User's Guide to

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

Release 2.0, March 14th, 2013

1. INTRODUCTION

Many database applications were written in programming languages 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, 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.

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 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 the combination of 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.

1.2 Who should use Froglingo?

Froglingo is a generic tool for software applications. It can be used to construct all kinds of information management systems involving data, files, and business logic, including

- database applications,
- content management systems,
- data warehouses,
- web sites,
- collaborative computing environment across multiple organizations.

1.3 Why is Froglingo?

Froglingo makes software development easier because

- it is a single language to express business data and business logic,
- database applications in Froglingo can communicate without data exchange agents,
- the EP (Enterprise-Participant) data model, part of Froglingo, is more expressive than any other existing data models including SQL,
- it stores files, data, and business logic in a uniformed format in a single storage space,
- 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, and
- it has its own built-in web server that communicates with web browser across network.

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 HTML related specification languages with which web pages can be constructed.

1.5 System setup

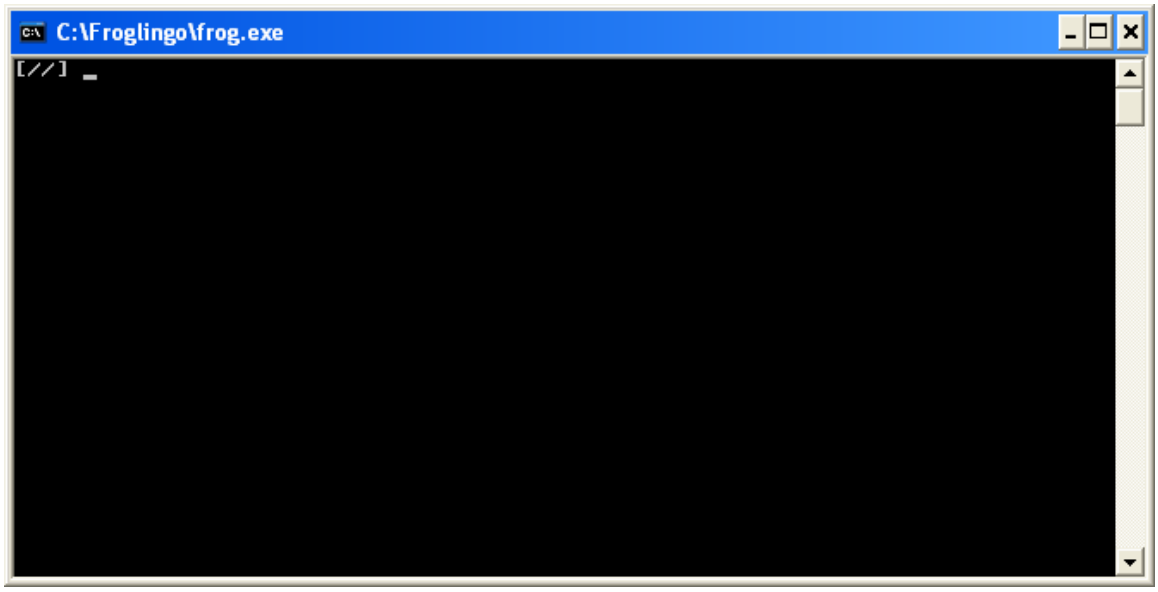
To run Froglingo, a computer needs to be installed with:

- Windows 7, or Windows XP system, and
- A web browser such as Internet Explorer, FireFox, or Google Chrome.

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 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 can be used to query data. They are more expressive than those traditional DBMSs. Queries on 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. 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;
```

We will see how it will be used in Chapter 5.

1.6.2 An information community

The data you entered into Froglingo databases is not shared with anyone else unless you explicitly granted permission to some one. 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 the Internet

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

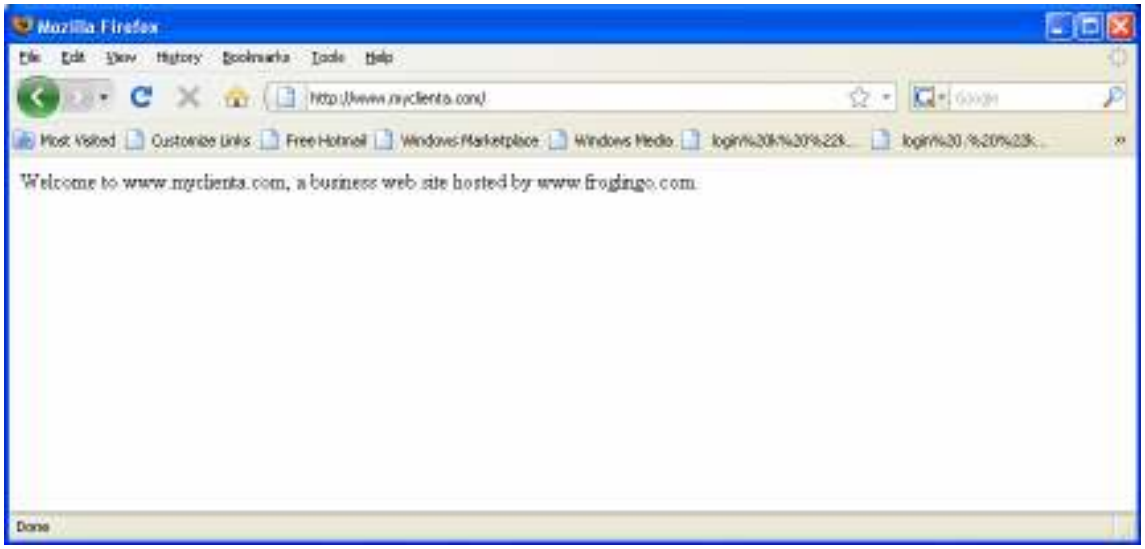
```
[//www.myclienta.com] load photo.jpg;
[//www.myclienta.com] load index.html;
[//www.myclienta.com] print index.html;
index.html = <html>
<body>
     Welcome to www.myclienta.com, a business web site hosted by
www.froglingo.com.
</body>
</html>;
[//www.myclienta.com] grtacc index.html anyone;
[//www.myclienta.com]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

The command `grtacc` above assigned the built-in user account `anyone` to have read-only 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
```



1.8 Document organization

In Section 2, we introduce the constructors with which business data is constructed in Froglingo databases. In Section 3, we introduce a rich set of built-in operators on databases with which business data is queried. In Section 4, we introduce variables and sequential terms, with which business logic can be expressed.

Section 5 is about data security, i.e., how user accounts are set such that data is accessed based on permission. We call a database as an information community to mean that Froglingo offers an environment in which users can construct their own data in private spaces, selectively specify data to be shared with others, and collaborate with others in data construction. In the community, individual users, business owners, and employees are correlated based on their roles in the community. This community is reachable across the Internet via web browsers, which is discussed in Section 7.

Froglingo manages files as well, including binary files and HTML/XML files. We discuss it in Section 6. In Section 8, we discuss miscellaneous issues that system administrators and developers need to consider. In Section 9, we provide some case studies.

In the document, we use the font `Courier New` for the text the Froglingo system would generate; 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.

Through the document, we assume that all the sample data appeared in the order of the document is entered to a single Froglingo database.

2 DATA CONSTRUCTION

In this section, we introduce the concepts that allow users to construct business data. There is not a precise mathematical correspondence to the concept of business data. But here we say that business data is computer presentations that represent a finite set of entities and their relationships in the world.

2.1 Constants

Like in other languages, integers, real numbers, and strings are constants. 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 operators 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 Identifiers

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 later in 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 Terms

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 consisting of an ordered pair of two terms, is called a combinatory term, denoted 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`, and `a b (c d)`.

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. Both the plus-term and the minus-term of a comb-term appeared in an assignee must not have an assigner,
3. One assignee has at most one assigner, and
4. If there is a sequence of assignments: $\mathbf{M_0} = \mathbf{M_1}, \mathbf{M_1} = \mathbf{M_2}, \ldots, \mathbf{M_{n-1}} = \mathbf{M_n}$, then $\mathbf{M_n}$ must not contain $\mathbf{M_0}$ as a sub-term.

All the constraints enforce a database to represent valid functions. The rule 1 says that a constant defines itself and it is not allowed to have another assigned value. A left-most sub-term of an assignee, except the assignee itself, has a derived value from the assignee, and therefore it cannot have an assigner by itself. Allowing a right-most sub-term of an assignee, except the assignee itself, would give users more flexibilities in constructing databases, but require more complicated rules to keep databases in shape and lower system performance. The rule 2 simply prevents a right-most sub-term of an assignee, except for the assignee itself, from having an assigner. Given an application *m n*, the rule 3 enforces that *m* is a function, i.e., applying *m* to *n* yields a unique value. Rule 4 ensures that each assignee is assigned with a valid value. Without Rule 4, we would have an assignment, e.g., *m n* := *m n 5*, with which applying *m* to *n* would not yield a normal form, i.e., *m n* == *m n 55 … 5*. The assignment *m n* := *m n* is invalid as well because it doesn't result in a valid function.

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.
```

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 indicates: John, born on 6/1/1990, is a resident registered with his SSN = 123456789 in the 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**.

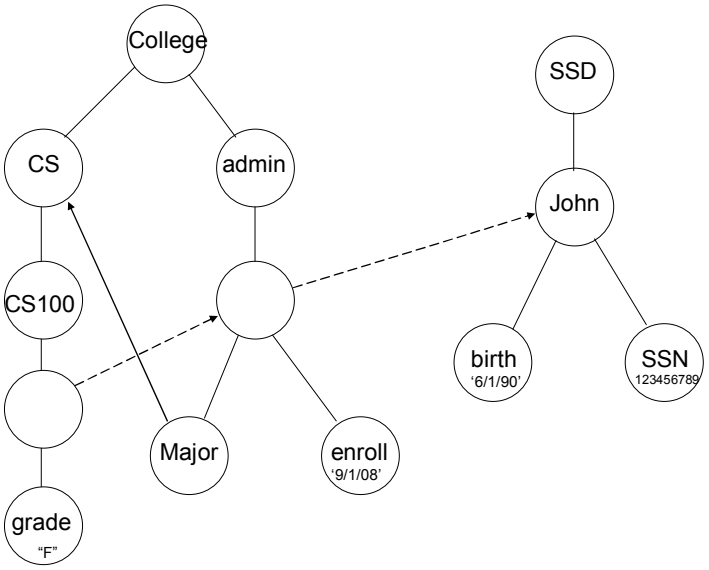
Examples:

```
[//] 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**.

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 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 Section 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.

3 DATA QUERIES

Given a database, an arbitrary term that was defined in Section 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 produce 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. Froglingo has 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 forms and evaluations

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

arbitrary terms can be reduced to unique normal forms under a database. Since database is evolving, the normal form of a term may change from time to 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 Booleans 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 booleans

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 queries

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, i.e., the outputs of queries are not single terms, but sets of terms.

3.3.1 Variables

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 operations

Select expressions are defined in the syntactical form:

`(select` **M_1, M_2, …, M_n** `where` **condition_clause**`)`

Where,

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

Here are a few examples.

```
[//] 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;
```

In this query, the system retrieves all the entities functionally dependent on SSD.

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

```
College admin (SSD John)
College CS CS100 (College admin (SSD John));
```
In this query, the system retrieves all the entities argumentatively dependent on John.

```
[//]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;
```
In this query, 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 not defined in the database). For each student selected, the system further prints out the names of the college, the department, and the student.

3.3.3 Index

A built-in operator `index` can appear as the listing $\mathbf{M_1}$, $\mathbf{M_2}$, …, $\mathbf{M_n}$ in a select operation (`select` $\mathbf{M_1}$, $\mathbf{M_2}$, …, $\mathbf{M_n}$ `where` **`boolean_operation`**). It indexes rows generated from the operation. 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 outputs can be alphabetically ordered by specifying a sort clause in a select operation. The syntax is

```
(select M1, M2, …, Mk
      where  boolean_operation
      [sort n1 order[, n2 order, …, ni order]]
)
```
where $\mathbf{i} \leq \mathbf{k}$, and a number $\mathbf{n_j}$, where $\mathbf{j} \leq \mathbf{i}$, is the index of an element $\mathbf{M_h}$ in the listing $\mathbf{M_1}$, $\mathbf{M_2}$, …, $\mathbf{M_k}$, i.e., $\mathbf{n_j} \equiv \mathbf{h}$ and $\mathbf{h} \leq \mathbf{k}$. The parameter **`order`** can be either `ascent` or `descent`. When the key word `sort` appears in a select operation, at least one order clause needs to be specified. 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;
```

Multiple order clauses can be specified. To sort person's names for the people who have the same salary, for example, we give the following expression:

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

```
(select M1, M2, …, Mn
      where  boolean_operation
      [sort n1 order[, n2 order, …, ni order]]
      [summary t1[,t2, …, tj]]
)
```

Here **t1**[,**t2**, …, **tj**]] is a sequence of terms, each of which may be a regular term, the built-in operator `count`, or an aggregate clause

aggregate n

where **aggregate** is one of the following: `max`, `min`, `ave`, and `sum` and **n** is a number representing a position of the output list **M1, M2, …, Mn**. 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;
```

In the select operation above, `"Summary"` and `(0.3 * (sum 2))` are regular terms, and `ave 2` and `sum 2` are aggregate clauses.

3.4 Derivative relationships

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

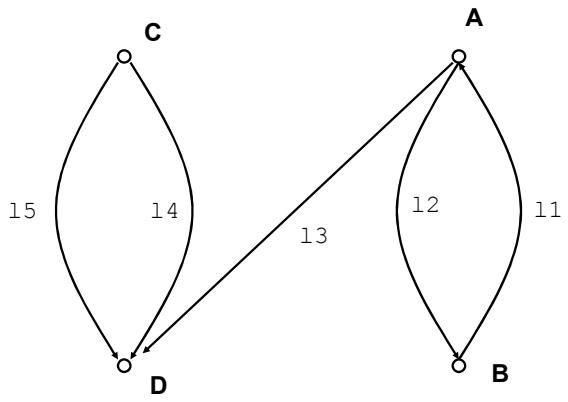
To show the way Froglingo manages directed graphs including cycles in the coming sub-sections, we create data representing a directed graph:

```
[//] 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 Equations

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 operation:

```
[//] 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.

3.4.2 Functional derivatives

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 derivatives

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 derivatives

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 earlier, 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 derivatives

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

The operators introduced earlier are are related. Here is a summary:
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.

4 BUSINESS LOGIC

Managing business data is the first, but not the final step in database application development. Business logic, though not having a precise mathematical correspondence, refers to computer presentation for business work 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 not red, green, and 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 a trade. A trade wouldn't happen unless at least two objects are exchanged between two parties simultaneously.

By adding variables and sequential terms, Froglingo becomes compete in business data and business logic presentation, 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, 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 Variables and databases

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.

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);
```

Variables serve as placeholders for the domains of functions. Variables bring infinite number of entities into databases. 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);
```

4.1.2 Variable ranges

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 containing the 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 terms as its range. Therefore functions without ranges are partial functions.

Sometime, a function may need to be define by cases. For example, we want the tax rate to be 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 the case-based definition above, we defined multiple variables under the function `tax` and each variable is associated with a range. When multiple variables are entered for a single function, they are in the alphanumerical order of the variables under the function. Here is a print out of function `tax` showing the order they are stored:

```
[//] 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);
```

Note that Froglingo is implemented without verifying if the variable ranges under a function are properly specified, e.g., if the ranges cover all the possibilities and if there is any overlaps among ranges. For example, if function `tax` is 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);
```

where there is no range specified for the variable `$r2`, the query `tax 300000` would not return its correct answer. It's user's responsibility to ensure the integrity of variable ranges.

With variables in databases, we can define functions with domains that are specified by both variables and individual cases. 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 a function with a domain of having both individual cases and variables is applied to an input, the system always tries to match one of the individual cases first.

4.1.3 Queries on infinite data retrieval

Variables allow functions with infinite data to be stored in databases. Given a function `f` and an input `t` ranging in an infinite domain, we can query databases to retrieve value `f(t)`.

```
[//] 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";
```

4.1.4 Function recursions

With variables, 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 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;
```

Before applying the function `path` to the sample directed graph given in Section 3.4, we add one more link into the graph:
```
[//] create D l6 = E;
```
Then,
```
[//] path C E;
CDE;
```

With variables, a calculation in Froglingo may never terminate. With the expression,
```
[//] fac 1.1;
```
for example, the evaluation process doesn't never terminate normally. Similarly the system on the

expression:
```
[//] path A B;
```
will not terminate normally as well because mathematically there are a infinite number of paths between `A` and `B`: `A B, A B A, A B A B, ….`

4.1.5 Update operations in assignments

Update operations can be assigners in databases. 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 enter `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 entered but not be displayed. Section 9 provides a full coverage of the identifiers `void` and `error`.

4.1.6 Assignment updates with variables

A term with variables cannot be queried. For example:
```
[//] fac $x;
The variable $x cannot be referenced directly.
```

To modify the assigner of an assignment with variables, one needs to specify the assignee and then redefine its new assigner. For example:
```
[//] update fac $x = ($x + (fac ($x - 1)));
```

Assignees in databases cannot be modified. To modify an assignee, e.g., the ranges of variables, one needs to delete the assignment first before creating a new assignment. For example:
```
[//] delete fun $x 1 $y;
[//] print fun;
fun $x 2 $y = ($x * $y);
```

where `fun $x 1 $y = ($x + $y)` is no longer in the database. When a variable in an assignee has its range defined, the range is not required to be specified when users intend to delete the assignment from a database.

4.1.7 Evaluation rules for variables

Now let's discuss how exactly system handles queries against functions with infinite properties. For the readers who are not interested, this section can be skipped.

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

$[\mathbf{v_0} := \mathbf{V_0}, \ldots, \mathbf{v_n} := \mathbf{V_n}]$

an environment.

Given a term **P** and an environment $[\mathbf{v_0} := \mathbf{V_0}, \ldots, \mathbf{v_n} := \mathbf{V_n}]$, we can obtain another term **P′** such that each instance of $\mathbf{v_i}$ appeared in **P** is substituted with $\mathbf{V_i}$. The substitution is done for

each **i** that is ranged from 0 to n. We use the form:

P:[$\mathbf{v_0}$**:=** $\mathbf{V_0}$**,…,** $\mathbf{v_n}$**:=** $\mathbf{V_n}$]

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

Given an assignee **M** without an assigner, in which there are variables $\mathbf{v_0}$**,…,** $\mathbf{v_n}$, here n >= 0, as sub-terms, and given a list of values $\mathbf{V_0}$, …, $\mathbf{V_n}$ from the corresponding ranges of the variables, then we define the substitution:

M:[$\mathbf{v_0}$**:=** $\mathbf{V_0}$**,…,** $\mathbf{v_n}$**:=** $\mathbf{V_n}$]

to be the normal form of **M** under the environment [$\mathbf{v_0}$**:=** $\mathbf{V_0}$**,…,** $\mathbf{v_n}$**:=** $\mathbf{V_n}$].

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

Now let's 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 $\mathbf{N_0}$, …, $\mathbf{N_k}$, here **k** is 0 or a positive integer, such that **P** $\mathbf{N_0}$, …, **P** $\mathbf{N_k}$ are in database,
        - 3.2.1 If there is a $\mathbf{N_i}$ such that $\mathbf{N_i}$ == **Q'**, here **i** is a number between 0 and **k**, then the normal form of **M** is the normal form of **P** $\mathbf{N_i}$**:ENV**.
        - 3.2.2 If there is not a $\mathbf{N_i}$ such that $\mathbf{N_i}$ == **Q'**, here **i** is a number between 0 and **k**, find the list of variable terms $\mathbf{v_0}$, …, $\mathbf{v_l}$, here **l** is 0 or a positive integer, such that **P'** $\mathbf{v_0}$, …, **P'** $\mathbf{v_l}$ are in database,
            - 3.2.2.1 If there is a $\mathbf{v_j}$, 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 [$\mathbf{v_j}$**:=Q'**],
                - 3.2.2.1.1 If **P'** $\mathbf{v_j}$ has an assigner **U**, then the normal form of **M** is the normal form of the substitution **U:ENV'**.
                - 3.2.2.1.2 If **P'** $\mathbf{v_j}$ doesn't have an assigner, then **M:ENV'** is the normal form of **M**.
            - 3.2.2.2 If there is not a $\mathbf{v_j}$, 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 sequential term 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 see how the system processes sequential terms. For those who are not interested, the remaining part of this section can be skipped.

Given an assignment **M** = $\mathbf{N_1}$, $\mathbf{N_2}$, …, $\mathbf{N_n}$, its substitution under an environment **[**$\mathbf{v_0}$**:=**$\mathbf{V_0}$**;** …**,** $\mathbf{v_n}$**:=**$\mathbf{V_n}$**]**, syntactically $\mathbf{N_1}$, $\mathbf{N_2}$, …, $\mathbf{N_n}$: **[**$\mathbf{v_0}$**:=**$\mathbf{V_0}$**;** …**,** $\mathbf{v_n}$**:=**$\mathbf{V_n}$**]**, is the sequence of the following sub substitutions:

$\mathbf{N_1}$: **[**$\mathbf{v_0}$**:=**$\mathbf{V_0}$**;** …**,** $\mathbf{v_n}$**:=**$\mathbf{V_n}$**]**,

…

$\mathbf{N_n}$: **[**$\mathbf{v_0}$**:=**$\mathbf{V_0}$**;** …**,** $\mathbf{v_n}$**:=**$\mathbf{V_n}$**]**.

The normal form of the substitution ($\mathbf{N_1}$, $\mathbf{N_2}$, …, $\mathbf{N_n}$): $\mathbf{v_0}$**:=**$\mathbf{V_0}$**;** …**,** $\mathbf{v_n}$**:=**$\mathbf{V_n}$ is the sequence of the normal forms of the sub substitutions.

5 INFORMATION COMMUNITY

A community in our daily life is a geographical area in which people live together with families, interest groups, cultures, and laws. By information community, we mean that Froglingo provides a computing environment with built-in security facilities for individual users, business owners, and their customers. They share and collaborate in constructing information inside and outside of their organizations.

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, users 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.

5.1 User accounts

A user account is actually an assignee in database. More than an assignee, it has additional attributes such as password, 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 functionally dominates 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 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 syntax 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 one can come back to Froglingo again as the user account `www.myclienta.com` through the Windows CMD window:

```
C:\Froglingo\frog.exe
```

```
User id: www.myclienta.com
Passwd: ********
```

An old password for `www.myclienta.com` can be changed:

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

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 the 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 demonstrated how a user changes his/her own password by the command `passwd`, where the user has to provide an old password before a new password was requested. In a case that a user forgets its password, a super user can reset the password without providing the old password. Here is the syntax:

```
passwd useraccount;
```

For example, the user `//www.myclienta.com` can reset the password for the user `jone.dow` with the following command:

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

5.2 Sessions

A session is a period of time between which a user logins to a Froglingo system.

5.2.1 Establishment

When a new database starts, a user enters to a session by invoking the executable `frog.exe` at the Windows OS level. In this case, the user is the sole root user and can exit from and enter to a session without providing a password as many time as the user wants until a password for the root account is created.

As soon as a password is created for either the root account or a non-root account, users must provide the password before starting a session. A session is started via a CMD window as we have discussed in Section 5.1.

The last method to start a session is to execute the following command within an earlier session

```
login UserAccount "passwd";
```

where **`passwd`** is displayed in plain text and must be surrounded by a pair of double quotes and **`UserAccount`** is a user account. The new session is associated with a new user. For example the following command is executed by the root user and the root user is switched to

```
www.myclienta.com:
[//] 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 Section 7.

A session is always associated with two parameters. The first one is its user account representing the user who is interacting with the system. The second one is a context the session sets up for the user when the user interacts with the system. We call a context a stand, e.g., where the user stands in a database.

5.2.2 Signature

The built-in identifier `signature` is automatically assigned with the value of a user account when the user starts a new starts. For example, one via the user account `//` in a session can have the following interactions:

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

Since the term `signature` is automatically assigned with a value and not editable by anybody, it serves as the signature or the footprint of the user account.

A user account is always associated with a set of privileges. Through a user account, the system knows the privileges the user is assigned with and will either grant or reject the user's requests depending on the privileges. We will discuss privileges in Section 5.4.

5.2.3 Stand

The second parameter associated with a session is its stand, a context with which the user interacts with system. 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 user account `www.myclienta.com`, for example, both the signature and stand of the session are `www.myclienta.com`.

In a session, the stand changes when the user navigates a system by the command `cd`. The stand can be printed out by the command `whereami`. In the coming section 5.3, we will discuss how a user navigates.

5.3 Locality

A Froglingo database can host multiple individual users and business owners, where a user account may host multiple applications, i.e., terms and assignments functionally dependent on the user account. To effectively manage data and share data with others, a user may want to "travel" through database. When a user travels, the data appear differently to the user, and therefore the user is required to express the data differently. The key to support users' traveling is to consistently name the data when a user travels from one place to another. This section gives the command `cd` that changes session stands, some common stands users normally take, and the naming scheme that makes traveling possible.

When a mom speaks to her own kids and shouts out "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.

In parallel to the ways of our daily communications, we support relative names for data entities in databases, that is local to and unique around a stand. An expression being uniquely referencing a database entity under a stand 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 Navigation

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 likes to move to. The new stand can be an assignee in database as long as the user has a privilege to access (we will discuss privileges in the 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 Section 6) by creating 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 the stand to one of the subordinates and perform anticipated tasks:

```
[//www.myclienta.com] cd appl2;
[//www.myclienta.com appl2] create a b (c d) = 8;
[//www.myclienta.com appl2] print .;
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` to be discussed in Section 6.

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

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

5.3.2 Current and upper stands

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 5.3.1, 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:

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

```
[//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

After 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 $\mathbf{A_1}$... $\mathbf{A_n}$ under the stand **S** appears to be the expression **D** $\mathbf{A_1}$... $\mathbf{A_n}$, or equivalently (...(**D** $\mathbf{A_1}$)... $\mathbf{A_n}$). 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

After a user created an entity $\mathbf{A_1}$... $\mathbf{A_n}$ at the stand **S**, the terms **S** $\mathbf{A_1}$, **S** $\mathbf{A_1}$ $\mathbf{A_2}$, ..., **S** $\mathbf{A_1}$ $\mathbf{A_2}$ ... $\mathbf{A_n}$ 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 names

The names we have discussed so far are relative. An entity in database changes its relative name when a user changes the stand from one to another. There is a name for an entity that doesn't change all the time no matter what stand a user takes. A term having "//" as the inner-most term is an absolute name. Therefore, one without "//" as the inner-most term is a relative name.

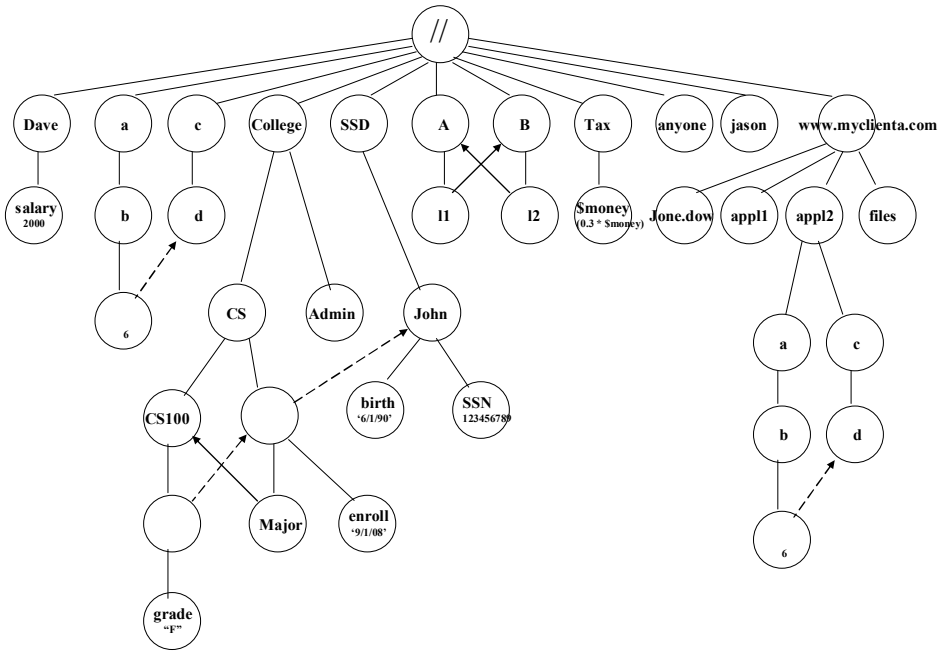
Given an assignee $\mathbf{A_1}$... $\mathbf{A_j}$ under an application home $\mathbf{S_0}$... $\mathbf{S_i}$ as the stand, the absolute name of the assignee is //$\mathbf{S_0}$... $\mathbf{S_i}$ $\mathbf{A_1}$... $\mathbf{A_j}$, or equivalently (... ((... (//$\mathbf{S_0}$) ... $\mathbf{S_i}$)$\mathbf{A_1}$)... $\mathbf{A_j}$). 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;
```

A stand a user takes is always displayed as absolute names.

To better understand database the naming scheme, we give a graphical view of the sample data we have created through this document so far. The graphical view also demonstrates how the Froglingo system uniformly manages data, user accounts, business logic, and files.



5.4 Privileges

In Froglingo, there are only three types of access privileges: administration, read-only, and no access. With the administration or read-only permission on a given entity, a user has the full controls or view only access on the entity and the entities functionally depending on the given entity. With no access on an entity, a user can do nothing about the entity.

By default, a user has the administration privileges to the entities functionally depending on the user account. Once a user account is created, the account can be suspended or deleted. A user has not access to his or her suspended account. A deleted account is physically removed from system.

By default, a user has no access to the entities not functionally depending on the user account. However a user can gain the administration or read-only permissions on entities if the owners of

the entities explicitly assign the privileges.

For the purpose of collaborations and information sharing, data entities also can be accessible to non-owner users. However it wouldn't happen until owners assign permissions to non-owner users. Before discussing them in detail, we give a few examples on how to use the two system calls `grtadm` and `grtacc`.

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.
```

When a data entity is made readable to non-owner users, we call it a service, i.e., the privileged users can retrieve and even update data entities below the service with the owner's wish. A data entity is called a partnership if it is created by a service call.

5.4.1 Administration privilege

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 under the administration privilege of the user. The administration privilege on an entity is meant that a user can perform all the operations available in Froglingo on the entity. These operations include create, record, update, delete, query operations and the user account management operations.

By default, the user possessing a user account has the administration privilege over the entities below the user account, i.e., functionally dependent. 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 **M** `{=+` **U** `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**, i.e., **M** `{=+` **U** `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 syntax is:

`grtadm` **M U**

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

5.4.2 Read-only privilege

When a user has the read-only 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 accessible to the user.

When a user has the administration privilege over an entity, it automatically has the read-only privilege over the entity and the entities below the given entity.

As stated in section 5.4.1, if **U** is a user account, and the entity **M** is not one below **U**, that is, **M** {=+ **U** and **U** != **M** doesn't hold true, then **U** has no privilege at all over **M**. To assign **U** to have the read-only privilege over **M**, the operation `grtacc` has to be used. The syntax 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 read-only 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 read-only privilege on `AA Associate`. The example assumes that each user account under `[//www.myclienta.com]` have 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 services. Not only for the purpose of viewing data, services 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 additional 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. Each row of the output is in the following format[2]:

```
//grpRoot entity users = true | false
```

Here `//grpRoot` always presents, and **entity** is a term, the object to be accessible to user account(s) **users**. When a row is associated with `true` or `false`, **users** is assigned with the

[2] Actually each row is a term, a native expression in the EP data model, in this release. The format can be improved in a future release.

administration or the read-only privilege respectfully. 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, the `anyone` user assigned with read-only permission was automatically converted to a variable having all the user accounts of database in the range.

The second command is to remove privileges. The privileges to be removable are those added by `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 data entity associated with an assigned privilege and **`users`** is the user or the variable representing a group of users who were assigned with a privilege to access **`entity`**. 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 row of the privilege list 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 Service

When an entity is granted a user with the read-only privilege, we call the entity an service. An service not only offers a way of retrieving information, it also can grant a permission to update data entities that are functionally dependent on the service. The update could happen even if a user triggering the action doesn't have the administration privilege. We give a sample application in simulating a lottery draw: 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 does 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 read-only privilege (and therefore was called a service).

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

5.4.5 Partnership

Through a service, 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 read-only 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 service:

```
[//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;
```

where the number `1243877000` is the integer for a timestamp. 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;
```

In the sample above, the plus-owner 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 service. A club, as an example, accepts new memberships. The only requirement for a person to become a member is to accept an agreement by providing his/her signature. Assume the owner `www.myclienta.com` establishes a club:

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

The user `jason` joins the club by calling the function `accept_mem` with his agreement, i.e., providing his signature:

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

With the identifier `signature`, the system understands that the user accepts the agreement that his identity will be verified and recorded. If the user provides his user account `jason`, or any

string other than `signature`, he would not be able to join the club.

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

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

A partnership can be temporarily suspended by a system command `susparti`. To suspend Dave's membership in the example above, we can issue a command:

```
[//www.myclienta.com club] susparti member (// jason);
```

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:

`actiparti` **`partnership`**

Note that the two commands can be performed only by the plus-owner. To permanently remove a partnership, either the plus-owner or the minus-owner can use the `delete` command.

6 FILE MANAGEMENT

When we talk about information today, file is always part of it. Froglingo manages three types of files:
1. HTML/XML files. They are composed in an editor at operating system level and then uploaded to Froglingo databases. The files can be Froglingo services, i.e., HTML/XML files containing Froglingo expressions, which would be rendered to web pages upon calls.
2. Froglingo script files. They contain Froglingo expressions, mostly assignments. They are composed in an editor at operating system level and uploaded to Froglingo databases.
3. Binary files. They are those not recognizable in Froglingo, They could be image, audio, and other textual files. Froglingo treats them all as binary files.

In this section, we introduce commands that 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, a XML parser places more restricted rules than a HTML parser does.

Like a HTML parser, Froglingo accepts XML and HTML files regardless the quality of file contents.

6.1 File 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 files or folders should be parsed:

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. If it does contain Froglingo expressions and at the same contains syntactical errors, the

	system will reject the file and report the errors.
`html`	A HTML document. The system treats it as a XML document.
`folder`	A folder at operating system 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;
```

When a file is uploaded, it is stored as a set of assignments. When a binary file is uploaded, it is stored as one assignment, where the file name is the assignee and the content of the file is the assigner. After the upload operation of the file `text.txt` above, for example, you can see that the file is stored as an assignment:

```
[//] 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 the assignments under the given entity. The command `ls` lists the assignees only. The command `ls` helps to avoid assignees with large size of binary streams to be displayed.

Uploading XML/HTML files as services is discussed in Section 7.

6.2 File download

A file in Froglingo databases can be downloaded back to a file at operating system level, displayed as an image, or rendered as a web page. A set of assignments in databases can be downloaded to a file at operating system. All these operations are performed by the the command `print` with 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 by the command `print`. When the optional parameter (`to` **`path`**) is provided, it is the path (including the file name) of a file to be generated at operating system level. If the parameter (`to` **`path`**) is not provided, the data entities related to **`term`** will be displayed on the CMD window or on web browser depending on where requests come from. When a request comes from a web browser, the 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:

Format	Description
`stream`	It is only for a term **`term`** which is the file name of an ASCII stream file. The command retrieves the entire stream and wraps it as a file.
`frog`	The command retrieves all the assignments where the assignees are functionally dependent on **`term`**. A string as a sub term in a assignment is printed out without a pair of double-quotes surrounding the string.
`backup`	It works as if it was the format `frog`. The only difference is that a string as a sub term in a assignment is printed out with a pair of double-quotes surrounding it.
`xml`	The command attempts to generate a XML file. When **`term`** is the name of a file that was uploaded earlier and the file has no Froglingo expressions embedded, it will simply print out the original file. For a xml file that embeds Froglingo expressions, see Section 7 on how a xml file is downloaded.
`html`	The system will act as if the format was xml.

When **`term`** is the name of a file uploaded before, the parameter **`format`** doesn't 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.

6.3 OS path

In sections 6.1 and 6.2, a precise format was not provided for the parameter **`path`** at operating

system 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 folder names followed by a file name, that are delimited by either "/" or "\". A path may also be proceeded with a hard drive name. In other words, the syntax of file path at Windows is:

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

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

Here the number **n** can be 0 or a positive integer. However, the delimiter "\"is not a symbol in Froglingo, or is interpreted as the escape character of special ASCII codes in a string in Froglingo. The delimiter "/" is the arithmetic operator division in Froglingo. Therefore, we express paths at operating system level differently, that is,

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

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

In addition, we require that each folder or file name **name**$_i$ must not include special characters space ' ' and dash '-'. Here are a few valid sampe paths in Froglingo:

```
[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";
```

A special character '*', serving as a wildcard, can be the last segment of a path, i.e., **name**$_n$. As a result, the following two paths have the same effect:

"[driver:/]name$_0$/**name**$_1$…/**name**$_{n-1}$**"**, and

"[driver:/]name$_0$/**name**$_1$…/**name**$_{n-1}$**/*"**.

7 ACCESS OVER THE INTERNET

Froglingo can be configured to work as a web server. With the function of web servers, users can interact with Froglingo via web browsers across the Internet. In Section 6, we have already talked about how a pure html/xml file is uploaded to and downloaded from a Froglingo database before rendered as a web page on a web browser. In this section, we discuss how Froglingo expressions can be embedded into html (Hypertext Marker Language) files such that an embedded html file can be filled up with Froglingo data entities before rendered to a web page. As a result, Froglingo supports web-based user interfaces for applications hosted in Froglingo and for administration work over the Internet.

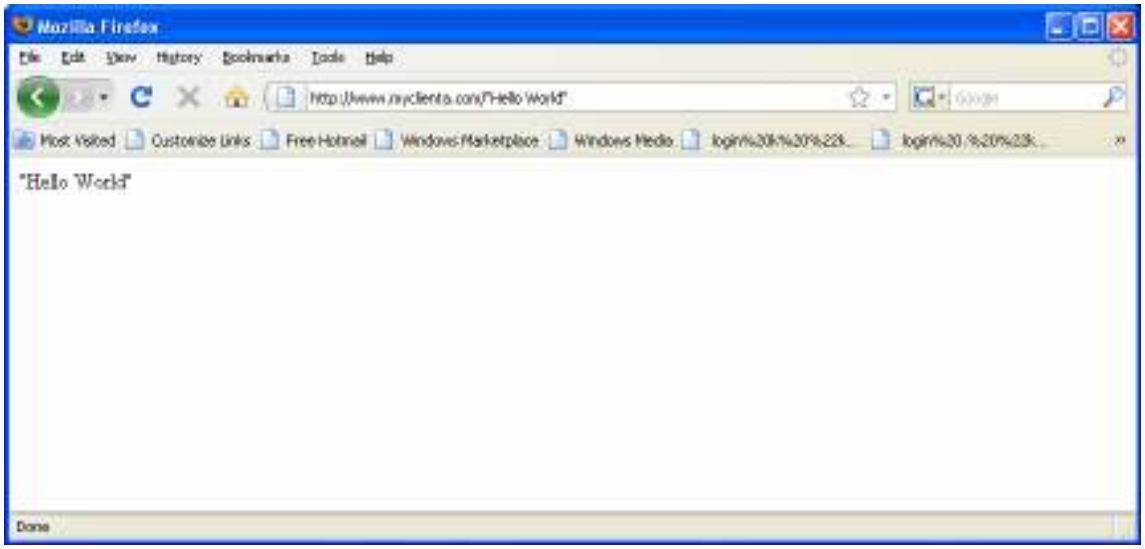
In this section, we discuss how Froglingo is configured to be a web server and how Froglingo expressions are embeded in both URIs and HTML files such that Froglingo would understand requests from web browsers and further the data from Froglingo database can be plugged into on web pages.

7.1 Web server setup

To run Froglingo as a web server, there are two things to be done:

1. We need to ensure that a network is set up. If it is not ready yet, here are the two ways to get it ready:
   a. If it is only for testing purpose, the network could be in a computer itself, where both a Froglingo system and a web browser are located. The only thing we need to do to set up the network is to add a domain name, in correspondence with a user account in Froglingo, to the system file of the Windows operating system:
      "C:\WINDOWS\system32\drivers\etc\hosts". A sample entry in the file would be::
      ```
      127.0.0.1        www.myclienta.com
      ```
   b. To make the application available on the Internet, a physical network needs to be 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. Launch Froglingo 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 a Froglingo database. It also serves as the Internet domain name of the application.

After Froglingo has been configured as a web server, one can enter a URI, e.g., `http://www.myclienta.com/"Hello World"`, in the address field of a web browser. See a screen shot below showing what a web browser obtained from the server with the domain name `www.myclienta.com`.



7.2 URI

Users via web browser use URI, Universal Resource Identifier to communicate with web servers. Froglingo parses URIs and converts them to Froglingo expressions. A simple form of the URIs used by Froglingo is:
`http://dn[:port][/t0.../tn-1/tn][/]`

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 field $/t_0.../t_{n-1}/t_n$ 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 (possibly 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 is in correspondence with a 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 two rows states that a default HTML file `index.html` is assumed when there is no specific request given in a URI. The third row states that if the last expression of a URI is a html file name, the URI is interpreted as to retrieve the html file. In the last row of the table above, where a URI is ended with an expression, instead of a html file name or '/', the URI is interpreted as a request t_n with a stand `[//dn` (t_0) … (t_{n-1})`]`. 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, i.e., an 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 their escaped forms:

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

For example, the Froglingo expression `$x:[$x > 0]` must be converted to `$x%3A[$x > 0]`. The form `$%78%3A[$x >%200]` is acceptable as well because the characters 'x' and ' ' have the 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 entities that are functionally dependent on the user account and are not accessible to everyone publicly. 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. But it is sufficient to fully interact with Froglingo server with the exception that users cannot use the root account to communicate with web servers.

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) of the expressions 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 web pages. Here we treat both HTML and XML equally and call them XML document.

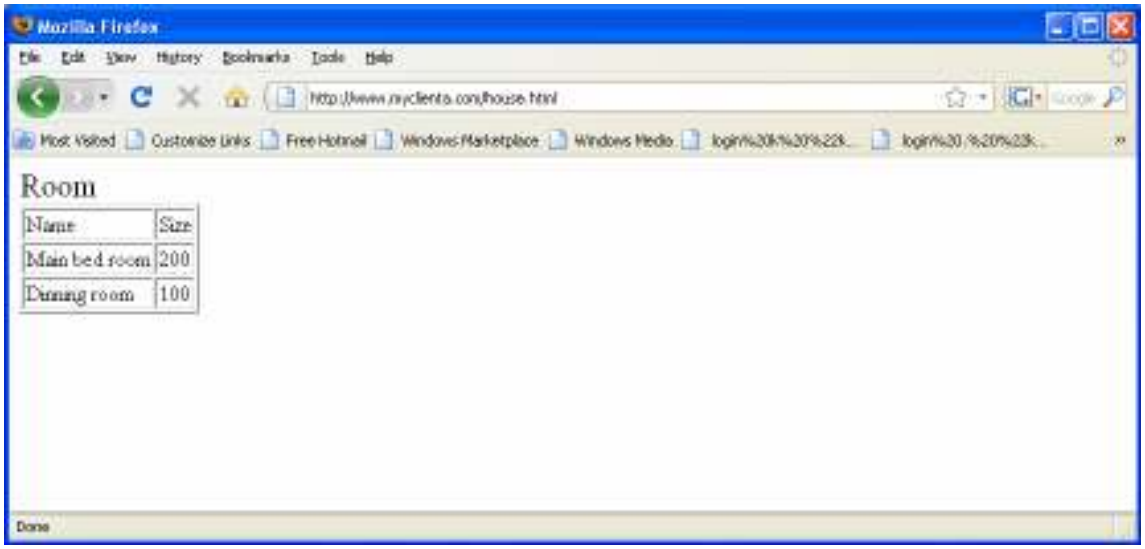
The basic structure of a XML document is a set of elements (or called blocks). Each element starts with an open tag and mostly ends with a close tag. An open tag is made up of a tag name, sometime followed by an optional list of attributes, all of which appears between angle brackets < >. The end tag of an element contains the same name as the one 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 Documents

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 at a operating system level where a Froglingo database server 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, commonly available web browsers 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 whether or not a document contains errors. When a document contains error, Froglingo reports the errors while it parses and stores the document. When a file embeds Froglingo expressions, Froglingo does a stronger syntax checking and stores the document only if there is no syntactical error.

Starting from the next section 7.3.2, we introduce a set of Froglingo-special tags and attributes through which embedded Froglingo expressions can be replaced with XML texts.

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;
```

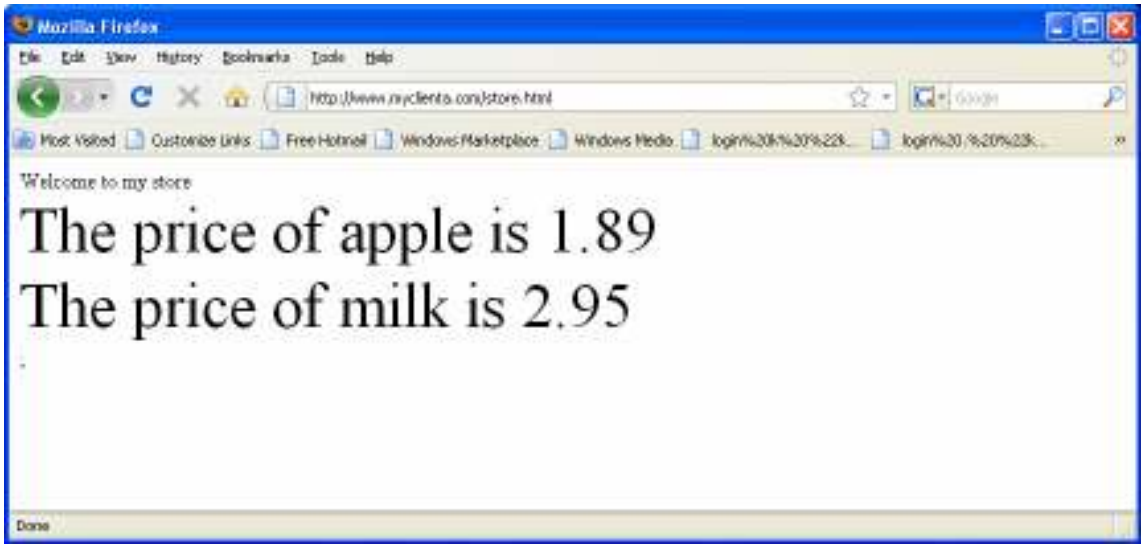
We grant the file accessible to `everyone`:

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

When a user print out the file locally via a CMD window, the output would have the terms embedded in the store.html file replaced with their normal forms:

```
[//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 user accesses 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 in a Froglingo database, 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>;
```

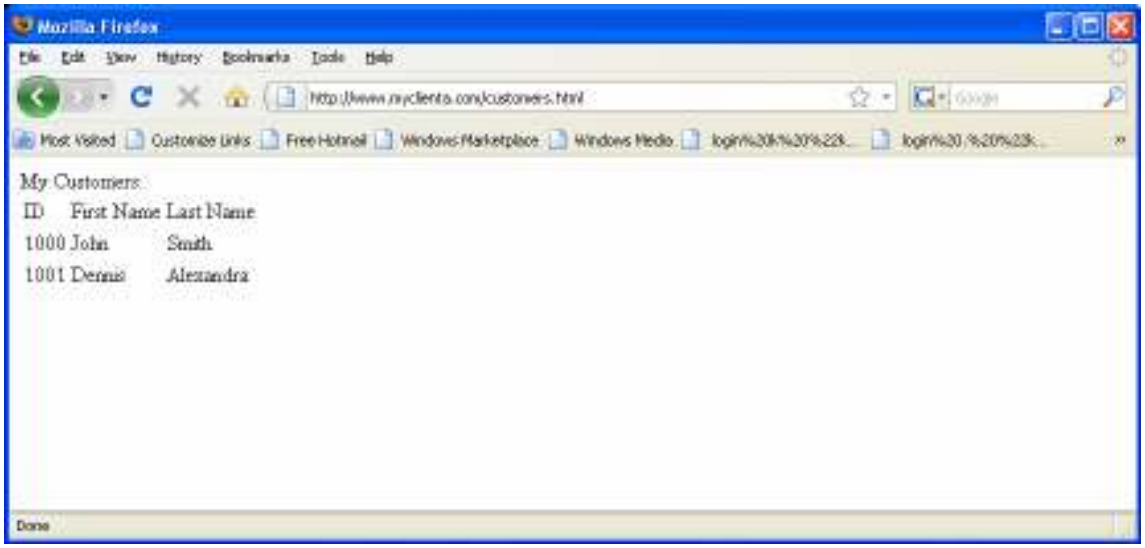
As another example, we upload another file customers.html that includes a <frog> block embedding select operation:

```
<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 attribute name and value pairs in a tag form: <**tagname** … frog**:attr**=**qtermq** …>

Here **tagname** is the tag name of a block, **attr** is an attribute name in html, and qtermq is a sequence of characters. Froglingo 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 a 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 may 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 7.3.2, 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:

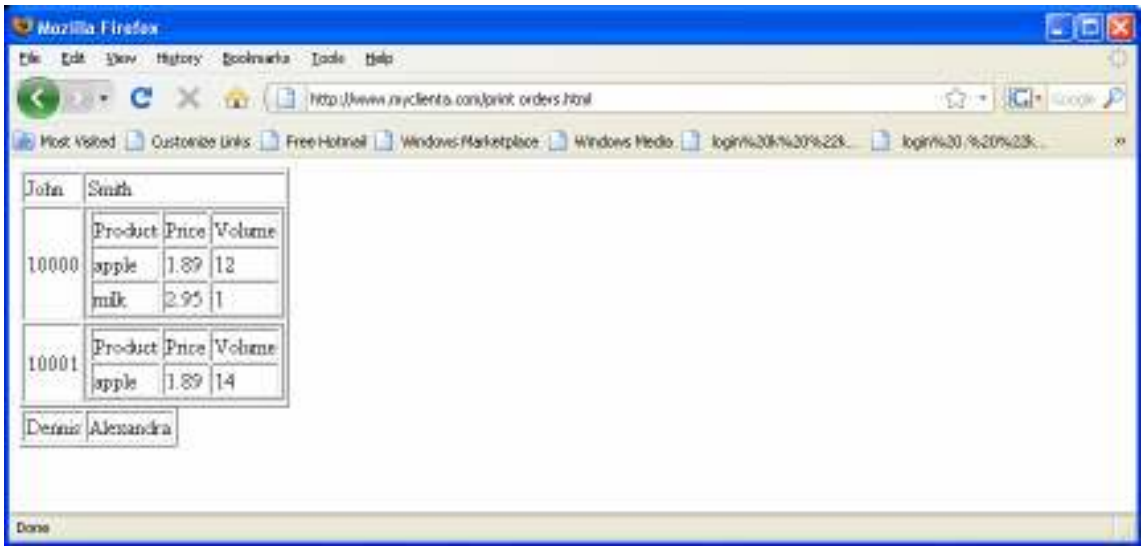
```
[www.myclienta.com] print customers.html;
<html>
      <body>
            My Customers: <br>
            <table>
                  <tr>
                        <td>1000</td>
                        <td>John</td>
                        <td>Smith</td></tr>
                  <tr>
                 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

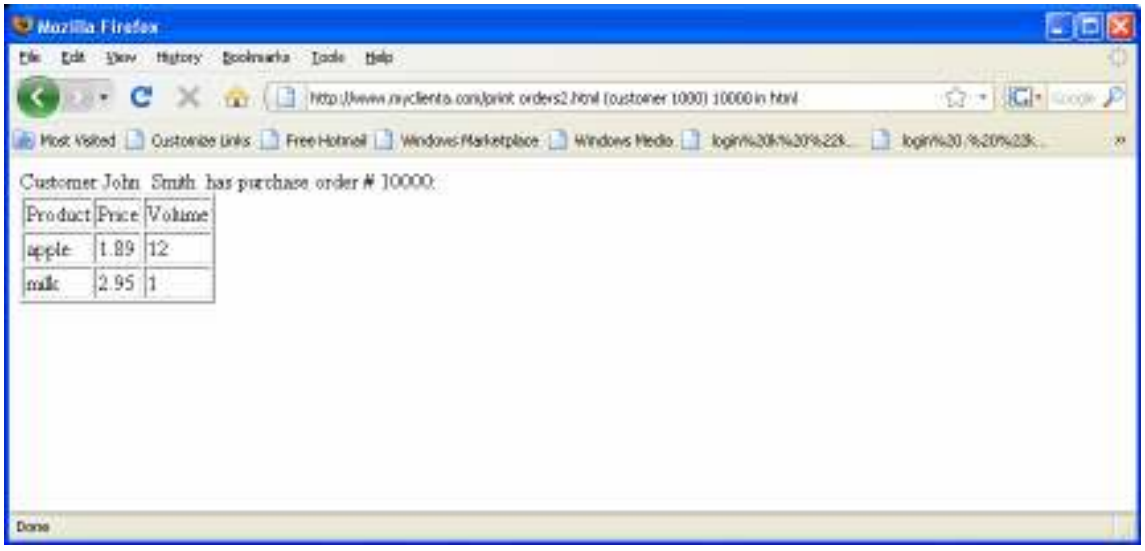
In Froglingo, one can pass parameters into a XML file. It is done via variables pre-inserted in XML file. When XML files containing variables are downloaded, values can be passed to the files through the variables such that the XML files are instantiated correspondingly.

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-specific 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 a proper privilege can view the following page by providing the URI:
`http://www.myclienta.com/orders2.html (customer 1000) 10000 in html`



7.4 Requests via HTML forms

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, which are the carriers of user requests entered through HTML forms with method "GET". In this section, we also 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://**dn[:port][/exp$_0$.../exp$_{n-1}$/exp$_n$][/]**

It is extended as
http://**dn[: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 **a$_1$, …, a$_m$**, prefixed with character '@', may appear in **exp$_n$** serving as variables. When Froglingo server receives an URI, it will replace all the instances of the variables in **exp$_n$** with the corresponding values **"v$_1$", …, "v$_m$"**.

7.4.2 HTML form

Assume that the owner of the account `www.myclienta.com` wants to collect customer information. The owner 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:

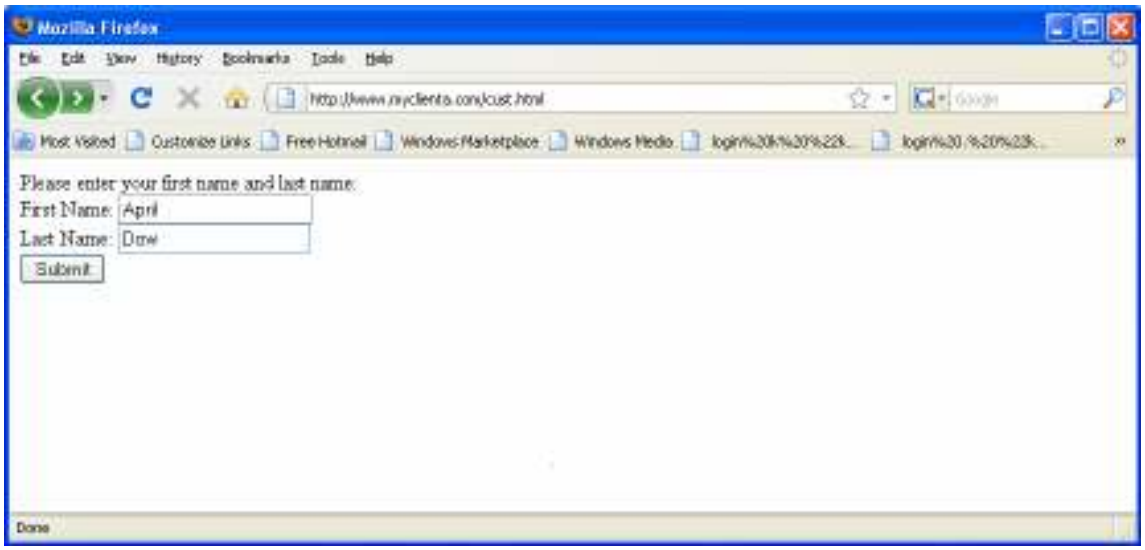
```
[//www.myclienta.com] cust_index;
1003;
```

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

```
<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="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:



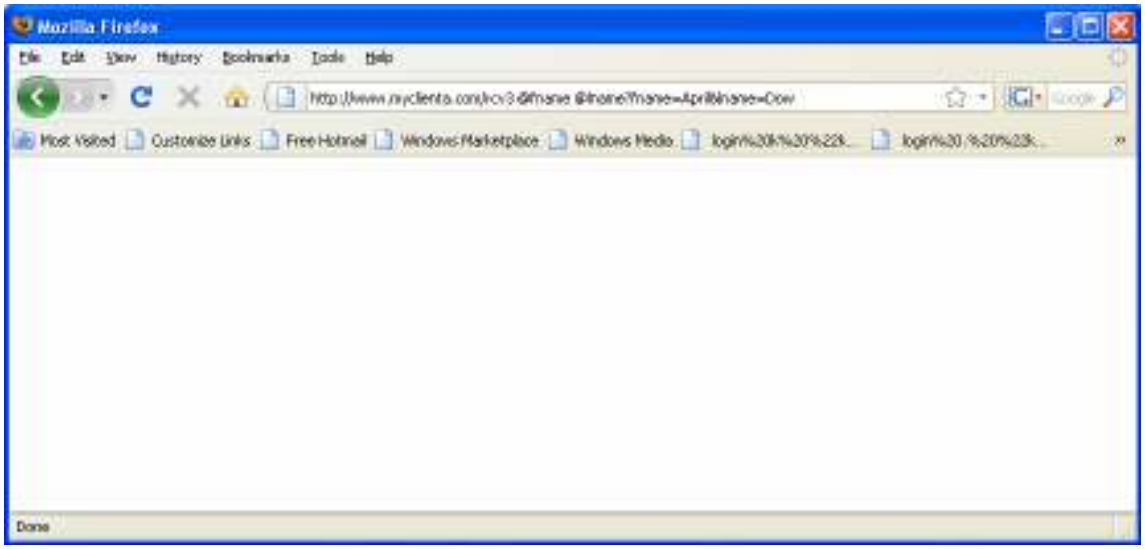
After the web user entered her first and last names and clicked "Submit" button, the request goes to the web server. The URI arrived at Froglingo is:

```
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 is 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 File upload via web browser

Files can also be uploaded to Froglingo databases by 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));
```

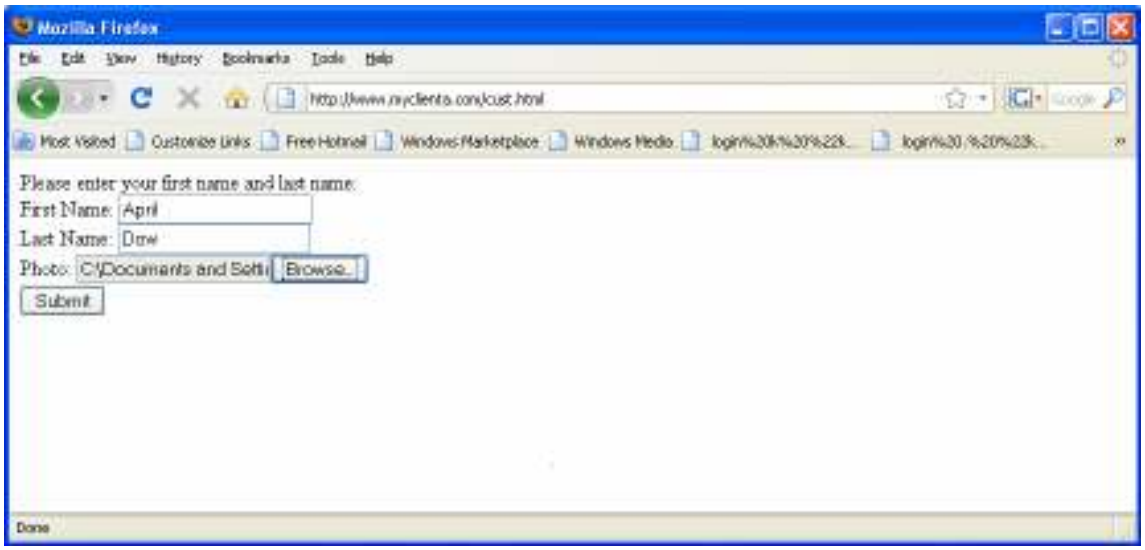
The term `cust_index` was created earlier:

```
[//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 will be able to enter the first name and last name, and upload a (image) file (see the following web page).



8 MISCELLANEOUS FEATURES

Froglingo is untyped, i.e., no user-defined types are necessary. However, it has finitely many basic data types, e.g., integers, strings, Booleans. We have given an example of using the operator `isa` and the data type `integer` in Section 5.4.5. In this section, we give a full coverage about the operator `isa`, a few other basic data types, and special values. At the end of this section, we introduce administrative tools for Froglingo database backup and user activity audit trials.

8.1 Basic data types and the membership operator

So far, we have introduced integers, real numbers, strings, and files. In corresponding to the different categories of the constants, the special constant terms `integer`, `real`, `string`, and `stream` are introduced as basic 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;
```

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.

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 the next line. This kind of interactions is not sufficient when a user accesses databases via web browsers because the prompt sign is not returned. End users may want to see a confirmation after they submitted update requests via web browsers. A developer can satisfy the end users' need by utilizing the special identifier `void`.

In Froglingo, the special identifier `void` has a special normal form: empty, i.e., no return messages from Froglingo. An update operation, after 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 and visible at all.

We interpret `void` to have empty as the normal form because developers can use it to detect the return status of an update operation. Based on the return status, developers can generate text

message to inform end users the status of the update operation. 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 code:

```
[//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 operation, as we have seen in the earlier sections, always returns a system error message. It is an ideal presentation for developers who have knowledge of Froglingo, but a presentation for end users who don't understand the error messages. Here we introduce another special identifier `error`.

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

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

The only exception is when an operation fails and returns an error message. At this moment the identifier `error` takes the error message as its normal form. In the earlier sections, we saw error messages rather than `error` because `error` is further reduced to the error messages. With the special identifier `error`, developers are able to display application-oriented error message instead of system error messages. Here is an alternative code for the sample application described in section 8.2:

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

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

8.5 Date and time

In this release, Froglingo supports date and time in the format:

'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 needs to use a 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 system time when `timestamp` is called. 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 database file has the default file name `.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 in this release. But the batch load tools `print` and `load` can be used to back up data and to restore data as discussed in Section 6.

8.6 Log files

Froglingo generates 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, may be followed with a system/error message indicating the result of the request.

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 discussed in Sections 1 to 4 was implemented by November 2004 (Release 0.1); and has been applied to many experimental applications.

Release 1.0, in March 2009, supports information community (data access controls), file management, and access over the Internet.

Release 2.0, in March 2013, included some bug fixes.

More features including multithreads need to be supported in the future.

APPENDIX B: Grammar

The grammar listed here is simplified. Hope it gives readers another view to better understand 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’ | ‘<’ | == | != | … | ‘{=+’ | ‘<-‘ | …
;
```