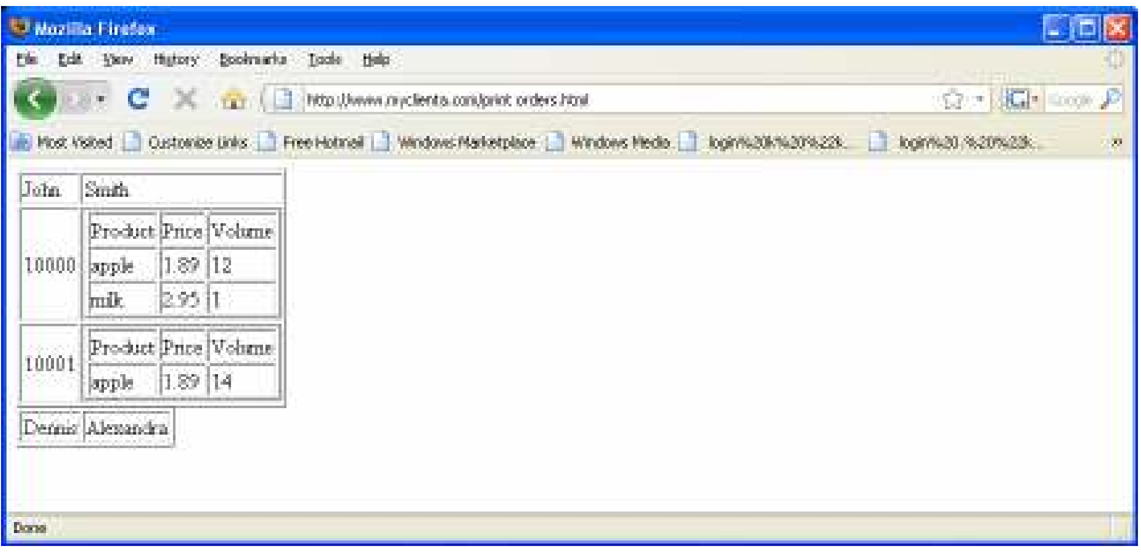
Then:

```
[www.myclienta.com] print customers.html;
<html>
      <body>
            My Customers: <br>
            <table>
                  <tr>
                        <td>1000</td>
                        <td>John</td>
                        <td>Smith</td></tr>
                  <tr>
                        <td>1001</td>
                        <td>Dennis</td>
                        <td>Alexandra</td><tr>
            </table>
      </body>
</html>
```

A while-loop for a block can be nested inside another block having its own while-loop. For example, a summary report about orders can be generated through the file `orders.html`:

```
<html>
<body>
     <table border=1 frog:while=($cust: [$cust {+ customer])>
     <tr> <td> <frog>$cust fname</frog> </td>
          <td> <frog>$cust lname</frog> </td>
     </tr>
     <tr frog:while=($ord: [order $cust $ord != null])>
          <td> <frog> $ord </frog> </td>
          <td>
               <table border = 1>
               <tr>
                 <td> Product</td><td>Price</td><td>Volume</td>
               </tr>
               <tr frog:while=($item:[order $cust $ord $item !=
null])>
                 <td>
                    <frog>
                    mterm (order $cust $ord $item product)
                    </frog>
                 </td>
                 <td>
                    <frog>
                    order $cust $ord $item product price
                    </frog>
                 </td>
                 <td>
                    <frog>
                    order $cust $ord $item volume
                    </frog>
                 </td>
               </tr>
               </table>
          </td>
     </tr>
     </table>
</body>
</html>
```

A request `http://www.myclienta.com/orders.html` from web browser will bring the following web page:



7.3.6 File Argument

In Froglingo, one can pass parameters into a XML file. It is done via variables pre-inserted in XML file. When XML files containing variables are downloaded, values can be passed to the files through the variables such that the XML files are instantiated correspondingly.

Document arguments are declared inside a <frog> block at the first line of a XML file, and there is no other text before the block. If there are more than one variable declared, they are delimited by comma ‘,’. A declared variable can appear anywhere in the rest of the file through Froglingo preserved tags or attributes. Here is a sample XML file `orders2.html` having variables:

```
<frog> $cust, $ord</frog>
<html>
<body>
      Customer <frog> $cust fname</frog>  
      <frog> $cust lname </frog>   
            has purchase order # <frog>$ord</frog>: <br>
<table border=1>
    <tr>
      <td>Product</td><td>Price</td><td>Volume</td>
    </tr>
    <tr frog:while=($item:[$item {+ order $cust $ord])>
      <td>
            <frog>
                  mterm ($item product)
            </frog>
      </td>
      <td>
            <frog>
                  $item product price
            </frog>
      </td>
      <td>
            <frog>
```

```
                  $item volume
            </frog>
      </td>
 </tr>
</table>
</body>
</html>
```

Once this file is uploaded to database, user with proper privilege, can view the following page by providing the URI:
`http://www.myclienta.com/orders2.html (customer 1000) 10000 in html`

7.4 Request via HTML Form

In section 7.2, we said that URIs are the messages that deliver user requests from web browsers to web servers. Users can enter URIs in web browser's address fields directly or give URIs as links (via the HTML HREF attribute). It however requires users to know URIs and Froglingo. An easy and commonly used method of obtaining business requests is to use HTML forms on web pages. This requires developers to write HTML documents embedding forms.

In this section, we extend the URIs discussed in Section 7.2. The extended URIs will be the ultimatum carrier of user requests entered through HTML forms with method "GET". In addition, this section discuss how files are uploaded via HTML forms with method "POST". Form data is ultimately recognized by Froglingo server.

7.4.1 Extended URIs

In Section 7.2 gave the basic URI syntax:
http://**domainname[:port][/exp**$_0$.../**exp**$_{n-1}$/**exp**$_n$**][/]**

The following syntax is also allowed:
http://**domainname[:port][/exp**$_0$.../**exp**$_{n-1}$**]/exp**$_n$?**a**$_1$="**v**$_1$"&...&**a**$_m$="**v**$_m$"

The extended form of URIs includes a set of optional attribute and value pairs. But if

there is at least one attribute name and value pair, the character ‘?’ must appear at the beginning of the pairs. The pairs are delimited by the character ‘&’; and the attribute name and value in a pair are delimited by ‘=’. An attribute name is an alphanumeric string; and an attribute value is a sequence of characters surrounded by a pair of character ‘"’. The value is normally entered by users via HTML form.

The attribute names $\mathbf{a_1}$, …, $\mathbf{a_m}$, prefixed with character ‘@’, may appear in $\mathbf{exp}_n$ serving as variables. When Froglingo server receives an URI, it will replace all the instances of the variables in $\mathbf{exp}_n$ with the corresponding values "$\mathbf{v_1}$", …, "$\mathbf{v_m}$".

7.4.2 HTML Form

Assume that a user with the account `www.myclienta.com` wants to collect customer information. The user has a function created in database:

```
[//www.myclienta.com] create rcv_cust_info $fname $lname =
           (create customer cust_index fname = $fname),
           (create customer cust_index lname = $lname),
           (update cust_index = (cust_index + 1));
```

An identifier `cust_index` had been created earlier:

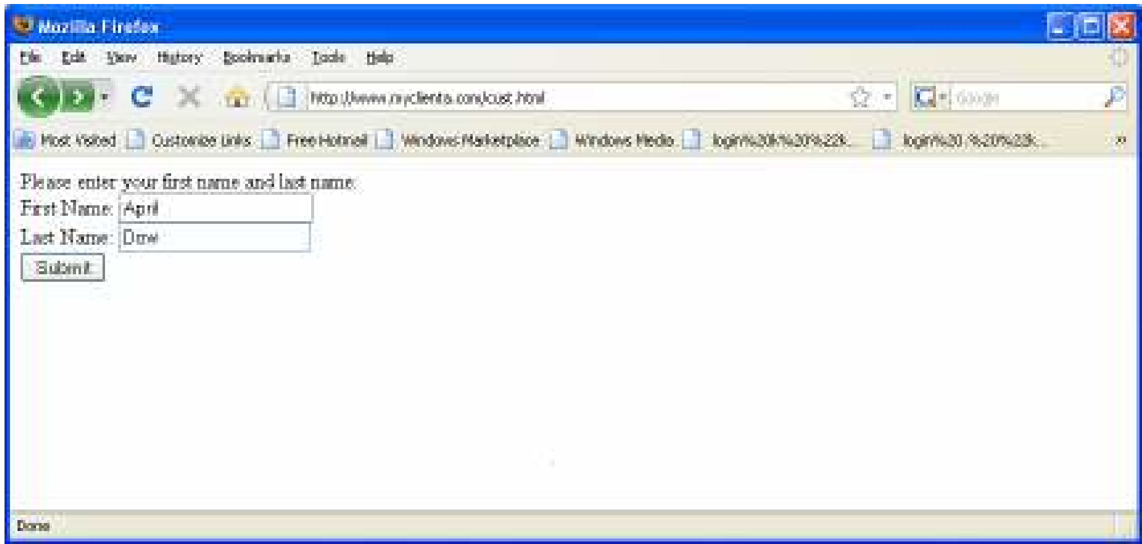
```
[//www.myclienta.com] cust_index;
1003;
```

A HTML file `cust.html` is available before being processed by Froglingo server:

```
<html>
<body>
       Please enter your first name and last name:
       <form name="input" action="rcv_cust_info @fname @lname"
              method="get">
       First Name: <input type="text" name="ident:fname"> <br>
       Last Name: <input type="text" name="lname">
       <input type="submit" value="Submit">
       </form>
</body>
</html>
```

```
[//www.myclienta.com] load cust.html;
[//www.myclienta.com] grtacc cust.html anyone;
```

The following page can be requested by anyone via web browser:



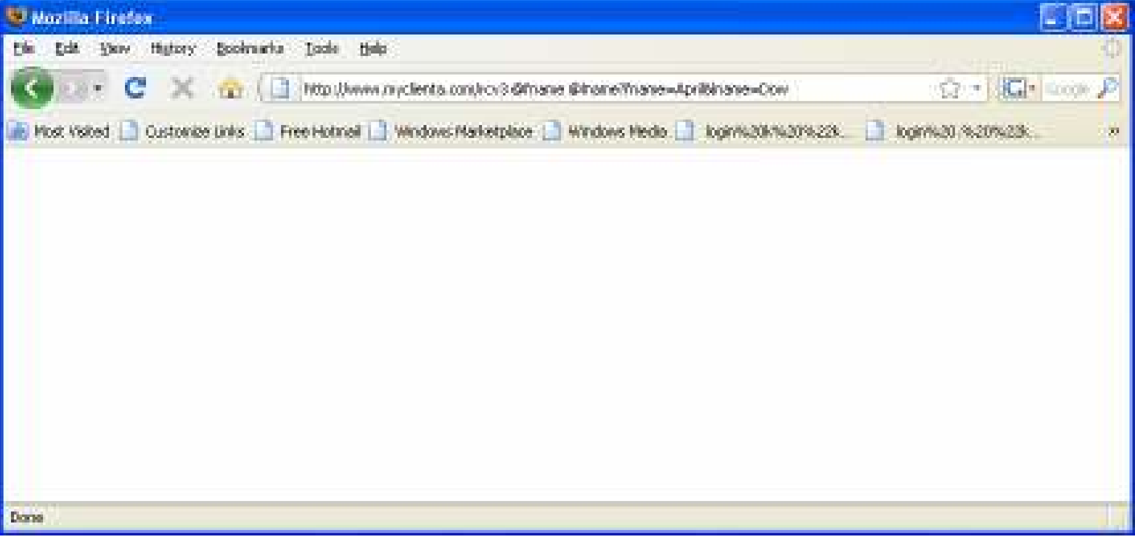
After the web user entered her first and last names and clicked "Submit" button, the request went to web server. The URI arrived at Froglingo server is:

```
http://www.myclienta.com/rcv_cust_info @fname
@lname?fname=April&lname=Dow
```

Froglingo server resolves the URI as the following equivalent request:

```
[//anyone] //www.myclienta.com rcv_cust_info "April" "Dow";
```

The URI was also displayed on the web browser. Since `rcv_cust_info` returns `void`, the web browser is blank. See the screen shot below.



7.4.3 Upload via Web Browser

Files can also be uploaded to Froglingo database by using web browser. Assume the owner of the website `www.myclienta.com` wants to collect customer information including image files. The owner has a function created in the database:

```
[//www.myclienta.com] create rcv_cust_info $fname $lname $photo=
              (create customer cust_index fname = $fname),
```

```
            (create customer cust_index lname = $lname),
            (load customer cust_index $photo in stream),
            (update cust_index = (cust_index + 1));
```

The term `cust_index` was created earlier:

```
[//www.myclienta.com] cust_index;
1003;
```

A HTML file `cust.html` is created and uploaded to the database:

```
[www.myclienta.com] load cust.html;
[www.myclienta.com] grtacc cust.html anyone;
[www.myclienta.com] print cust.html;
<html>
<body>
       Please enter your first name and last name:
       <form name="input" action="cust_index @fname @lname
@photo"
              enctype="multipart/form-data" method="post">
       First Name: <input type="text" name="fname"> <br>
       Last Name: <input type="text" name="lname">
       Photo: <input type="file" name="photo">
       <input type="submit" value="Submit">
       </form>
</body>
</html>
```

When a user browses the HTML document, the form is displayed on the web browser and the user is able to enter the first name and last name, and upload a (image) file (see the following web page).

Chapter 8: End User, Developer, and Administrator

Users possessing user accounts can be classified into 3 categories according to their assigned privileges and their responsibilities:

1. End users. An end user sends requests to database server through HTML forms. He or she doesn't need to have any knowledge about Froglingo beyong web pages. An end user interacts with database server via `anyone` user, or a user account created via a super user account.
2. Developers. A developer is responsible for constructing a web site embedding business logic. He or she must know Froglingo and possess an administration privilege to the user account named as the domain name of a web site. A good web design is to hide footprints of Froglingo, and to display business contents intuitively in native languages, tables, images, animations, or any other presentation media.
3. Administrators. An administrator is responsible for keeping database server up running. He or she has the root (`//`) administration privilege over the entire database. The main focus is system reliability, performance, and security. An administrator is required to deal with server startup, shutdown, data backup, performance tuning, and network and operating system trouble-shootings.

End users may not want to see special values from database such as `null`, `void`, error messages, and integer presentations of times. Developers may not want to know the details about how a database is backed up. This section discusses the facilities that help to segregate responsibilities between end users, developers, and administrators.

8.1 Simple Data Type and Membership Operator

We have given an example of using the operator `isa` and the data type `integer` in Section 14.3. Here we give a full coverage about the operator and the other simple data types. Then we discuss special values in the rest of this section.

So far, we have introduced constants, integers, real numbers, strings, and files. In corresponding to the different categories of constants, the special constant terms `integer`, `real`, `string`, and `stream` are introduced and called data types. The binary operator `isa` is used to evaluate if a term belongs to a data type.

Given the binary operation form: **`I`** `isa` **`T`**, **`T`** is normally expected to be a data type, and **`I`** is to be an instance of the data type **`T`**. Here are a few examples:

```
[//] 3.2 isa real;
true;
[//] 3.2 isa string;
false;
```

Given the binary operation form: **`I`** `isa` **`T`**, **`T`** also can be a term that is not a data type, when this is the case, the instance **`I`** is expected to syntactically match **`T`**. An example has been given in Section 20.2 about the special identifier `signature`:

```
[//] signature isa signature;
true;
```

A few more examples are:

```
[//] 3.2 isa 3.2;
true;
[//] Dave salary isa Dave salary;
true;
[//] www.myclienta.com isa signature;
true;
```

Using `isa` to compare a term to another non-data-type term is more meaningful when it is applied to the special values `void` and `error` in the coming sub sections.

Like data schemas in traditional DBMSs, data types can be further expanded to allow users to define their own data types. Serving as data schemas, data types check and reject invalid data entered by end users. While practically it is a direction for a Froglingo future release, it technically is not necessary because developers can construct their own business logic to substitute user-defined data types.

8.2 Void for Nothing

When a developer initiates an update operation via a CMD window, the developer will know that the update is successfully executed if he/she didn't see any error message but a prompt sign (such as `[//]`) for next line. This interaction is not sufficient when a user interacts with database across network because the prompt sign is not returned (see a few blank web pages in the screen shots in Section 23 after update operations). End user who submitted update requests may want to see a confirmation in the context of business logic through a web page rather than a blank page. A developer satisfies end user's needs by utilizing the special identifier `void`.

In Froglingo, the special identifier `void` has a special normal form: empty, or no action taken by database. An update operation, when executed successfully, returns `void`. We didn't see anything (empty) after a successful update operation in the earlier sections because the return value `void` is further reduced to empty. One may count empty to be a term, but empty is not expressible at all.

The reason to assign `void` an empty as the normal form is that developers can spell out the identifier `void` in business logic construction and detect the status of update operations before presenting a right web page according to the operation status. Assume that a developer constructed a web page that allows end users to submit an update operation. If the operation is successful, the developer intends to show `successful.html` to the end users. Otherwise, another page `failed.html` will be displayed. Here is a sample coding:

```
[//www.myclienta.com] create build_op $x =
           (create an_assingee $x = somefun $x);
[//www.myclienta.com] create detect_fun $x: [$x isa void] =
          print successful.html;
[//www.myclienta.com] create detect_fun $y = print failed.html;
[//www.myclienta.com] detect_fun (build_op 3);
```

The first line constructed a method `build_op` to be exported via `grtacc`. It returns `void` when an update operation of `an_assingee` is successfully executed. The second and the third lines constructed another method `detect_fun` that takes different actions depending on what is the value passed via the variable `$x`. The last line is to call the two methods.

8.3 Null for Undefined

As it has been clearly defined in section 2, `null` is a constant by itself, representing

undefined as a return value of a query expression. However, end users may not want to see it when a query yields with `null`. In this case, a developer may use `void`, or a textual message such as "no data was retrieved from database", to hide `null` from end users. Here is a sample code:

```
[//www.myclienta.com] create display_logic null = void;
[//www.myclienta.com] create display_logic $x = $x;
[//www.myclienta.com] display_logic "A string can be displayed";
A string can be displayed;
```

8.4 Error for Failure

A failed execution to a (either query or update) operation, as we have seen in the earlier sections, always returns a system error message. It is an ideal way of presentation when a developer, knowing Froglingo, interacts with database via a CMD window. But end users may not want to see the error message. If end users want to see it, where it would be displayed in a returned web page a question. Here we introduce another special identifier `error`.

The identifier `error` is given with a normal form. What is the normal form depends on the state of a user session. The normal form is always `null` in most cases. For example:

```
[//] error;
null;
```

The exception is when the execution of an operation is failed and returns an error message. At this moment the identifier `error` is assigned with the normal form, i.e., the error message from the execution. In the earlier sections, we saw the error messages rather than `error` because `error` is further reduced to the error message itself. Provided with the identifier `error`, developers can manipulate the returned error messages as the way end users wants. Here is a re-write to the sample code presented in section 24.2:

```
build_op $x = (create an_assignee $x = somefun $x);
detect_fun $x: [$x isa error] = print failed.html $x in html;
detect_fun $y = print successful.html;
```

The first construction is the same as the previous one. The second and the third lines used `error` rather than `void` to detect values passed to method `detect_fun`.

8.5 Day and Time

Currently, Froglingo supports a form of expressing day and time:

`'M[M]/D[D]/YYYY [hh:mm:ss]'`

For examples:

```
[//] '3/5/2009';
1236229200;
[//] '03/05/2009 15:03:35';
1236283415;
```

To display a time in its nature forms, one can use the built-in operator `dtform`:

`dtform` **`[format] number_for_time`**

Here the optional field **`format`** is a string showing the format on how a given **`number_for_time`** is to be displayed. When **`format`** is not provided, the default format is assumed:

`"Day Mon Date hh:mm:ss YYYY"`

Here **`Day`** is one of the followings: `Mon, Tue, Wed, Thu, Fri, Sat,` and `Sun`; **`Mon`** is one of the followings: `Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov,` and `Dec`. For examples:

```
[//] dtform 1236229200;
"Thu Mar 05 00:00:00 2009";
[//] dtform '03/05/2009 15:03:35';
"Thu Mar 05 15:03:35 2009";
```

The field **`format`** is a string including one or more than one percentage '%' followed by a special character. A 2-character sub string in **`format`** that starts with "%" and ends with another character is to be replaced with its corresponding substitution value according to the mapping table listed below during execution:

pattern	Substituting values
%a	Mon, Tue, Wed, Thu, Fri, Sat, Sun
%A	Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday
%b	Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec
%B	January, Feburary, March, April, May, June, July, August, September, October, November, December
%d	01, 02, …, 30, 31
%D	A date in the format 'MM/DD/YYYY'
%j	A yyyy in the format 'YYYY'
%H	the time format: 'hh:mm:ss'

For examples:

```
[//] dtform "Date: %D" 1236229200;
Date: "3/5/2009";
[//] dtform "Year: %j, Month: %b, and Date: %d" '03/05/2009 15:03:35';
"Year: 2009, Month: 03, and Date: 05";
```

Froglingo supports a special identifier `timestamp`. When it is called, it returns the integer presentation of the current system time. For example:

```
[//] timestamp;
1236294968;
[//] dtform timestampe;
"Thu Mar 05 18:18:32 2009";
```

8.6 Database File

As a root administrator, who has the physical access to the files related to Froglingo at operating system level, he/she needs to know two files at operating system level: a database file and an event log file.

The first one is database file. The default file used to store Froglingo database is `.EPDB`

located in the Froglingo home folder. To start a new database, you can explicitly name a database file name. Here is the syntax:

```
c:\frogdir frog -f db_file_name
```

The database files are not maintained or monitored by other facilities at this time. However, the batch facilities "print" and "load" can be used to back up data and to restore data.

8.6 Log Files

There are currently three log files. They are located in the same folder (the Froglingo home folder) where `frog.exe` is located.

The first file is `frog.log`. It records all the requests in Froglingo expression from either local CMD windows or from web browsers across network. Each request, in the log file, is followed with any system or error messages.

The second log file is `http_in.log`. It records all the requests in HTTP messages from network. If a HTTP message includes uploaded file(s), the file contents are ignored by default. To record file contents, one needs to add an option `–whole_http_in` in the command line of launching Froglingo server:

```
c:\frogdir frog –p 80 –whole_http_in
```

The third log file is `http_out.log`. It is to record all the out-going HTTP response messages. By default, they are not recorded. To record it, one needs to add an option `–http_out` in the command line of launching Froglingo server:

```
c:\frogdir frog –p 80 –http_out
```

The log files are not maintained or monitored by other facilities.

Appendix A: Release Notes

The core system having the concepts covered in Unit 1 and Unit 2 was implemented in November 2004 (Release 0.1); and has been applied to many experimental applications.

The current version is the first public release (Release 1.0), covering all the features discussed in this document.

There are still many functions including multithreads support to be added in a future release.

Appendix B: Grammar

The grammar listed here is simplified. Hope it gives readers another view to know the core concepts of Froglingo.

```
input:
      | input line ';'
      | input error ';'
      | input ';'
;
```

```
line:   term
      | bin_exp
;
term: '(' term ')'
      | atom_term
      | application
;
bin_exp: bool_exp
      | num_exp
      | bin_assign
;
term_list: term
      | term_list ',' term
;
bin_assign: term '=' term
;
assign_value :
      | '='  term_list
;
opt_num:
      | term
;
order_clause: DESCENT
      | ASCENT
;
sort_clause:
      | SORT sortClauses
;
sortClauses: one_sort
      | sortClauses ',' one_sort
;
one_sort: term order_clause
;
where_clause :
      | WHERE bool_exp
;
summary_clause :
      | SUMMARY term_list
;
atom_term: LABEL
      | TO_UNIOP /* mterm, pterm, pfirst, … */
      | variable
;
query_name: SELECT
      | IS_THERE
;
```

```
application: term atom_term
      | term '(' term ')'
      | term  '(' bin_exp ')'
      | '(' bin_exp ')'
      | query_name term_list where_clause sort_clause
summary_clause
      | VOID_OP term_constraint assign_value
      /* VOID_OP is create, record, update, or delete */
;
term_constraint : term
      | term ':' term
      | term ':' term ':' term
;
num_exp: term
      | num_exp PLUS num_exp
      | num_exp MINUS num_exp
      | num_exp MULT num_exp
      | num_exp DIVID num_exp
      | num_exp QUOTI num_exp
      | MINUS num_exp %prec NEG
;
range:
      | ':' '[' bool_exp ']'
;
variable: '$' LABEL range
;
bool_exp:  atom_bool
      |  '(' bool_exp  ')'
      |  bool_exp  high_bool_optr bool_exp
high_bool_optr: AND
      |  OR
;
atom_bool:  term
      |  term bin_optr term
      |  '(' term bin_optr ')'
;
bin_optr: ‘isa’ | ‘<’ | == | != | … | ‘{=+’ | ‘<-‘ | …
;
```


Acknowledgement

The authors would like to thank Yuhui Huang, Phillips Lu, Ed Heaney, and Ellen Xu for their work during the experimental application development of using Froglingo. Authors also would like to thank those anonymous reviewers of research conferences and many of our friends who provided encouraging comments on the ideal of having a language system like Froglingo. We specially

thank our family members who constantly supported us on this project over the past decade.

Reference

1 K. H. Xu, S. Gao, J. Zhang, R. R. McKeown. “Data Model and Total Recursive Functions”. http://www.froglingo.com.

2 K. H. Xu, J. Zhang, S. Gao. “Assessing Easiness with Froglingo”. The Second International Conference on the Application of Digital Information and Web Technologies, 2009, page 847 - 849.