# Lazy User's Manual and Reference

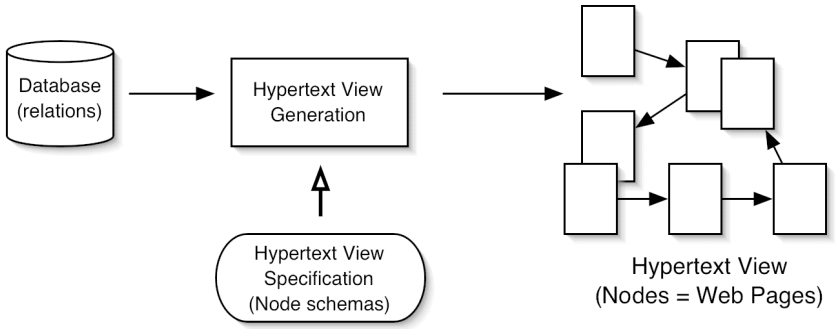
G. Falquet
Université de Genève, Centre universitaire d'informatique

March 2005

## Table of contents

# 1 Introduction

Lazy is a language and system to publish databases on the Web and to create Web application interfaces. In other words, Lazy creates an active hypertext view of a database. This hypertext view is a set of nodes (the Web pages) and hyperlinks that represent the contents of the database, and a set of associated actions to update the database. In the declarative approach, the hypertext components (nodes and links) are derived from the database content (relation tuples) according to a hypertext view specification, as shown below.




A hypertext view specification consists of a set of **node schemas**. Every node (Web page) is an instance of a node schema written in the Lazy specification language. Node instances are dynamically generated by a **node server** that takes as input a compiled formed of the node schemas.

A node schema specifies: the table(s) from which the node's content is to be drawn, the selection and ordering criteria, the elements that form the node content, and links to other nodes. The node definition language is described in sections 4, 6, and 9 of this manual. The node schemas must be compiled in order to be usable by the node server. The compilation process first checks the schema's syntax, then translates it into a form (in fact SQL queries) that is directly executable by the node server. It is described in section 3.

# 2 Installation

## 2.1 Requirements

Since Lazy is entirely written in Java you need a Java 1.4 run-time environment installed on your machine (you can get it from Java soft http://java.sun.com). Lazy does not work with Java 1.3 or earlier versions because it uses the new Java encryption scheme.
The Lazy node server is a servlet application. So you need a servlet container (server). You can either use the tomcat servlet container that comes with the Lazy distribution or download a more recent version from (http://jakarta.apache.org/).

## 2.2 Downloading and installing

Download the zip file from http://cui.unige.ch/isi/lazy4/download

1. Unzip or gunzip the file into some directory. This should create a new directory named `lazy-`*x*`.`*y* (*x*`.`*y* is the version number) that contains the following subdirectories :

| | |
|---|---|
| `admin` | Lazy source code of the interactive development and administration hyperspace |
| `bin` | scripts to start and manage the Lazy system |
| doc | documentation |
| `examples` | example hyperspaces |
| `hsqldb` | the hsql database engine (http://hsqldb.sourceforge.net/ ), with a `demo` directory that contains the data of the example Web site. |
| `src` | the Lazy source code |
| `tomcat` | (part of) the tomcat servlet container from the Apache Jakartaproject (http://jakarta.apache.org/)<br>webapps/lazy contains the lazy web application (including the lazy node server servlet) |

NOTE. If you already have a tomcat server on your machine, copy `tomcat/webapps/lazy` to the `webapps` directory of your tomcat server and restart tomcat (if required).

## 2.3 Starting the Lazy system with the example application

Lazy comes with an example web application for a virtual museum. It is comprised of a small database that stores information about works, artists, exhibition, etc. and a set of hypertext nodes (dynamically generated Web pages) to navigate and update the museum.

### Step 1. Set environment variables

Before running the Lazy system on the example database, you must define the following environment variables:

| | |
|---|---|
| `LAZY_HOME`: | the installation directory of your Lazy system (= the full path of your lazy-x.y directory) |
| `JAVA_HOME`: | your java installation home |

`TOMCAT_HOME`: the home directory of your Tomcat server (set it to `$LAZY_HOME/tomcat` to use the server that comes with Lazy)

Then you must run a `lazyenv` script that sets additional variables that depend on these ones.

**Unix/Linux/MacOSX with bourne shell or bash**

Add the following lines to your .login or .bash_profile:

```
export LAZY_HOME=your lazy home directory
export TOMCAT_HOME=$LAZY_HOME/tomcat
       ### or your own tomcat home
export JAVA_HOME=your java installation directory
source $LAZY_HOME/bin/lazyenv.sh
```

**Unix/Linux/MacOSX with cshell**

Place the following lines in your .login or .bash_profile:

```
export LAZY_HOME=your lazy home directory
export TOMCAT_HOME=$LAZY_HOME/tomcat
       ### or your own tomcat home
export JAVA_HOME=your java installation directory
source $LAZY_HOME/bin/lazyenv.sh
```

**Windows NT/2000/XP**

Place the following lines in your autoexec.bat
or define these variables in the **System > Advanced > Variables settings**.

```
set LAZY_HOME=your lazy home directory
    ### for instance C:\lazy-4.3
set TOMCAT_HOME=%LAZY_HOME%\tomcat
    ### or your own tomcat home
set JAVA_HOME=your java installation directory
    ### for instance C:\jsdk1.4.1
run %LAZY_HOME%\bin\lazyenv.bat
```

**Caution.** Avoid white spaces and other special characters in these variables, they will probably cause errors when starting the tomcat server.

## Step 2. Start the database server

Open a new shell/terminal/command window and make sure the environment variables are correctly set. Then type

**Unix/Linux/MacOSX**

```
% runServer.sh
```

**Windows NT/2000/XP**

```
C:> cd %LAZY_HOME%\bin
C:> lazyenv
C:> runServer
```

(do not close the command window!)

### Step3. Start the node server (with the provided tomcat servlet container)

Open a new shell/terminal/command window and make sure the environment variables are correctly set. Then type

**Unix/Linux/MacOSX**

```
% startns.sh
```

**Windows NT/2000/XP**

```
C:> cd %LAZY_HOME%\bin
C:> lazyenv
C:> startns
```

(do not close the command window!)

This starts the tomcat servlet container, which is a HTTP server that manages servlets. The node server is a servlet (named lazy/ns).

### Step 4. Start browsing the virtual museum

With your preferred Web browser, open the example home page at `http://127.0.0.1:8080/lazy` and follow the links to the generated nodes (if your Windows machine is not connected to a network, it may take some time before it realizes that it can connect to itself, sometimes more than 1 min., you can also try to stop and restart loading).

You can also choose to explore the Lazy system by following the *Lazy Administration* link. In this case you'll need to log in as a Lazy administrator (user = admin, password = x).
If you already use ports 8080 and 8007 for another purpose, you can select other ports in the tomcat configuration file $TOMCAT_HOME/conf/server.xml

### Step 5. Stop the node server

Open a new shell/terminal/command window and make sure the environment variables are correctly set. Then type

**Unix/Linux/MacOSX**

```
% stopns.sh
```

**Windows NT/2000/XP**

```
C:> cd %LAZY_HOME%\bin
C:> lazyenv
C:> stopns
```

# 3 Using the Node Compiler

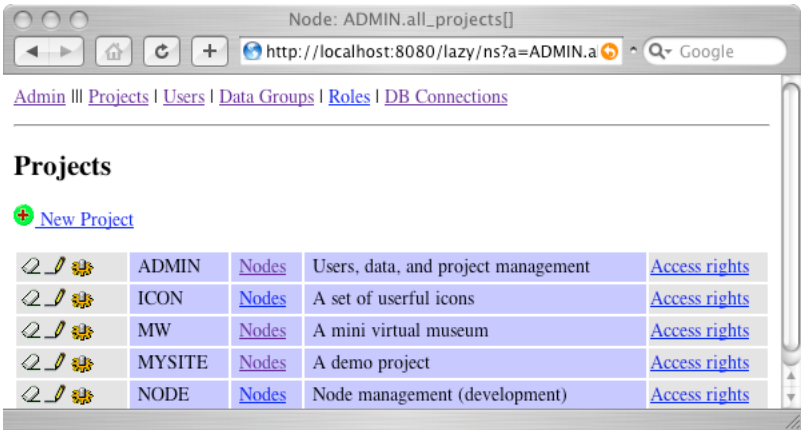
The node compiler checks the syntax of a node schema and translates it into a form that is directly executable by the node server. It can be used either from the Lazy interactive development environment, or in batch mode from a terminal/command window.

The following paragraphs show how to define and compile a new node for the virtual museum example, both interactively and in batch.

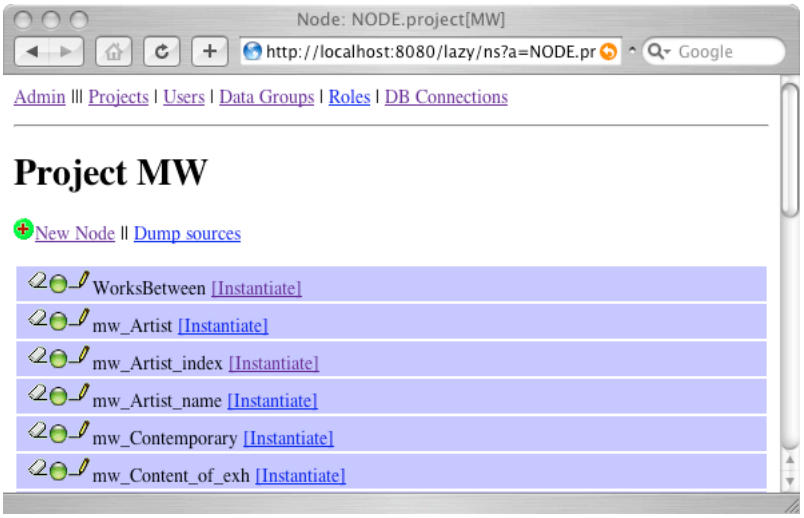
## 3.1 With the interactive development environment

The interactive development environment can be reached from the Lazy start page (`http://127.0.0.01:8080/lazy`) by following the *Lazy Administration* link. The Lazy administration page has links to projects, users, database connections, etc.

Navigate to the *Projects* page to see all the currently *defined* projects. At this point the Lazy server may ask for a username and password, by default user `ADMIN` has password `x`.

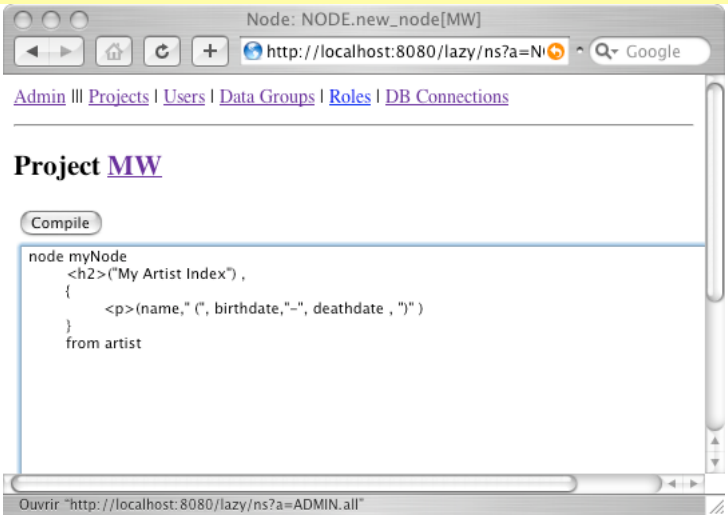



To define a new node in the virtual museum, follow the *Nodes* link of the *MW* project. It leads to the list of node schemas of the *MW* project.



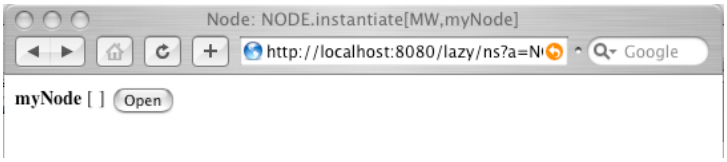
Click the *New Node* link to create a new node, this opens a node input page. In the text field type the following node schema text:

```
node myNode
    <h2>("My Artist Index") ,
    {
        <p>(name," (", birthdate,"-", deathdate , ")" )
    }
    from artist
```

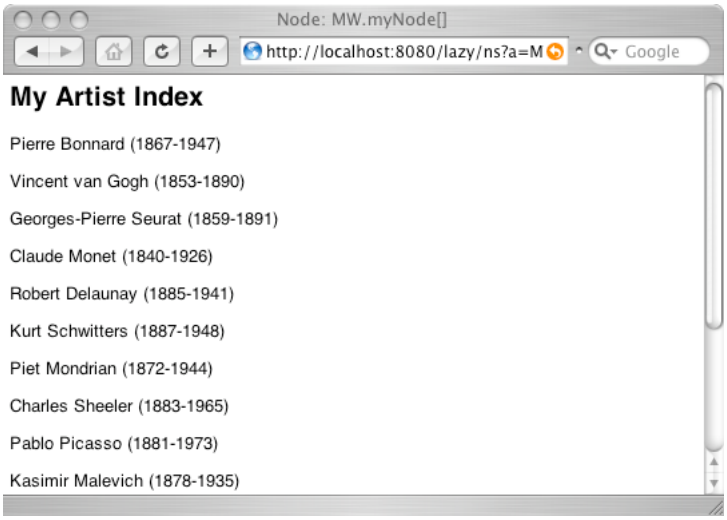



Click the Compile button to save and compile the node and return to the list of node schemas.

Now there is a *myNode* line in the node list with either a red or green ball. The red ball indicates a compilation error. In this case click the pencil icon to edit the node. Once the ball is green click the *Instantiate* link




and the the *Open* button to generate a Web page from this new schema.




## 3.2 In batch mode

Create a new text file (say `myNode.lzy`) with the following content :

```
define
project MW
    node myNode
    <h2>("My Artist Index") ,
    {
        <p>(name," (", birthdate,"-", deathdate , ")" )
    }
    from artist
end
```

A source file must begin with 'define' and end with 'end', it may contain one or more node schemas. The 'project MW' statement indicates that this node belongs to the MW project.

Open a new terminal/command window and make sure the environment variables are correctly set and that the `lazyenv` script has been executed (as indicated in the installation instructions under "Start the node server"). In necessary, start the database server and the node server (the node compiler stores the compilation result (an executable form of the node schema) into the database).

Compile and install your node schema by typing (Unix or Windows)

```
lc myNode.lzy
```

Once you get an error free compilation, instantiate a node by sending the following URL to the node server: `http://127.0.0.1:8080/lazy/ns?a=myNode`. It should display a list of artists together with their birthdate and deathdate.

The LAZY system is a dynamic system, once a node definition has been modified and recompiled, the new version is immediately available to the clients (there is no site generation phase). Try modifying myNode, recompiling it and reloading the corresponding page into your browser.

IMPORTANT NOTE. To speed up complex node generation, the Lazy node server maintains a server-side node cache. When a node schema is recompiled all its instances must be erased from the cache. However, in the current version, the batch compiler does not clear the cache. This must be done manually from the Lazy administration page by clicking the *Clear node cache* link.

# 4 Introduction to the Node Schema Language (part I)

This section presents the basic elements of the Lazy node schema language. The examples are based on a simple database that stores information about a digital museum. The database schema is as follows ((key attributes are in bold):

Work (**wno**, author, title, c_date, height, width, picture)
Artist (**ano**, name, birthdate, deathdate)
Exhibition (**exno**, title, desc, organizer)
Museum (**mno**, name, location, URL)
Ownership (**work**, **owner**, acquisition)
Art_cnty (**artist**, **country**, activity)
Ex_content (**work**, **exhibition**, comment)

## 4.1 Principles: node schemas and nodes instances

The definition of a basic node schema takes the following form:

```
node node-name [parameter-name, ...]
    { content-specification }
from tables selected by condition
```

An instance of this schema (an actual hypertext node) is obtained as follows:

1. select the tuples of the specified table(s) that satisfy the given *condition*
2. for each selected tuple, generate a content according to the *content-specification*.

For example, a node generated from the schema

```
node WorksBetween[d1, d2]
    { title, height, width }
from work selected by d1 <= c_date and c_date <= d2
```

will contain the title, height, and width of all the works of art created between the dates *d1* and *d2* given as parameters. The actual content of a node instance depends of course on the parameter values and on the current database contents.

## 4.2 Specifying the contents of a node

### 4.2.1 Content elements

The content of a node is specified by a list of elements. An element is either, a database attribute, or a quoted string, or a number literal, or a parameter name, or an artithmetic expression. An element or a sequence of elements usually has a markup type and attributes (an HTML or XML tag), defined with the following syntax:

*<tag attribute=expression ...>( element , ...)*

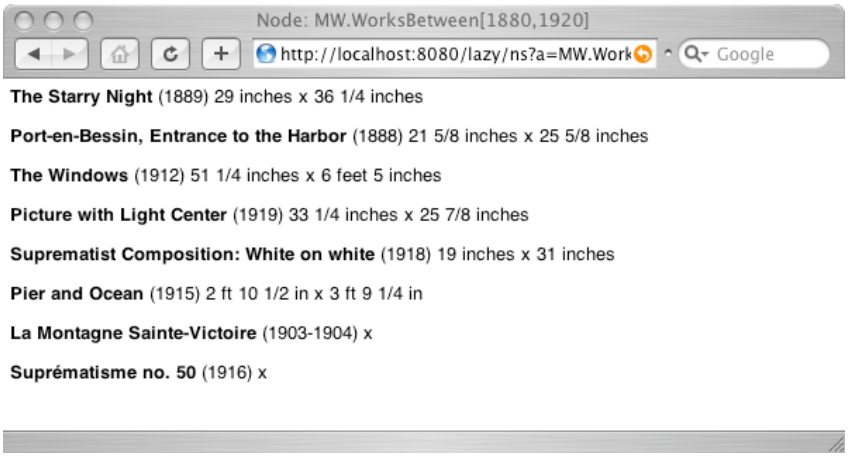
This notation is an abbreviation of the standard HTML/XML notation

*<tag attribute=value ...> element ... </tag >.*

We can now refine our previous example by adding HTML tags and white spaces or texts to produce a niceer HTML document:

```
node WorksBetween[d1, d2]
    { <p>(<b>(title), " (", c_date, ") ", height, " x ", width) }
from work selected by d1 <= c_date and c_date <= d2
```

You can try compiling and instantiating this node, as indicated in section 3. The following screenshot shows the instance `WorksBetween[1880, 1920]` on the example database.




### 4.2.2 Tuple and non-tuple elements

While the content elements placed between { and } are generated for each selected tuple, the elements outside or enclosing the { … } are generated only once. They generally serve as titles, header, or footers that appear only once in a node. In addition, these elements may contain aggregate functions like sum(), max(), min(), count().

**Example.** The purpose of the following node schema is to display a list of artists born after a given date *d*.

```
node mw_Artist_after[d]
  <h2>( "Artists born after ", d ) ,
  <ul>(
     {<li>(name," (born ",birthdate,") ") }
  )
from artist
    selected by birthdate >= d order by name
```

An instance of this node will contain a title (the <h2> element) followed by an unnumbered list (<ul>) of artists. A typical instance of this schema (with *d* = 1880) looks like


The non-tuple elements may also contain the usual aggregate functions sum, avg, count, min, and max, applied to table attributes.

Note that if the selection expression does not yield any tuple, the node will be totally empty. Even the non-tuple elements are not generated. This property is sometimes useful when one wants to display something only if a given condition is satisfied.

### 4.2.3 Selection conditions

The selection condition is a boolean expression made of literal constants (character strings or numbers), attribute names, parameter names, arithmetic and logical operators, comparison operators, or function calls. The logical operators are *and*, *or*, and *not*. The comparison operators are: <, <=, =, >=, >, <>, *like*, *is null*, *is not null*. The arithmetic operators are: +, –, *, /. The main difference with SQL syntax is that strings are delimited by double quotes instead of single quotes.

Embedded queries are not allowed in conditions. However, existential conditions can be expressed with a specific syntax as we will see in section 6.

## 4.3 Links

The Lazy language provides three kinds of links: the usual (reference) hypertext links, inclusion links (to build complex contents by including nodes in one another), and expand-in-place links that are inclusion links triggered by a user action.

### 4.3.1 Navigation links

Any content element or list of elements can serve as the source anchor of a reference link. An expression of the form

```
href target-node-name [parameter-value, ... ]( anchor-element(s) )
```

defines a reference link to a target node whose source anchor (the clickable text) is the anchor element(s).

*Exemple*. Here we have added to the `mw_Artist_after` schema a link to a node that shows the works of art of each artist in our.

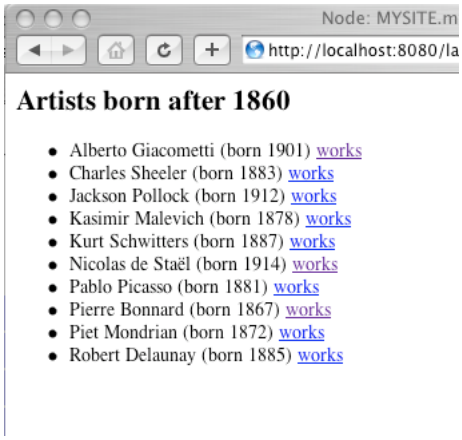
```
node mw_Artist_after[d]
  <h2>( "Artists born after ", d ) ,
  <ul>(
     {<li>(name," (born ",birthdate,") ",
           href mw_Works_by_artist[ano]("works")) }
  )
from artist
```

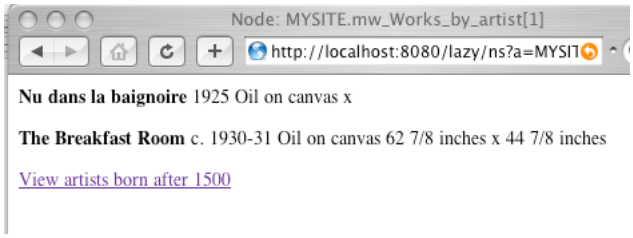
```
    selected by birthdate >= d order by name
```

The `mw_Works_by_artist` node can be defined as

```
node mw_Works_by_artist[artist]
 {
   <p>(<b>(title)," ",c_date, " ", support," ",
       height," x ",width
   )
 },
 <p>(href mw_Artist_after[1500]("View artists born after 1500"))
from work selected by author = artist order by c_date
```

It displays the title, creation date, support, height, and width of all the works created by the artist whose number is given as parameter. After the work list it contains a link to the list of artists born after 1500.

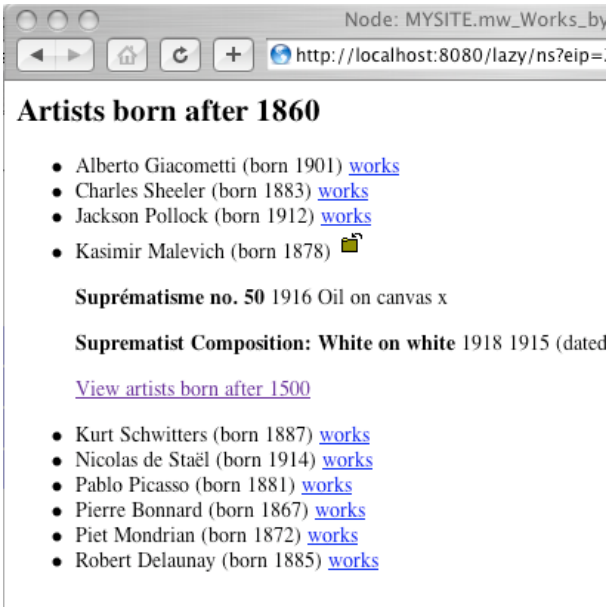



### 4.3.2 Expand-in-place links

In addition to standard hyperlinks (as found in HTML) it is possible to define expand-in-place links. These links have a different behaviour: when the anchor is clicked, it is replaced, within the current page, by the referred node.

Replace the href keyword by expand href in the `mw_Artist_after` node schema, recompile it an try clicking on links in the instance `mw_Artist_after[1860]`.




### 4.3.3 Inclusion links

The third kind of links is inclusion links. An inclusion link immediately includes the content of the referred link at the indicated place. An inclusion link does not have any anchor text.

Replace

```
href mw_Works_by_artist[ano] (name)
```

by

```
include mw_Works_by_artist[ano]
```

in the node schema and recompile it. The node `mw_Artist_after[1880]` should now look like this

Inclusion and expand-in-place links are powerful mechanisms to build nodes with rich, heterogeneous, and adaptive contents. Indeed, inclusion and expand-in-place form the basic construct to jump out of the “relational box”, i.e. to build data presentation that don’t look like tables or list of record.

In fact, with inclusions it is possible, in most cases, to avoid join expressions in the selection of expressions of the node schemas.

## 4.4 Examples

The following node schema presents information about a work of art. It uses a one-row and two-column table. The first column (td) presents textual information and the second one displays a picture of the work. The picture's URL is stored in the database attribute picture, note how this attribute is used to give its value to the "src" attribute of the "img" element.

This node is based on two tables: work and artist because we want to display the author's name, which is not in work. So we have to join the two tables with the condition work.author = artist.ano

In order to show the name of the work's owner we employ another technique, which consists in including the `mw_Owned` node (described below).

```
node mw_Work[id]
  {
  <table>(
    <tr>(
      <td>(
        <h2>(title), <p>(c_date),
        <p>(href mw_Artist[ano] (name)),
        <p>(support," ",height," x ",width),
        <p>(include mw_Owned[wno], ", " , acquired ) ,
        <p>(href mw_Works_by_artist[ano] ("Works") , " by " , name)
      ),
      <td>(
        <p>(<img width="200" src=picture>())
      )
    )
  ) ,
  <hr>() ,
  href mw_upd_Work[id]("Update") , " this description"
  }
  from work, artist
    selected by work.author = artist.ano and wno = id
```

The following node is simply here to display the name of a museum whose number is given as parameter. Its purpose is to replace a numeric value (here the museum no.) by a more explicit one. This node is used by mw_Work.

```
node mw_Owned[w]
  href mw_Museum[museum](name)
  from ownership, museum
  selected by work = w and museum = mno
```

## 4.5 Some design guidelines

Designing node schemas that form a good hyperspace is like designing software or buildings: there is no general method. However, you can find design ideas and a design and analysis methodology in the papers published by the ISI research group [http://cui.unige.ch/isi/reports]. What follows is a brief presentation of design guidelines that can help you creating a first version of a hypertext view on a database.

Although a database schema obeys to different design objectives and requirements, it can serve as a starting point for a hypertext view design. The idea is to create node schemas that correspond to the database tables and links that correspond to the foreign keys between these tables. It can be summarized in the following points.

### Nodes to display database tuples

For each table T create a node t to display the tuple that corresponds to a key value passed as parameter. The general node pattern is:

```
    node t[p]
        { attributes of T }
```

```
from T
selecty by Tkey = p
```

where TKey is a key of T.

Of course, if the key of T is made of several attributes K1, …, Kn, t must have n parameters p1, …, pn and the selection condition must be K1 = p1 and … and Kn = pn.

### Create hypertext links according to the foreign keys

For each foreign key UK of table U in T, i.e. UK holds a key value of U (it points to a tuple of U), add a navigation link to u in t:

```
node t[p]
    { attributes of T ,
      href u[UK]("some anchor text") , other links
    }
from T selected by Tkey = p
```

### Add inverse links

To allow navigation in the reverse direction (from u to t) it is necessary to create an additional node to display all the tuples of t that point to a given tuple of u (identified by its key value)

```
node _t_from_u [ uKey ]
    { href t[K]( K ) }
from T
selected by F = uKey
```

This node must be accessible from u through an href link:

```
node u[k]
    { …
       href t_from_u[UKey]
    …
    }
from U selected by Ukey = k
```

### Create entry points

Entry points are starting points for the navigation within your hypertext view. The simplest entry points are index nodes that display anchors to all or a subset of the tuples of a table. They are defined as follows:

```
node t_idx
    { ... href t[Tkey](some attribute(s)) ...
    }
from T
```

### Evolve to a more efficient hypertext design

The hypertext view obtained so far enables the user to view all the information content of the database. However, it is probably not a good hypertext in many respects: For instance, some navigation path may be too long, requiring too many clicks to reach the desired information, or information that one expects to see on one page may be scattered in the hypertext.

### Inclusion to factor out common parts.

The inclusion mechanism is interesting to re-use nodes and thus to avoid writing many times the same statements. For instance, the following node defines a node footer (also called navigation bar) that can be included in many nodes

```
node mw_To_index
  <hr>() ,
  href mw_Work_index ("[Work index]") , " " ,
  href mw_Artist_index("[Artist index]") ,  " " ,
  href mw_Exhibition_index("[Exhibition index]")
```

### Inclusion to "decode" numeric values

# 5 Creating Hypertext Views on Existing Databases

Lazy is aimed at creating hypertext views for existing databases. There are two ways to do this, depending on where you want to keep the Lazy dictionary. The Lazy dictionary is a set of database tables that store compiled node definitions, project definitions, access rights, etc. You can either continue having HSQLDB manage the Lazy dictionary (in a the Hsqldb database that comes with Lazy) or create a fresh Lazy dictionary in the existing database.

## 5.1 Keeping the Lazy dictionary in the HSQL database

### 1. Install a JDBC driver for your database management system

If your dbms has an ODBC interface (e.g. MS Access) you can use the sun.jdbc.odbc.JdbcOdbcDriver driver that comes with the Java installation. All you have to do is to create an ODBC source connected to your database. Otherwise you'll find the driver either in your dbms distribution or on specific web sites.

The JDBC driver must visible for the Tomcat server. So put it in $TOMCAT_HOME/lib.

**Oracle** JDBC drivers for Oracle can be downloaded from Oracle's site, they are called `oracle.jdbc.driver.OracleDriver` and come in `.zip` files (e.g. `classes12.zip`)

**Hsqldb** the hsqldb driver is called org.hsqldb.jdbcDriver, it comes with the Lazy distribution in file lazycat/webapps/LAZY/database/hsqldb/lib/hsqldb.jar

One important point is to find the correct URL schema to access your database through JDBC. Here are some examples:

**Oracle thin client** jdbc:oracle:thin:@hostname:1521:instancename

**ODBC source** jdbc:odbc:sourcename

**hsqldb** jdbc:hsqldb:hsql://hostname

### 2. Define a new database connection and a new project

a) Go to `http://127.0.0.1:8080/lazy/ns?a=ADMIN.all` and click the *Connections* link
b) Create a new database connection.
c) Define a new project and associate it to the new database connection (all the nodes of a project access the same database, given by the project's database connection).
d) Start creating node schemas in the new project

That's all

## 5.2 Installing the Lazy dictionary in an existing database

### 1. Install the appropriate JDBC driver

(see above)

### 2. Update `Lazy.properties` and `LazyCompiler.properties`

Edit the files `Lazy.properties` and `LazyCompiler.properties` in `$TOMCAT_HOME/webapps/lazy/WEB-INF/classes/` and modify the following lines

```
database.url=your database JDBC URL
database.user=your database user name ('sa' for hsqldb)
database.password=your dataase password (empty for hsqldb)
database.driver=class name of your JDBC driver
```

Although these files are not accessible form the Web, they should be carefully protected on your file system since they contain database passwords.

### 3. Modify the scripts to indicate the location of the JDBC drivers

The `JDBC_DRIVER` environment variable indicates where the JDBC drivers are. Update its definition in `$LAZY_HOME/bin/lazyenv.xxx` by adding the location of the the new driver(s).

### 4. Create the Lazy dictionary

Create the tables that will handle the Lazy dictionary (node definitions, users, roles, grants, etc.). The `$LAZY_HOME/admin` directory contains the SQL scripts to create these tables

| | |
|---|---|
| `lzydict-schema.sql` | the tables that stores node definitions |
| `lzydict-hsql-schema.sql` | the same for Hsqldb |
| `lzydict-ora-schema.sql` | the same for Oracle |
| `admin-schema.sql` | table schemas for all the security/administration related tables |
| `admin-hsql-schema.sql` | the same for Hsqldb |
| `admin-init.sql` | initial values for the administration tables |

NOTE. Although SQL is supposed to be the standard data definittion, query, and manipulation language, every DBMS has its own interpretation of that standard. Moreover, many important functions are left undefined in the standard (data conversion, string manipulation, etc.). Hence the specific definition scripts and node definitions.

### 5. Recompile the Lazy administration and development nodes

Compile the node schemas that are in `$LAZY_HOME/admin`. These nodes define the administration and development environment (to create and compile nodes, manage projects, users, aceess rights, connections, etc.). This is a good test to see if your `LazyCompiler.properties` file is correct. If you forgot to modify this file, the compiled nodes will be stored in the

```
% cd $LAZY_HOME/admin
% lc icon.lzy
Lazy node compiler 4.0b  (-->>< the url of your database >>)
project: ICON connectionId: DICTLAZY
node maj
node del
node new
```

....

```
% lc node.lzy
```

...

```
% lc admin.lzy
```

...

### Optionally reload the virtual museum example

The database schema, data, and node definitions par the museum example are located in $LAZY_HOME/examples/museum. Define the tables with `mw_dbschema_ora.sql` or `mw_dbschema_hsql.sql`, then load the data by executing `mw_data.sql` (there are variants in Macintosh or UTF-8 coding). Finally compile the museum nodes

```
% lc mw.lzy
% lc mw-contemp-for-hsql.lzy (only if using hsqldb)
% lc mw-updates-hsql.lzy or mw-updates-ora.lzy
```

Once again, since SQL does not define everything, different nodes had to be defined to cope with Hsqldb or Oracle (or other) dbms's specificities (like the (non-)automatic conversion of strings into numbers, auto-increment attributes, sequences, etc.).

## 6. Start the node server and test the installation

From your Web browser open the URL `http://localhost:8080/lazy/ns?a=ADMIN.all` and start browsing the administration and development environment.

# 6 The language (part II)

## 6.1 More details on the evaluation of tuple and non-tuple elements

### Dependency on the DBMS

The node compiler translates Lazy expressions into SQL statements that are directly executable by the node server. Since the semantics of SQL operators and functions may vary from DBMS to DBMS (in fact SQL is only partially standardized in this respect), the instantiation of a Lazy node may yield different contents, depending on the underlying DBMS. In particular, some DBMS's try to convert strings to numbers when they appear as argument to arithmetic operators, while others consider this as an error, or apply string operations.

### HTML or XML symbols in attribute values

It may happen that some attribute contains a value that have a special meaning when interpreted as HTML or XML texts. For instance, if the attribute *condition* contains the value "It is necessary that b<a before calling …", when displayed by a standard HTML browser it could appear as "It is necessary that b" because <a is considered as the beginning of a <a> tag.

To prevent attribute value from being interpreted as HTML or XML codes, the "potentially dangerous" characters must be replaced by their explicit representation: < for < and & for &. This can be done with the (almost) standard replace function, as in REPLACE(REPLACE(condition, "&", "&"), "<", "<"),

This replacement is not automatically carried out by Lazy because attributes may intentionally contain HTML tags. For instance, if the attribute is intended to store some part of a Web page.

## 6.2 Conditions and embedded queries

Embedded queries are not allowed in conditions. However, existential conditions can be expressed with the following syntax

`exists (`*table* `:` *condition*`)`

This expression is true if there is at least one tuple in *table* that satisfies the given *condition*. It corresponds to the SQL expression **exists** (**select** * **from** *table* **where** *condition*) For example, the following node schema generates node instances that contain all the artists who produced at least one work before the given date *d*.

```
node ArtistsWithWorksBefore[d]
    { <p>(name) }
from artist
    selected by exists( work : author=artist.ano and c_date < d)
```

The existential expression can also comprise several variables. An expression of the form `exists (`$table_1$ $var_1$, …, $table_n$ : $var_n$: *condition*`)` is true if there exist tuples $var_1$ in $table_1$ , …,, $var_n$ in $table_n$, that make *condition* true.

One can express universal conditions with expressions of the form

`forall (`*table* : *condition*$_1$ `=>` *condition*$_2$`)`

This expression is true if all the tuples in *table* that satisfy *condition*$_1$ also satisfy *condition*$_2$. It corresponds to the SQL expression **not exists** (**select** * **from** *table* **where** *condition*$_1$ **and not** (*condition*$_2$)). The multi-table form is

`forall (`*table*$_1$ *var*$_1$, …, *table*$_n$ : *var*$_n$: *condition*$_1$ `=>` *condition*$_2$`)`

which is true if all the tuples *var*$_1$, …, *var*$_n$ in *table*$_1$ × … × *table*$_n$ that satisfy *condition*$_1$ also satisfy *condition*$_2$.

## 6.3 More details on links

### 6.3.1 Generated URLs

Normally URLs are automatically generated by the Lazy system when you write an href statement. However, if you want to refer to a Lazy node from a "standard" HTML page, or directly by entering a URL in the address field of the browser, you must use the following URL scheme:

`http://`*hostname*`:`*port*`/lazy/ns?a=`*node_name*`&u=`*parameter*$_1$`&u=`*parameter*$_2$`&u=...`

(the standard port used by the tomcat server is 8080)

For example, to open the node `mw_Artist_after[1880]` on the local machine you must open `http://127.0.0.1:8080/lazy/ns?a=mw_Artist_after_2&u=1880`

## 6.4 Active Links

### 6.4.1 Principle

An active link is a reference link (href) that triggers a database action when clicked. In addition to usual elements, the source anchor of an active link may have one or more attribute setting elements and it must have exactly one action element. The general syntax of an active link is

`active href` *node_name*[*parameters*] (*standard-or-attribute-setting-or-action-elements* )

An attribute setting element takes the form

`set` *attribut_name* `=` *expression*

and an action element takes one of the following forms:

`on` "*button-label*" `do insert` *table-name*
`on` "*button-label*" `do delete` *table-name*`[`*key-attributes*`]`
`on` "*button-label*" `do update` *table-name*`[`*key-attributes*`]`

**Example.** The following node has an active link to insert a new tuple in table T and then jump to node m. This link will appear as a button labelled "insertion".

```
node aNewT [p1, p2]
    "If you click the following link, it will insert (",
```

```
    p1, ", " p2, ") into T ",
    active href m (
        set A=p1, set B=p2,
        on "Insertion" do insert T
    )
```

### 6.4.2 Syntax and semantics of the actions

Each link action corresponds to a SQL statement on the underlying database. The following paragraphs show what SQL statements are actually executed by the different types of actions.

#### Insertion

An active link of the form

```
active href N[…] (
        set A1 = e1, …, set An = en,
        on "b" do insert T
)
```

will execute the SQL statement

```
insert into T(A1, …, An) values(e1, …, en).
```

#### Deletion

An active link of the form

```
active href N[…] (
        …
        on "b" do delete T [K1=e1, …, Kn=em]
)
```

will execute the statement

```
delete from T where K1=e1 and … and Kn=em
```

There is also an abbreviated form of the delete action

```
delete T[K1, …, Kn]
```

is an abbreviation for

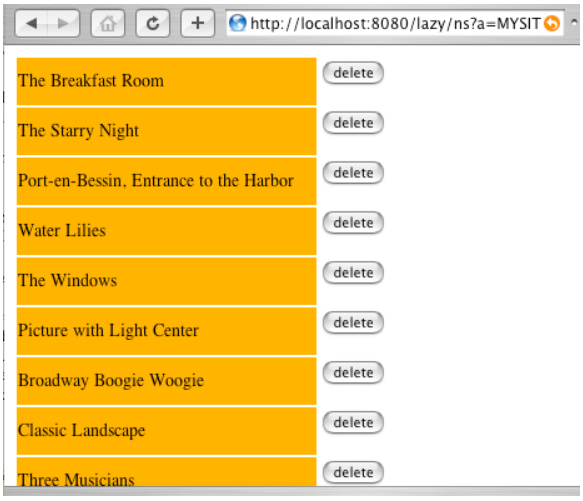
```
delete T[K1=K1, …, Kn=Kn]
```

It is used in the (frequent) case when the key value expressions are simply the attribute values of the displayed tuple. As in the following example:

```
node deleteAnyWork
    <table>(
```

```
    { <tr>( <td bgcolor="orange">(title),
            <td>(active href deleteAnyWork (
               on "delete" do delete Work[wno] )
            )
          )
    }
    )
from Work
```

An instance of this node will look like this:




## Update

An active link of the form

```
active href N[…] (
   set A1 = e1, …, set An = en,
   on "b" do update T [K1=f1, …, Kn=fm]
   )
```

will execute the SQL statement

```
update T set A1 = e1, …, An = en where K1=f1, …, Kn=fm
```

The abbreviated form update T[K1, …, Kn] means update T[K1=K1, …, Kn=Kn]

Examples of active links are given in the next subsection.

**Remark.** The `set` elements must not be confused with assignment statements. The effect of a set takes place only after the corresponding active link has been followed (by clicking the button. Thus, elements like

```
    set name = "Zorro", "The name is", name
```

Will display the value of the `name` attribute of the currently selected tuple, not `"Zorro"`. More on this in section 5.6.

### 6.4.3 A Remark about access rights

A use can execute a database action only if he or she has the required access rights. The user must have a role R that can access the database table of the active link. See section 7.

## 6.5 Inputting Values with Active Links

### 6.5.1 Input Elements

Active links are also the standard way to input values to the database. For this, we use attribute setting elements with input expressions. An input element takes the form:

```
set attribute-name = expression
                   | textfield(width)
                   | textfield(width, initial-value)
                   | textarea(nb-lines, width)
                   | textarea(nb-line, width, initial-value)
                   | select(expression_list | include node_ref)
                   | free( element )
```

Attribute-name must refer to an attribute of the table mentioned in the action element.

A node to create a new artist

```
node mw_new_Artist
  <h2>("Adding a new artist") ,
  active href mw_Artist_index (
     <p>("Unique identifier: " , set ano = textfield(10) ,
           " (a number)") ,
     <p>("Name: ", set name = textfield(60) ),
     <p>("Birthdate: ", set birthdate = textfield(10)),
     <p>("Deathdate: ", set deathdate = textfield(10)),
     on "Add" do insert artist
  )
```

Although this node will act on the "artist" table, by inserting a new tuple, its content does not depend on any particular table. When generated, this node displays the heading text, four input fields (for ano, name, birthdate, and deathdate), and an"Add" button. When the user clicks on on"Add", the system takes the input values to form a new tuple, inserts this tuple into the artist table and then jumps to the node mw_Artist_index. It is not mandatory to define all the table's attributes to insert a new tuple, only those that are either part of the primary key or not "nullable" are required.

The following node enables the user to change the name of a given artist.

```
node mw_upd_name_Artist[w]
  <h2>("Changing the name of artist no. ", w) ,
  active href mw_Artist[w] (
     <p>("Name: ", set name = textfield(60, name) ),
     on "Update" do update artist[ano]
```

```
  ) ,
  href mw_Artist[w] ("Don't change")
  from artist selected by ano=w
```

This one adds 5 to the birth date of an artist (very useful indeed!).

```
node mw_add5_Artist[w]
   <p>("Click on the \"Plus 5\" button to add 5 years to the birth date
of " , name ) ,
  active href mw_Artist[w] (
     set birthdate = birthdate + 5 ,
     on " Plus 5" do update artist[ano]
  )
  from artist selected by ano=w
```

### 6.5.2 Using the Input Values as Parameters

The input values can be used as parameters to the destination node. For instance, the following node reads attribute values for a new artist, creates a tuple when the button is clicked, and then jump to a node that displays the newly created artist.

```
node mw_new_Artist_2
  <h2>("Adding a new artist") ,
  active href mw_Artist[ ! ano ]
      p>("Unique identifier: " , set ano = textfield(10) , " (a num-
ber)") ,
     <p>("Name: ", set name = textfield(60) ),
     <p>("Birthdate: ", set birthdate = textfield(10)),
     <p>("Deathdate: ", set deathdate = textfield(10)),
     on "Add" do insert artist
  )
```

When the `"Add"` button is clicked, it inserts a new tuple into `artist` and then jumps to the node mw_Artist[!ano]. The ! ano notation indicates that the value given to the ano attribute in the set ano = textfield(10) element must serve as parameter to instantiate mw_Artist. In this case, it will jump to the node that displays the artist with the number we just entered.

## 6.6 Active Nodes

Active nodes offer yet another means to update the database when navigating a hypertext view. An active node is a normal node that is equipped with a list of "pre-actions". These are database actions that are executed just before the instantiation of a node. The syntax for pre-actions is:

```
   node N[parameters]
      ...
   from ...
      selected by ...
      order by ...
      on open { action , ... }
```

Actions can be either insertions, deletions, or updates expressed with the following syntax:

### Insertion

insert *tablename* [ $attribute_1$ : $value_1$, ..., $attribute_n$ : $value_n$]

it corresponds to the SQL statement insert into *tablename*($attribute_1$, ..., *attribute*) values($value_1$, ..., $value_n$)

### Deletion

delete tablename ( condition )

corresponds to the SQL statement delete from tablename where condition.

### Update

update tablename ( conditions) set [attribute1 : value1, …, attributen : valuen]

corresponds to the SQL statement update tablename set attribute1 = value1, …, attributen = valuen where condition.

Pre-actions are particularly useful when updating the database requires several actions on different tables. In addition, these actions can act on the database "behind the scene", without the user even noticing that his or her navigation did something on the database. For instance, to log the accesses to a particular node, we could write the following node:

```
node record_access[par]
    // no contents
on open {
    insert rec_table[time: sysdate /* for Oracle */, what: par]
}
```

Then in another node :

```
node W ...
    include record_access["access to node W"]
    ...
```

## 6.7 Session Variables

### 6.7.1 System variables

The Lazy node server maintains a number of "system variables" that store information about the current session. These variables are:

| | |
|---|---|
| [USER] | the current username |
| [GRP] | the datagroup the user is working in |
| [LANG] | the user's preferred language |

They can appear anywhere in the definition of a node (in the content or in the selection part).

### 6.7.2 User variables

A user variable is a variable whose value is either entered by the user in an input field (set var = textarea, textfield, etc.) or obtained as the value of an expression (set var = expr). The purpose of user variable is to store values for later use by other nodes. Thus, these variables must be set within the definition of an *active link* and they take their value when the user clicks the link's button.

It is important to note that the value of an input variable is not available until the active link has been followed. Hence it cannot be used elsewhere in the node that sets its value (more precisely, reference to the variable in the node will yield to old value of the variable).

However, input variable values can be passed as parameter to the target node of the active link, with the *!variable* syntax, as shown in the following example. Within the target node, and subsequently visited nodes, expressions of the form [!*variable*] will yield the value of the corresponding input variable.

#### Example

The Selector node displays an input form to read a string and level of detail. It stores these values in two variables when the user click the "Go" button. Since no database action is required, the active link simply navigates to the Work_with_name[!word] node (after storing the values in the appropriate user variables).

```
node Selector[]
    // input a word and a level of detail, then go to the search node
    active href Works_with_name[ ! word ] (
        <p>("Word of the work's title ", set word = textfield(20)) ,
        <p>("Level of detail (1-9)", set detail = textfield(1)),
        on "Go" do navigate
    )

node Works_with_name[w]
    // select works whose title contain w and show them at the specified
level of detail
    { title, … ,  include W_Description[ [!detail] ] ... }
from works
    selected by title like concat("%", concat(w, "%"))
```

## 6.8 Some design principles

There are basically two ways to design active nodes

1. Design "form nodes", like the `mw_new_Artist` node above, that let the user type in data in the input fields;
2. Design navigation strategies to collect the requested information through navigation (using parameters to transmit data from node to node)

The following nodes show an example of the second design alternative:

```
node mw_exhibition_content[e]
    { // display the content of exhibition e
     title , . . .
```

```
    }
    href mw_select_work[e] ("Add a Work to this exhibition")
from ex_content, work
   selected by exhibition=e and work=wno

node mw_select_work[e]
    <h3>("Select a work to include in the exhibition") ,
    <dl>(
          {<dd>( expand href mw_Work[wno](title) , " " ,
                    href mw_add_work_exh[e, wno]("[Add]")
              )
          } )
from work
```

### 6.8.1 Collecting input values

### 6.8.2 Generating key values

In many occurrences relational tables have a primary key which is a sequential number with no particular meaning, These keys are here to play the role of a unique object identifier and to avoid referencing tuples through complex multi-attribute key values. The problem is how to generate new key values when inserting a new tuple. Almost each DBMS has its own approach to this problem:

Oracle has sequences, which are objects that yield new values when asked for;
other systems have auto-incrementing attributes;
etc.

so Lazy does not propose a standard key value generation system and the hypertext view designer must rely on the DBMS generation system.

# 7 Managing projects

## 7.1 Projects

A project is mechanism to group a set of node schema that have a common purpose. It is similar to a module in a programming language. The IDE has an environment to manage project (create new projects, delete them, add, update, and delete nodes in a project).
Every project has a set of properties that are inherited by all its nodes:
a database connection
formatting files (CSS, XSL, background image)
a default node type (HTML or XML)
The full name of a node n belonging to project p is p.n. When refered from a node of the same project, the p. prefix can be omitted.

## 7.2 Connections

A database connection is a JDBC connection, it is comprised of a database URL, a JDBC driver (a class name), a username, and a password. With different projects connected to different databases it is possible to build a single Lazy interface that federates several databases.
After creating a new connection (with the Admin>Connection interface) it is necessary to "reinitialization all connections" in order to activate this connection. A reinitialization is also required when some database connection has been lost due to a network or database problem.

## 7.3 Users

A Lazy user is characterized by his or her username and password. Access rights to Lazy nodes and database tables can be assigned to users. When a new session is established, the user is automatically logged in with ther username PUBLIC. If he or she requests a node that has access restriction, he or she will be asked to enter a username an password. In node schemas, the session variable [USER] returns the username.

## 7.4 Internationalized strings

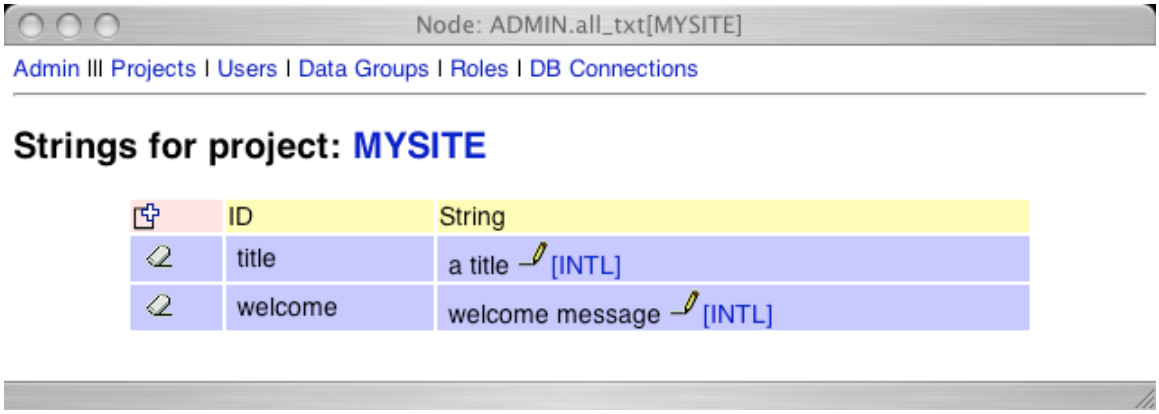
When developing an international site, it is necessary to produce pages in different languages. However, it would be tedious and error prone to define a version of each node in each language. To internationalize a node, the approach taken in Lazy (and in most systems) is to replace all the literal strings that appear in the node by internationalized variable of the form [?*variable*].

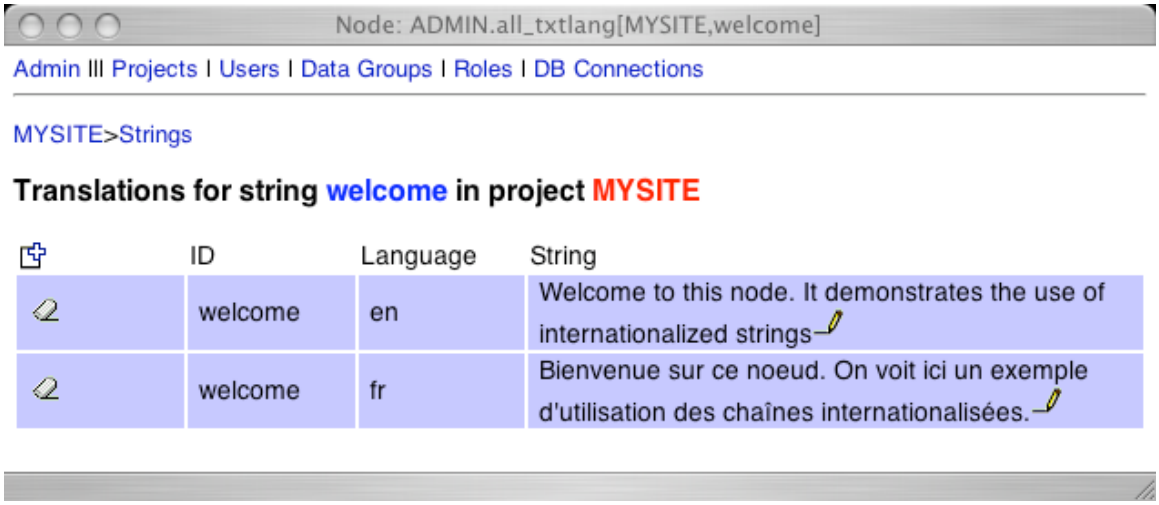
There are two ways to define internationalized strings: 1) with the ADMIN interface; 2) by storing their definitions directly in specific database tables (this can be useful if you have many strings to define and you already have them in a file or somewhere else)

### Defining strings with the AMDIN interface

Go to the Projects page and click the String link beside your project (internationalized strings are project dependent). On the String page you'll see the currently defined strings. This is the "ge-

neric" level. Then you can define translations for each string and each language by clicking [INTL].



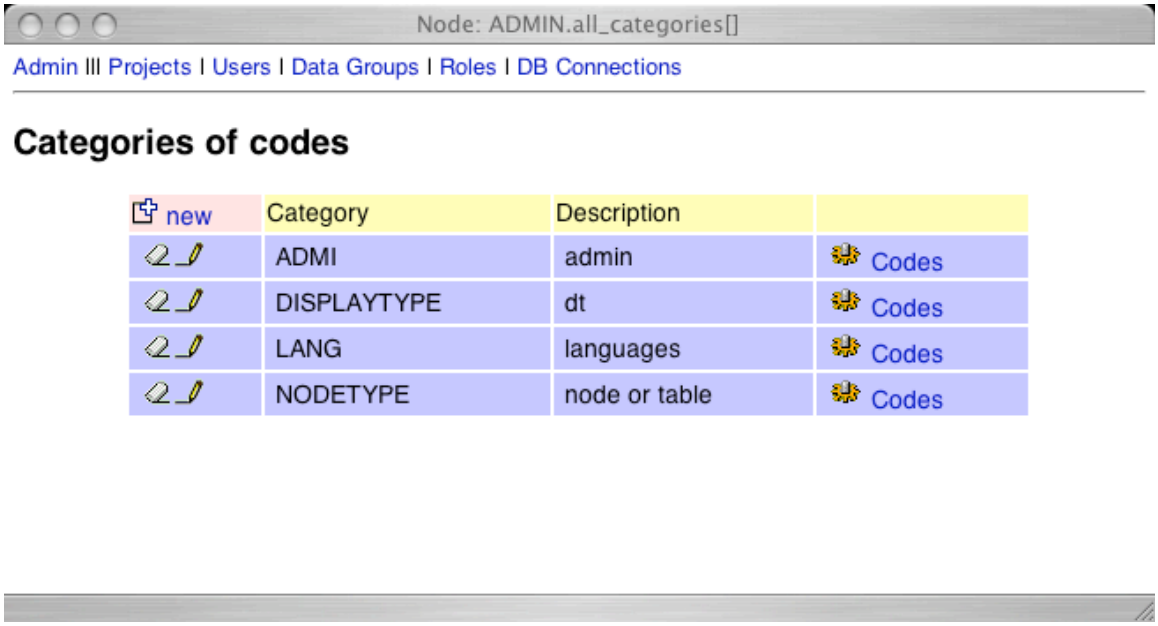

When an internationalized variable of the form [?*var*] appears in a node, the evaluation proceeds as follows: If *var* is defined for the user's language, the variable is replaced by its value for this language. If it is not defined for this language, then it takes the generic definition.

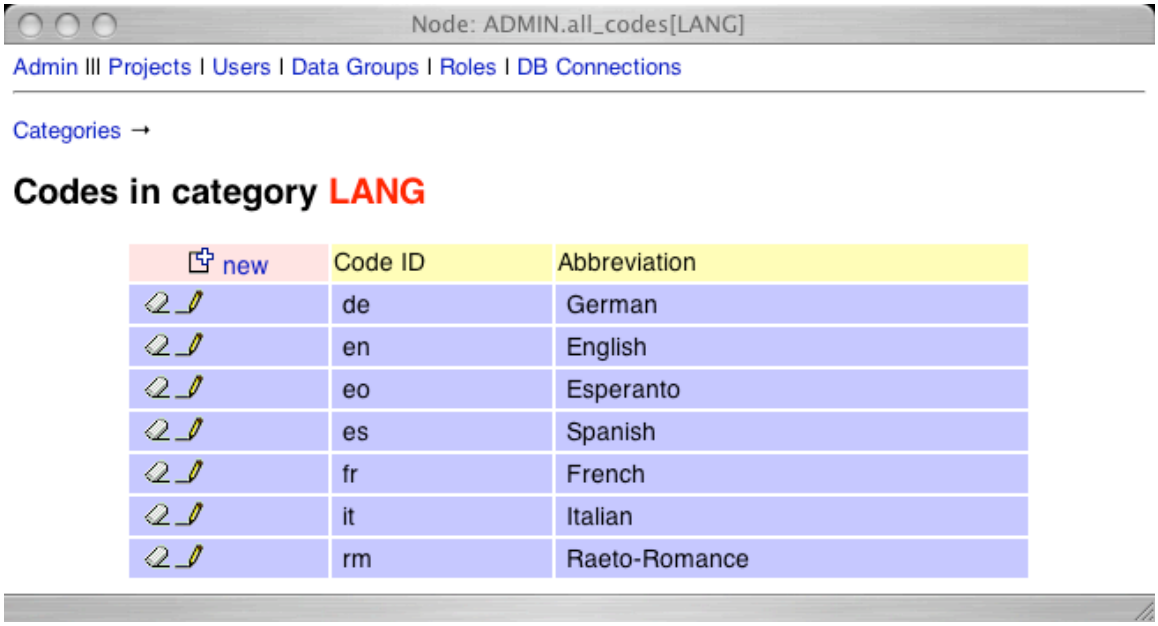
### Storing string definitions directly into the database

The table LAZY_TXT(ProjectID, TxtID, Lib) holds the generic string definitions. A triple (*p*, *t*, *l*) in this table means that the string *t* has the generic value *l* in project *p*. The table LAZY_TXTLANG(ProjectID, Lang, TxtID, Lib) holds the string values for the different languages. A quadruple (*p*, *t*, *l*) in this table means that the string *t* has the generic value *l* in project *p*.

### Adding new languages

To add a new language: from the administration home page go to the *Codes* page and then click Codes on the LANG row. This will display a list of currently defined languages. You can then update this list.




It is also possible to directly update this table: LAZY_CODES(CatID, CodeID, Abr, ListDef). For instance, to add the Elfic language, insert a tuple ('LANG', 'elf', 'Elfic', 'Y').

## 7.5 Datagroups

Datagroups are intended to horizontally partition data between different groups of users. For instance, the same database could be used to manage data belonging to different departments. Datagroups are useful when different groups share the same data structures but not the same data. A standard way to handle this situation is to horizontally partition the tables by adding a *Group* (or

something similar) attribute. When a user belonging to group g accesses the database, he or she must see only those tuples that have the value g in their Group attribute.

The global variable [GRP] represents the datagroup of the user. A typical use of this variable is thus

```
node …
    …
from MyTable
selected by an expression and Group = [GRP]
```

The dictionary schema given in appendix shows the relationships between all the security and project management related concepts. It is a UML view of the Lazy dictionary tables.

# 8 Encryption

## 8.1 Security in Lazy

Because the property files Lazy.properties and Compiler.properties contain the database password of the db where the Lazy dictionary (if not the data) is stored, it is of the highest importance to protect them against unauthorized access. Normally, read access to these files should be granted only to the Lazy owner.

## 8.2 Why encrypt ?

Lazy can connect to several databases, the connection password of these databases should be protected. In addition, the user passwords are stored in the 'dictlazy' database. These password should also be protected. Finally, an probably most importantly, when Lazy is used to update a database, important information (such as table names, attribute names, key attribute values, etc.) may appear (as hidden fields) in the pages sent to the user. A malicious user 1) get information about the database schema of the application and 2) forge pages containing updates (in forms) and send them to the server in order to slip wrong data into the database.

## 8.3 What is encrypted ?

When encryption is turned on, the following items are encrypted

- All user passwords
- All database connection password (except the initial 'lazydict' connection)
- All the table names, attribute names, key attribute values appearing in forms in HTML pages send to the user.

## 8.4 How to migrate to the encrypted form ?

The first step consists in choosing a permanent encryption key -- this key will encrypt user passwords and database connection passwords -- it must never change. This key (made of 16 hexadecimal digits) is stored in Lazy.properties as the encrypt.key attribute value.

encrypt.key= <<16 hex digits key>>
encrypt.off=on

in Lazy.properties

Now there is a bootstrap problem: if you turn encryption on, by setting the encryt.off parameter to on in Lazy.properties, you won't be able to log-in any more because the system will try do decypher the stored passwords before comparing them to the typed passwords. So you'll need to manually encrypt the ADMIN password in the secure_user database table before proceeding to the next steps. This can be done by as follows

% $LAZY_HOME/bin/gpwd your-admin-password
33833A88E8F10377 <--- your password encrypted with the Lazy key

In the database :

update secure_users set pwd = '33833A88E8F10377'  where userid = 'ADMIN';

Now you can log-in as ADMIN an set the other user's passwords (they will be encrypted)

# 9 Language (part III)

## 9.1 Computing with inclusions

## 9.2 Java Nodes

When complex computations are needed (or simply functions that do not exist in SQL), it is better to express them in an algorithmic language rather than hack “computation-oriented” Lazy nodes. For this purpose one can define “external nodes”, the content of which are computed by calling a Java method.
The name of an external node starts with two underscore characters followed by a class name and a method name. The instantiation of such a node will call the corresponding method to generate a node content. For instance, a content element of the form include _MW.artistAbbreviatedName[a] will call the static method artistAbbreviatedName of class MW and include the resulting string in the node content.
A "node" method must have one of the following signatures

```
static String methodname(String parameter1, …, String parameterN)
static String methodname(String[] parameter)


class MW {
    static String artistAbbreviatedName(String n) {
        String af = n;
        int x = n.indexOf(" ");
        if (x > -1) af = n.charAt(0) + "." + n.substring(x+1);
        return af;
    }
    /* other methods */
    }
```

. The second form is intended to implement nodes with a varying number of parameters.
For instance,

```
    static String concatIf(String [] params) {
        String r = "";
        for (int i=0; i<params.length-1; i+=2) {
            if (params[i].equals("yes")) r += r.params[i+1]
        }
        return r;
    }
```

This class must be compiled and the .class file stored in the webapps/lazy/WEB-INF/classes directory. Use for instance the following compilation parameters:

javac MW.java -d $TOMCAT_HOME/webapps/lazy/WEB-INF/classes
Accesing the database from an external method
The following Lazy server methods can be invoked from an external method to execute SQL statements on a database corresponding to a database connection.

```
class DBServices
```

```
/*
* sql : the statement
* db : name of the database connection
* select : must be true for a 'SELECT' statement and false for
*   an 'INSERT', 'DELETE', 'UPDATE'
*/
public  static QueryResult execSQLonDB(String sql,String db, boolean
select)

/*
* executes the sql statement on the LAZYDICT connection
* (the connection to the Lazy dictionary database)
*/
public  static QueryResult execSQL(String sql, boolean select)

/*
* Query results are stored in a QueryResult object defined as follows
*/

class QueryResult {
    Statement stmt;
    public ResultSet result;
    public boolean valid = true; // false => error in query processing
    public String msg = ""; // error message if valid==false
    public String sql;
    public int nbUpdated = 0;
```

Examples:

```
static String updel(String keyval, String val) {
    QueryResult q1 = execSQLonDB(
        "delete from T1 where TK='+keyval+"'", "Z_CONNECTION", false);
    QueryResult q2 = execSQLonDB(
        "update T2 set A='"+val+"' where T2K='"+keyval+"'",
        "Z_CONNECTION", false);
    String r="";
    if (!q1.valid) r += q1.msg;
    if (!q2.valid) r += q2.msg;
    return r
}
```

The DBServices and QueryResult classes are located in `webapps/lazy/WEB-INF/classes`. To compile methods that refers to them, it is necessary to add their location to the CLASSPATH. For instance

javac MW.java -classpath .:$TOMCAT_HOME/webapps/lazy/WEB-INF/classes \
-d $TOMCAT_HOME/webapps/lazy/WEB-INF/classes

# 10 Language reference

## 10.1 Lazy Syntax in BNF

```
startrule =
"define"
        “project” project_identifier
                [ “[“ nodetype = (“xml” | “html” | “purexml” | “purehtml”) “]” ]
        { node_def }
"end".

node_def = "node" node_identifier parameters [cachesize=number]
                content
                from_part
                on_open

parameters = [ "[" parameter { "," parameter  } "]" ]

parameter = identifier

identifier = letter { letter_or_digit | "_" }

content = field { “,” field }

field =   ["expand"] "href" [ "in" target_identifier] node_reference "(" field_list ")"
        | "include" node_reference
        | "active" "href" node_reference_for_active_link "(" field_list ")"
        | set_attribute_or_variable
        | on_action
        | content


field_list = field { "," field }

content =   element_type [ "(" [ field_list] ")" ]
      | simple_expression
      | "{" field_list "}".


element_type = "<" element_type_identifier { element_attribute_identifier "=" simple_expression
} ">"

elem_type_identifier = identifier { (':' | '-') identifier }

element_attribute_identifier = identifier { (':' | '-') identifier }

node_reference = [project_identifier "."] node_identifier
```

```
        [ "[" simple_expression | "include" node_reference { "," (simple_expression | "include"
node_reference)} "]"

node_reference_for_active_link = [project_identifier "."] node_identifier
        [ "[" ("!" identifier) | simple_expression | "include" node_reference
        { "," (("!" identifier) | simple_expression | "include" node_reference}) "]"


active_link = "href" [project_identifier "."] node_identifier
                        [ "[" simple_expression { "," simple_expression } "]" ]

set_attribute = "set"
                (attribute_identifier | "parameter" | "parameter_encoding"
                | input_function_identifier "(" identifier ")" )
                "=" (input_field | simple_expression)

input_function_identifier = "encoded" | …

input_field = "textfield" "(" simple_expr [ "," simplexpr ] ")"
                | "textarea"  "(" simple_expr "," simple_expr [ "," simplexpr ] ")"
                | "free" "(" field ")"
                | "select" "(" ("include" node_reference | simple_expr { "," simple_expr }) ")"

on_action = "on" string_expression "do" (
                        "navigate"
                        | "insert" table_identifier
                        | ("delete" | "update") table_identifier "[" keyvalue {"," keyvalue} "]"
                        )

keyvalue = keyattrident [ "=" simple_expression ]

from_part = ["from" table_identifier [alias_identifier] { ',' table_identifier [alias_identifier] }
                ["distinct"]
                select_part group_part order_part
        ]

group_part = [ "group" "by" identifier {"," identifier} ]

select_part = [ "selected" "by" condition].

order_part =  ["order" "by" simple_expression ["desc"] {',' simple_expression ["desc"]}]

condition = ["not"] logical_term { "or" logical_term }

logical_term = logical_factor { "and" logical_factor }

logical_factor =  simple_expression (  comparision_op simple_expression | "is" ["not"] "null" )
        | exists_expression | forall_expression

comparison_op = "like" | "<" | "<=" | ">" | ">=" | "=" | "<>"
```

exists_expression = "exists" "(" collection_id [alias_id] { ',' collection_id [alias_id] } ":" condition ")"

forall_expression = "forall" "(" collection_id [alias_id] { ',' collection_id [alias_id] } ":" condition "=>" condition ")"

simple_expression  = [ "+"|"-"] term {( "+"|"-") term }

term = factor {("*"|"/") factor }

factor =  stringconstant
        | numberconstant
        | function_identifier "(" simple_exression { "," simple_exression } ")"
        | [table_identifier '.'] attribute_identifier
        | parameter_ident
        | ("[" session_variable_ident | ("!" input_variable_ident) | ("?" q_variable_ident) "]")
        | "(" expression ")"

expression = condition

Content expression may not yield a boolean value (because the boolean type does not exist in SQL). This is not forbidden by the grammar, neither checked by the compiler. Boolean contents will generate ill-formed SQL statements.

on_open = "on" "open" "{" action_list "}"

action_list = basic_action {"," basic_action }

basic_action = insert_action | delete_action | update_action

delete_action = "delete" table_identifier [ "(" condition ")" ]

update_action = "update" table_identifier "(" condition ")"
        "set" "[" attribute ":" simple_expression {"," attribute ":" simple_expression} "]"

insert_action = "insert" table_identifier
        "[" attribute ":" simple_expression {"," attribute ":" simple_expression} "]"

# 11 Glossary of terms

### Node schema

A text, written in the Lazy language, that defines how to construct a node instance (usually an HTML or XML page). It is comprised of
a selection part
a content part (with links)
and parameters

### Node instance

A hypertext node (usually a HTML or XML page) that is part of a hypertext view. It is an instance of a node schema. Its content depends on the schema definition, the actual parameter values, and the current database contents (tables). A node is instantiated each time the user clicks on a link to the node, or when the node is include in another one (see inclusion link).
Node type
The type of content produced when instantiating a node. Currently defined types are : html, xml, pure html (no heading tag is generated), pure xml, svg. The node type is specified at project level, i.e. all the nodes of a project have the same type.

### Node server

The role of the node server is to answer to node requests by instantiating the required node schema with the given parameters. It also performs database actions corresponding to active links.

### Node compiler

The program that checks the syntax of a node and transforms is different parts into SQL statements (which may still contain placeholders for parameters).
Link
A way to interconnect nodes. The source of a link is a content element of a node, its destination is a node. Lazy links can have four different behaviours: reference (jump), inclusion, expand-in-place, and active .

### Reference link

A link from a content element to a node. When the node is instantiated, such a link appears as an undelined text (an HTML anchor). When it is clicked, the referred node is instantiated an replaces the current node (the user "jumps" to the referred node.

### Expand-in-place link

A link from a content element to a node. When the node is instantiated, such a link appears as an undelined text (an HTML anchor). When it is clicked, the content of the referred node is instantiated "in-place", it replaces the anchor text.

### Inclusion link.

A link intended to include the content of the target node withing the source node, at the link location. Inclusion links are useful to create complex node contents.

### Active link.

A link that triggers a dabase action when it is followed.

### Project

A set of nodes that make up a hyperspace intended for a specific purpose. All the nodes of a project have the same type (html, xml, purexml, purehtml) and refer to the same database connection. Every node belongs to exactly one project. Thus the full name of a node has the form projectname.nodename .

### Database connection

an access to a particular database (schema). It is comprised of a database URL (e.g. jdbc:hsqldb:hsql://localhost), a JDBC driver class (e.g. org.hsqldb.jdbcDriver), a username, and a password. The URL and driver are dbms specific (refer to your dbms documentation).
User
a name under which somebody can connect to a Lazy server. By default the username PUBLIC is used when opening a Lazy session. A user can be defined as "administrator". In this case she will have full access to every node and data.

### Datagroup

a symbolic name for a set of data in a database. A tuple in a table may belong to zero or one datagroup, thus the same table can hold data belonging to different applications. For instance, a single database can be used to manage different museums. The datagroup of a tuple is indicated by the value of an attribute reserved for this purpose.
When loging in, a user can select a datagroup, otherwise she will work in her default datagroup.

### Role

a role represents a set of access rights to nodes (and database tables). For instance, role VISITOR can accees the node MUSEUM.artist_index while role CURATOR can access MUSEUM.update_catalog, etc. An access right is thus a pair (role, project.node) or (role, table). The wildcard form project.* means "all the nodes of this project". The main advantage of roles is to avoid defining the access rights of each user in each datagroup (see grant).

### Grant

when connecting, a user is granted a set of roles. These roles depend on the datagroup she chose to work in. So, for instance, user JOE could have role VISITOR in the MUSEUM_OF_MODERN_ART datagroup and roles VISITOR and CURATOR in CITY_MUSEUM. A grant is thus a triple (user, datagroup, role). If datagroup is * it means that this user has this role in all datagroups.
Servlet

### Servlet container

this is an http server that can host servlets written in Java, Apache Tomcat is the reference servlet container. When the server receives a URL request that correspond to a servlet, it invokes the doGet or doPost method of this servlet.

# 12 Bibliography

See http://cui.unige.ch/isi/reports

## Appendix A. The Lazy dictionary