NODEWISE CONNECTRIX

or

A METHOD FOR IMPLEMENTING A WEB-BASED INTERFACE TO HEIERARCHICAL TREE SYSTEMS FOR NON-PROGRAMMERS

An Interactive Qualifying Project Report

submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

_____ _____

Peter Goodspeed **Ryan Munro**

4 Mar, 2004

_____

Professor Michael Ciaraldi, Advisor

Abstract

The Nodewise Connectrix IQP consisted of the design and implementation of a simple interface. With this interface, users may expedite the production of complex tree structures stored in a database with a web interface.

This interface is designed expressly for non-technical people, and can be easily used to create family trees, business hierarchies, and other useful models for people unused to programming or the internal workings of computers.

Acknowledgements

The Gentoo Project (gentoo.org) -- For putting together Gentoo Linux and its excellent Emerge utility

The Apache Foundation (apache.org) -- For making the most reliable web server in common use, and giving it away for free

The MySQL Project (mysql.com) -- For making a free database management system which interoperates with the above

The PHP Project (php.net) -- For making a language designed for the web, instead of one that simply supports it

Sun Microsystems (sun.com) -- For building commercial quality GUI tools into the most portable language to date

WPI Professor of Practice Michael Ciaraldi -- For advising this IQP, and pointing it in the right direction when it certainly needed it

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Prologue

1.1.1 Nodewise Connectrix started almost a year ago, when Peter's mother suggested that someone should really write a good piece of software to make communication between family members easier. This led to the design of a simple database system with a web interface to handle that, but in the end, even after several hours of work, the system was not nearly well enough implemented to be called complete. This, in turn, led to the realization that nobody had ever written a generic setup program to set up databases with web interfaces; the only people who had the necessary skills to set up such a database were programmers, and it still took nontrivial amounts of time. Further contemplation revealed that putting such generic setup program together, and making it simple enough that even a non-programmer could use it well, would be an interesting IQP.

1.2 Summary of Project

1.2.1 Nodewise Connectrix is designed as a simple setup utility for design and implementation of any generally hierarchical tree or graph as a database structure, with a web interface. It currently assumes the existence of Apache, PHP, and MySQL on the server already; however, later releases will be smart enough to detect the absence of any of these ingredients and initiate the install process for them automatically.

1.2.2 The project was designed with an average computer user with little to no programming experience in mind. There are useful help files containing definitions and walkthroughs for every implemented part of the project, and the user interface was specifically designed to help someone unfamiliar with programming terms and concepts design a workable database.

1.2.3 There is a complete, simple language included with this project. Although it is not Turing-complete, it is designed to be an easy to understand mechanism with which users can define custom, automatically updating relational queries.

2. DESCRIPTION OF PROJECT

2.1 Completed Version

2.1.1 Nodewise Connectrix is divided into three main subsystems: The Development Tool, the Language Parser, and the Web Interface. The first two systems are only used once, at setup time, by an administrator. The last of these is produced by the first two, and is the only part the end-user will ever see.

2.1.2 Development Tool

2.1.2.1 The Development Tool is a graphical user interface which steps the user through the process of configuring their database and PHP scripts, without requiring any knowledge of databases or PHP. It is written in Java for portability. User documentation for the Development Tool can be found in Appendix A1.

2.1.2.2 The best way to learn what the Development Tool is and does is to download and run it yourself. Source files may be found in the source CD in the /devtool directory.

2.1.3 Language

2.1.3.1 The Language is a custom-designed relational language called FunctionCode, which as of this writing is unique to this project. It is designed to be simple and easy to learn, yet provide an expressive way to design complex queries without having to know any SQL. Although all the user sees of the parser is in the 'Parser' tab, it is actually a complete package which can be imported into any Java application which wants to use this functionality.

2.1.3.2 User documentation for the Language can be found in Appendix A2. Source files can be found in the source CD in the /devtool/parser directory

2.1.4 Web Interface

2.1.4.1 The Web Interface is a user-friendly way for the end user to view and modify the contents of their database. It is composed of several PHP pages, all of which are either generated by the Development Tool or come with Nodewise Connectrix without needing any modification. An example can be found at http://amalcon.res.wpi.net/actual/ . Authentication: test/test .

2.1.4.2 Due to the constraints of life, it is impossible to guarantee the operation of this website except during the work week (Monday – Friday) before 1 April 2004. After that point, in order to see the web forms in action, it will be necessary to set up a server of your own, with MySQL and PHP support. At that point, you should be able to drop these files in, tweak only a few settings, and see them work. Alternately, after setting up the server, you may as well create your own database by running the Development Tool.

2.2 Initial Specifications

2.2.1 This project went through several design incarnations before we finally implemented it. Here's a review of some of the changes it went through.

2.2.2 Initial Concept

2.2.2.1 Initially, the project was to be a complete package, with a cross-platform installer which checked for the necessary prerequisites (Apache, PHP, MySQL) and installed and configured any which weren't already present. It would take the user from point of download (a single file) to completion of setup (a web address where they could begin entering information into their database) in one seamless process.

2.2.2.2 The languages to be used initially included PHP for the web interface, MySQL for the database implementation, Prolog for the user-function language, and Java for the development tool and as an interface to allow the PHP to communicate with the Prolog.

2.2.3 Revision 1

2.2.3.1 In the interests of time, we decided to leave the design and implementation of the installation package and the advanced security settings for a later release. This left us with three components to work on: the SQL Development Tool, the Parser, and the Web Interface.

2.2.3.2 Additionally, we decided that, as MySQL didn't support subqueries, we would use PostgreSQL instead.

2.2.4 Revision 2

2.2.4.1 Due to various configuration problems, we couldn't make PostgreSQL work with PHP at all, so we decided to revert to MySQL and avoid the use of subqueries through the use of PHP functions.

2.2.4.2 Additionally, as we couldn't make Prolog work in a reliable or consistent manner, we decided to scrap the notion of using it, and instead write our own function-based language. Writing our own language has the added advantage of allowing us to build in SQL generation directly, instead of having to write a converter. This had the side effect of removing the Java translation layer from between the PHP and the Prolog; instead, we generated relevant PHP in the Development Tool.

2.2.5 Morph into Final Version

2.2.5.1 The final version of the project is essentially identical to Revision 2. However, the design of the web forms changed slightly, as did some implementation details in the Development Tool.

2.3 Differences between Initial Concept and Final Version

2.3.1 Scope

2.3.1.1 The biggest difference between the initial concept and the final version is here, in the whole scope of the project. Implementation aside, designing the

complete package could have easily consumed the seven weeks we had available. Therefore, we had to pare down the project to the most significant portion, and assume that the uninteresting bits happen in later releases. The majority of the postponed and dropped functionality is purely technical and thus not as relevant to the focus of this project.

2.3.2 Added Functionality

2.3.2.1 The project did grow in certain ways from the initial specification. The most noteworthy of these ways was the design and implementation of our own language for the user to write functions in. The original idea was to use a combination of public domain software packages to generate SQL queries on the fly, by having PHP use its Java library functions to run a java program which interpreted Prolog, to run a package (written in prolog) which converted prolog statements into SQL queries.

2.3.2.2 Though it is probably possible to configure such a system properly, we decided that it would be easiest to implement our own SQL generation language, and generate PHP code at development time which would be run later.

2.3.3 Removed Functionality

2.3.3.1 Originally, there was to be a concept of a 'group,' which was to be a cluster of nodes which could be acted upon as one. This was to make communication easier. Thus, one could group all nuclear families, and compose an email to an entire family, instead of looking up the email address of each member of the family. This functionality was removed because it simply didn't fit the nature of the project. It would have required a great deal of additional configuration for only a little bit of added functionality. This conflicted with the design goal of making configuration as simple as possible, to achieve a minimum of required proficiency in the end user.

2.3.3.2 Additionally, there was originally going to be an email page in the web form, which could be used to compose an email to any recipient or group thereof. This was removed because there is no guarantee that the user will need or include email addresses in his database. It was replaced by a configuration option: if the user specifies that a certain field (or fields) in his database contains email addresses, those fields get displayed as mailto: links.

2.3.4 Postponed Functionality

2.3.4.1 Nodewise Connectrix could currently be best described as alpha-version software. The interesting parts work, but it's still got a long way to go before it's ready for public release.

2.3.4.2 The entire installation package has been postponed to a later release. Functionally speaking, this means that it will be up to whichever team decides to continue this project to design and implement a good installation package. This functionality was postponed because writing the proper installer is at once highly time-consuming (it needs to exhibit complex behavior to configure a web server from a dry start), and tangential to the goals of this project. However, in order to properly let an inexperienced user set up their web database, it will have to be written eventually.

2.3.4.3 Similarly, there are many complex behaviors which could be written to make the web interface much more secure than it currently is. They will need to be incorporated into the code before it can be considered ready for release, but at this point, it is sufficient to include just enough security to discourage random web trolls from attempting to break into the design and test machine.

2.3.4.4 Finally, though it would be useful to be able to load and save the state of the Development Tool (to create 'snapshots' of various useful configurations), and work was put into developing this functionality, it turned out to be overly complex to actually make it work. Thus, it is postponed until further

developers become available to refine this project. Though there is no indication that these functions have even begun to be supported in the UI that the user sees, documented functions do exist in the code which should give a useful starting point for further developers to work from.

3. REVIEW OF PROGRESS

3.1 Distribution of Work

3.1.1 Work was divided fairly evenly through the duration of the project. The design work happened collaboratively, and during initial installation and package testing both team members participated basically equally. Implementation was more divided, but still fairly even: Peter implemented the majority of the Development Tool, while Ryan designed and implemented the language and parser. Though Ryan wrote more of the web forms, Peter wrote more of the documentation.

3.2 Progress vs. Time

3.2.1 This project was run, from start to finish, in a single, seven-week term. This put everything on a very short schedule.

3.2.2 Approximately the first third of the project time was spent in design and configuration. This included setting up Ryan's computer as a Linux box with all the requisite software, and a good number of packages which ended up getting discarded from the final design. However, by week 3, the design had progressed to Revision 2 and we began to implement the final product.

3.2.3 The vast majority of the code was complete by week 6, though there were still a few tweaks to the web interface and Development Tool still to be implemented at that point. However, that was the point at which the focus shifted from developing new code to documentation and testing.

3.3 Lessons Learned

3.3.1 The lessons we learned were threefold: Designing a good user interface is a surprisingly complex task; it is good to plan ahead ambitiously, but check for feasibility while doing so; and simplifying a complex task such that it is performable by someone unqualified to do so, is itself a complex task.

3.3.2 While we spent a good amount of time designing the user interface for the Design Tool, it was clear that users with a little bit of competence might want to design the web interface themselves. Everybody has different tastes in terms of website design. Therefore, though we provide a default web interface, it is simple for the user to customize it however they like.

3.3.3 In terms of the Development Tool, the biggest challenge for the user interface was to make everything clear, simple, and helpful. Functionally, what this means is that we spend quite a bit of time designing good error messages and useful results when those errors were generated.

3.3.4 As you can see from the Revision History, we designed the project fairly ambitiously, and then subsequently had to pare down extraneous work, and change the implementation plan repeatedly based on what features were supported. Though we ended up able to implement all the key features of the project, it would have been good to discover earlier which components and packages just work together, as opposed to requiring hours of configuration time. On the other side of the coin, if it had turned out that the Prolog interface had just worked with only minor configuration, there is a good chance that we would have been able to implement more of the original design.

3.3.5 Finally, the main focus of the project was always to simplify a complex task, and make it possible for the inexperienced user. However, along the way, we ran into many challenges and hurdles that we simply hadn't anticipated. We spent many hours in the first few weeks trying to configure incomprehensible packages which advertised themselves as performing in a certain way, but only did so after much configuration. This gave us a powerful object lesson in what the user shouldn't have to deal with. There were a few packages, however, that just worked without any configuration at all. That became one of our goals for this project: to make a system that, in the end, just works.

4. CLOSING

4.1 Project Overview

4.1.1 Nodewise Connectrix was initially conceived as a setup utility, to enable non-programmers to create and administer a web interface to a complex database system. In that respect, we have succeeded; the interesting and useful parts are complete. However, the project could still use quite a bit of polishing and finishing before it gets released to the public. If this were a commercial product, I would peg it as being in a mid alpha stage right now.

4.1.2 Particularly, the installer/configurer for the software on which this project depends (Apache, PHP, MySQL) is vital to the success of the original intent of the project. However, it would require enough extra work that there simply wasn't enough time to do it.

4.2 Conclusions

4.2.1 Nodewise Connectrix is sufficiently developed and functional that it certainly expedites the process of creating and administering a database with a web interface. However, to do a clean install still requires manual configuration of the server, and will probably involve the user rewriting the output web pages just to make them look prettier. This implies a required level of competence far higher than that of the intended end-user. Fixing these problems, and polishing and adding to the feature set, would perhaps make a good MQP.

4.2.2 Despite this incompleteness, this project accomplished the goals initially set for it. Setbacks were overcome and documented, and everything was accomplished according to schedule. The project is complete enough that both of us feel comfortable leaving it at this stage.

4.2.3 Developing this project was an adventure. It was the first time either of us had attempted a project this big with such an aggressive timeline. However,

everything seems to have come together satisfactorily. We're both proud to have worked on this project.

Peter Goodspeed
Ryan Munro
4 March 2004

Appendix A1: Development Tool Specifications

From http://amalcon.res.wpi.net/docs/devtool.html:

Configuring your Database with the Development Tool

We will assume that everything has been properly installed and configured such that the Devtool now runs properly. If that has not yet happened, this is not the right help file for you.

The Development Tool is a configuration utility which is used to design the basic structure of your data.

As you know, Nodewise Connectrix represents Hierarchical Tree Structures as large, interconnected graphs of nodes, connected to each other with links. The Development Tool helps you design the structure of these Nodes and Links.

1. The first two panels of the Development Tool are designated 'Nodes' and 'Links'. These each have the same format. In each of these, you designate the fields which will appear in every Node and Link, respectively.

   There are several rules which field names must follow:
   - Field names are *case-insensitive*. This means that capitalization does not matter at all; the names will be converted to uppercase later.
   - Field names must be composed of word characters. This means that they must be in the set [a-zA-Z_0-9]. Any other characters are disallowed
   - Field names may not begin with the string 'user_'. This is reserved for language functions.
   - Field names may not be any of the following reserved keywords:
     - *and*
     - *or*
     - *not*
     - *bidirectional*
     - *id*
     - *type*
     - *reverse*

2. The third panel is entitled 'Types' and comes in two sections, Nodes and Links. This is where you define what types nodes and links may be. This is

the most basic level at which you can define what your database is about.

For example, if you are designing a family tree, each node will be of type 'person', and you can have links of type 'parentof' and 'marriage'.

Alternately, if you are designing a business organization chart, you will probably want to support nodes of types 'employee', 'department', 'division', 'group', and 'company'. You will also want links of type 'manager', 'controlling_authority', 'project_partner', etc.

3.

The final panel, 'User Code', is the most complex panel and the most useful. This is where you define the functions that specify how things are related to each other in implied ways. For example, on a family tree, it will e useful to define a function 'sibling', which is anyone who has the same parents as you, but isn't you. In the same way, you can define 'uncle' which is anyone who is male, and is a sibling of your parent, 'aunt' similarly, and 'cousin' as any decendent of any uncle or aunt.

The User Code section supports recursion, which means that a function can reference itself. For example, you could define the function 'ancestor', which means 'your parents, or any ancestor of your parents'.

You should always create as many relations as possible as implied relations defined by this User Code, instead of defining more Link Types and putting them in manually. This is because the more things that are generated automatically, the less maintenance you have to do by hand to keep the database up to date.

For more information about how to use the User Code section, read the User Code Tutorial.

Appendix A2: FunctionCode Specification

From http://amalcon.res.wpi.net/docs/usercode.html:

Nodewise Connectrix FunctionCode Specification

Introduction

What Is FunctionCode?

FunctionCode is not a programming language. It is a language for describing how relationships can be built upon other relationships. For example, the FunctionCode to say "Someone is my child if I am their parent" is:

(child me) = (reverse parent me)
That probably doesn't make much sense yet. Don't worry; it will soon.

Function Definitions

A function is the actual construct describing how a relationship is constructed from other, established relationships. A function is defined in the example:

(child me) = (reverse parent me)
FunctionCode's sole purpose is to describe these functions; as such, each line in FunctionCode is a function definition.

The function name above is **child**. A function name is what you will use to identify the function later. It is what the web interface will display to the user when your function is an option for a given field. It is also what you will use later on, should another function use this function.

Function Names

A function name should be a sequence of capital or lowercase letters, numbers between 0 and 9, and underscores (_). Thus, a function name like number1_function would be OK, but a function name like #1_function would not. Also, a function name may not be the same as a node property name or a link type name, as defined in the first three pages of the devtool, nor may it be the same as another function name.

Parameters

In the child function above, me is what's called a parameter. Every function in FunctionCode has exactly one parameter, which is placed as shown in the example. The parameter is a value to be filled in later. In other words, giving a parameter is like saying "I want some value here, but I don't know what yet." It is set to the current node when doing a relative search in the web interface. A

function does not have to use its parameter; however, to describe most meaningful relationships, it is necessary.

A parameter is also a sequence of capital or lowercase letters, numbers between 0 and 9, and underscores(_), just like function names. They do not, however, face any further restrictions; your parameters may have the same names as functions, node properties, link types, or even other parameters. It is, however, suggested that you make them unique to avoid confusion later.

The remaining part of the function definition, (reverse parent me) is known as a statement. Statements are described below.

To recap, a function definition looks like this:

(function_name parameter) = (statement)

Statements

For the following, we'll assume the following properties and link types are defined. If you want to copy the functions and follow along, you probably want to define them:

Link type= parent src=the child dest=the parent
Node property= gender type=text
Furthermore, wherever you see "parent" or "gender" in this section, you could replace it with the name of any link type or node property, respectively.

Unlike in many programming languages, in FunctionCode, a statement cannot exist on its own. It may only exist as part of a function definition. Statements in FunctionCode may take several forms.

Note on reading statements: Statements are best read by starting from the innermost parenthesis and working your way out. Try to do this with each of the statements in this section, and try to figure out what each one is doing. This practice will make writing them much easier.

Constants and Parameters

Though not strictly statements, there are four things which may be placed anywhere a statement may go:

> The current function's parameter, in plain text, unquoted. This will become the value of the current node in the web interface, or the value of the given parameter in a function call statement.
> A string literal, in double-quotes, i.e. "male"

A numerical literal, in single-quotes, i.e. '2', '3.14', etc.
A date or time literal, in single-quotes. Dates must follow the YYYY-MM-DD format, i.e. '2004-06-07'. Times must follow the HH:MM:SS format, i.e. '12:30:00'.

Link Trace Statements

The first of these is the link trace, which looks like the underlined:
(parentfunc you) = (parent you)

This (not particularly useful) function retrieves all parents of its parameter. This is not particularly useful because links appear in all the same places as functions in the web interface. When combined with other statements, however, the link trace statement becomes a powerful tool. For example, the following function will retrieve all grandparents of the parameter:

(grandparent me) = (parent (parent me))

Reverse Link Trace Statements

Similar to the link trace statement is the reverse link trace statement. It looks like this:

(child me) = (reverse parent me)
This function will retrieve all children of its parameter. This is not particularly useful because links in fact appear in both directions whenever they appear in the web interface. Again, it becomes more useful when we combine it with other statements. For example, the following function will retrieve all of your siblings plus yourself (we'll learn how to leave yourself out later):
(siblingorself me) = (reverse parent (parent me))

Node Property Statements

The third type of statement is the node property statement. This statement retrieves the given property of the nodes it's passed. It looks like this:

(mygender me) = (gender me)

Note that this function actually results in an error when used in the web interface, because the web interface wants to get node IDs, and this will give it names. This statement is actually useless except when combined with the reverse node property statement.

Reverse Node Property Statements

The reverse node property statement retrieves all nodes with the given property.
(potentialroommate me) = (reverse gender (gender me))
This retrieves all nodes with the gender name as the parameter. This works because the inner node property statement (name me) gives the gender of its parameter. The reverse node property statement then finds all the nodes with that gender.

Special Statements

There are three types of special statement: the AND, OR, and NOT statement. It doesn't matter if the AND, OR, or NOT in these statements is capitalized. The NOT statement finds all values not in its parameter. Constants put into a not statement will be interpreted as IDs. NOT statements look like this:
(notme me) = (not me)
This function will find all nodes OTHER than its parameter.

An AND statement finds all values in BOTH its parameters. AND and OR are special cases in that they have two parameters. The AND statement looks like this:

(father me) = (and (parent me) (reverse gender "male"))
This function will find all of its parameter's male parents. Also, as promised:
(sibling me) = (and (reverse parent (parent me)) (not me))
This function will find all of its parameter's parents' children who are not the parameter; in other words, siblings.

The OR statement finds all values in EITHER parameter. It is syntactically identical to the AND statement:

(parentchild me) = (or (reverse parent me) (parent me))
This function will find all nodes who are either parents or children of the parameter.

Function Call Statements

The last type of statement is a function call, which looks like the underlined (where child is the name of a function):
(something you) = (child you)

A function call invokes another function you have defined. In this case, the function named something would do the same thing as child. This is not very useful on its own, but again it can be combined with other statements to save a lot of typing:

(cousin me) = (child(sibling(parent me)))

A very powerful technique called recursion takes advantage of the function call statement:

Recursion

Recursion is a technique used to automate overly repetitive tasks and reduce the size of your code file by not forcing you to explicitly account for each repetition. Essentially, one defines a function in terms of itself. Writing a recursive function is like saying "Someone is my descendant if they are my child or a child of one of my descendants." In fact, the code for that looks like this:
(descendant me) = (or (reverse parent me) (reverse parent (descendant me)))
Recursive statements in FuncionCode seem like they could potentially go on forever. While normally they could, a given query is only allowed to go so many function calls deep. This number is defined in metadata.php as $meta_maxdepth. Its default is 250. The higher this number is, the more exhaustive recursive searches will be. The lower this number is, the faster recursive searches will be. In general, the more nodes you have in your database, the larger this number needs to be. A good rule of thumb is that you should set this number to twice the maximum expected number of nodes in the database. Depending on what your recursive functions look like, it may need to be higher, or it may benefit from being lower.

Conclusion

FunctionCode is a language with a lot of expressive power. With all this expressive power, it could take some getting used to. It is suggested that you take a look at the FunctionCode for the familytree application, and try to get familiar with it. Happy FunctionCoding!

Appendix A3: Glossary

From http://amalcon.res.wpi.net/docs/glossary.html:

Glossary

Apache

Apacheis a Web Server. This is a program that runs on a computer, and allows other computers to connect to it and request web pages. Every web page on the Internet is on one server or another.
This particular program is free.

Boolean

A boolean field contains one of only two possible values: *true* or *false*.

Data Type

There are many different types of data which can be used in your database:
Boolean
Date
Integer
Real Number
String
Text
Time

Date

A date field contains a representation of a specific date. This must follow the following format: **YYYY-MM-DD** This means that the years come first, followed by the months, followed by the days. All of these are separated by dashes. Moreover, all of these fields are numerical, and must be filled in.
2004-02-25 is a *valid* Date.
02-25-2004, 2004-2-25, 25-02-2004 are all examples of *invalid* Dates.

Devtool; Development Tool

The Development Tool is a program that you run when you are setting up Nodewise Connectrix. It generates many things for you, including the SQL Table Definitions, and many of the PHP files which you see after setup is complete.

Field

A named representation of a single simple piece of data. A field has a data type which defines what kind of information is put into it. You specify what fields exist, and what data types each has, in the Development Tool.

Additionally, you may decide to require certain fields. Think carefully before doing this. Though there are some benefits to requiring fields, you may want to add an entry into your database later, and be stuck, because you don't have access to a field that you required here.

HTML

HTML stands for HyperText Markup Language. It is a very simple language which browsers understand. All Web pages are written in HTML. The browser reads the HTML, and renders the page according to the instructions given in the HTML so that it looks nice to you, the user.
To see an example of HTML, go to the 'View' menu of your browser, and choose the 'View Source' or 'Page Source' option. This will show you the HTML used to create this page.

Integer

An Integer is a whole number. An Integer field can accept whole numbers from -2147483648 to 2147483647. However, it cannot ever accept decimal or fractional numbers.

Link

A Link is a pathway between two nodes. It indicates a relationship between the two of them. A Link may contain one or more fields, depending on how you set things up in the Development Tool.
A Link always has a type. The type of a link indicates what the exact relationship between the two nodes is. You can define what types a link can be in the Development Tool.

Node

A Node is a collection of fields. Each field in a node has a particular value; this is where the majority of your data will be.
A Node always has a type. The type of a node indicates what that node represents. You can define what types a node can be in the Development Tool.

PHP

PHP is a recursive acronym for 'PHP: Hypertext Processor'. PHP is a programming language used to generate the HTML pages your browser uses to show your data to you. Don't worry about your users not having it! Like SQL, PHP runs on the server, and returns plain HTML that any browser can use.

Real Number

A Real number is any fractional or integer number. Allowable values are -1.7976931348623157E+308 to -2.2250738585072014E-308, 0, and 2.2250738585072014E-308 to 1.7976931348623157E+308. NOTE: Although Real numbers seem more versatile than Integers, they have a serious flaw: All Integers are always perfectly precise. Real numbers, on

the other hand, have an inherent rounding error which over the course of many calculations can add up and cause incorrect results.
However, unless you are going to be performing repeated calculations on your data without periodic recalibration, you can feel safe using Real Numbers.

Recursion

Recursion is a programming technique used to automate repetitive tasks. A recursive function calls itself on a subset of the input stream, until there is only a single unit of information left. Then, it operates on that single unit of information, and returns the result to itself, and it continues to add to the result set returned by itself until it has arrived at a complete solution.
An example of a recursive function is one which returns a person's ancestors, in a family tree: An ancestor is a person's parents, plus all the ancestors of a person's parents.

Server

The Server is the computer on which you installed Nodewise Connectrix. As part of the installation process for Nodewise Connectrix, you also installed Apache, PHP, and MySQL. All of these are free software packages which run on the Server, and combine to allow Nodewise Connectrix to run.

SQL

SQL stands for Structured Query Language. It is a standard protocol for the storage of large amounts of data. Nodewise Connectrix uses MySQL, a free implementation of SQL, to store your data.

String

A String is a type of field which contains a short amount of text. A String may contain up to 255 characters. A String is good for, as an example, holding somebody's name.

Text

A Text field contains an effectively unlimited amount of text. It can hold just under 4 GB of characters, which works out to over 500 very long novels. A Text Field is good for holding anything too large to fit into a String.

Time

A Time field contains a representation of an interval of time, stored in HHH:MM:SS format. TIME values may range from '-838:59:59' to '838:59:59'. The reason the hours part may be so large is that the TIME type may be used not only to represent a time of day (which must be less than 24 hours), but also elapsed time or a time interval between two events (which may be much greater than 24 hours, or even negative).

You may specify a Time value in three formats: As a number in HHMMSS format, provided that it makes sense as a time. For example, 101112 is understood as '10:11:12'. Alternately, you can use one of the following formats: 'HHH:MM:SS', or 'D HH:MM:SS'. Here, 'D' refers to the number of days, and is a number between 0-33.

Tree; Tree Structure

A way of arranging data. In a Tree Structure, data is gathered into clusters, called nodes. Nodes are connected to each other with links. Any node may have any number of links connecting it to other nodes.
The reason this is called a tree structure is that the links are *directional* ; that is, they have a forward and a backward direction. This makes sense in most contexts, like parent-child relationships: the two are not interchangeable.
However, there are other contexts in which a link *should* be bidirectional, such as in the case of marriage. Luckily, Nodewise Connectrix doesn't force you to use bidirectional links; every link individually knows whether it should be bidirectional. You can set whether links default to be bidirectional in the Development Tool.

Type

Both nodes and links are required to have at least one Type. A type determines what each Node or Link represents. For example, a node might have type 'person', and a link might have types 'parentof', or 'marriedto'. Any particular node or link will have only one type, but there can be any number of possible types that nodes and links can be. You set up which types nodes and links can be in the Development Tool.

Web Interface

The Web Interface to Nodewise Connectrix is the web page, hosted by your Apache server, powered by PHP. It can be found by using your web browser to go to the point in your document root where you put your .php files.

Appendix A4: Language Definition

From http://amalcon.res.wpi.net/docs/language-def.txt:

```
<function>  ::= (<func-id> <param-id>) = <statement>

<func-id>  ::= [Any string of word characters as defined by Perl regex's
               not otherwise used]

<param-id>  ::= [Like a func-id]

<statement> ::= (<func-id> <param>)
                        // For evaluating a function
            ::= (<link> <param>)
              // For finding all those who are a single link of this
                        // type from the given set of people, in the forward
              // direction
            ::= (<property> <param>)
              // For finding a property value of a given set of people
            ::= (and <statement> <statement>)
                        // For finding people satisfying both of two conditions
            ::= (or <statement> <statement>)
                        // For finding people satisfying at least one of two
                        // conditions
            ::= (not <statement>)
                        // For finding all people not fitting some criteria,
                        // or all values of a property other than those given
            ::= (reverse <link> <param>)
                        // For following a link backwards
            ::= (reverse <property> <param>)
                        // For finding all people with a given property value
            ::= (with <property> <param>)
                        // Synonym for (reverse <property> <param>

<link>      ::= Name of a link
<property>  ::= Name of a property

<param>     ::= [Value that can be contained in a property OR an id.
                Strings double-quoted, everything else in MySQL format
                single-quoted]
            ::= <param-id> //of the function currently being defined
            ::= <statement> //to be evaluated
```

References

All research for this project was conducted on the Internet. Though there were far too many pages consulted to list individually, the following document roots are good places to find similar information to that which we had:

http://www.apache.org
http://www.gentoo.org
http://www.mysql.com
http://www.php.net
http://java.sun.com/j2se/1.4.2/docs/api/
http://www.google.com