# Using Prolog for Internet Protocol Applications

by

Lin Jiang

B Sc , Brandon University, 1993

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

We accept this thesis as conforming
to the required standard

---

Dr M Levy, Supervisor (Dept of Computer Science)

---

Dr B Kapron (Dept of Computer Science)

---

Dr J Provan (Dept of Mechanical Engineering)

---

Dr Meyer Nahon (Dept of Mechanical Enginering)

© Lin Jiang, 1995

UNIVERSITY OF VICTORIA

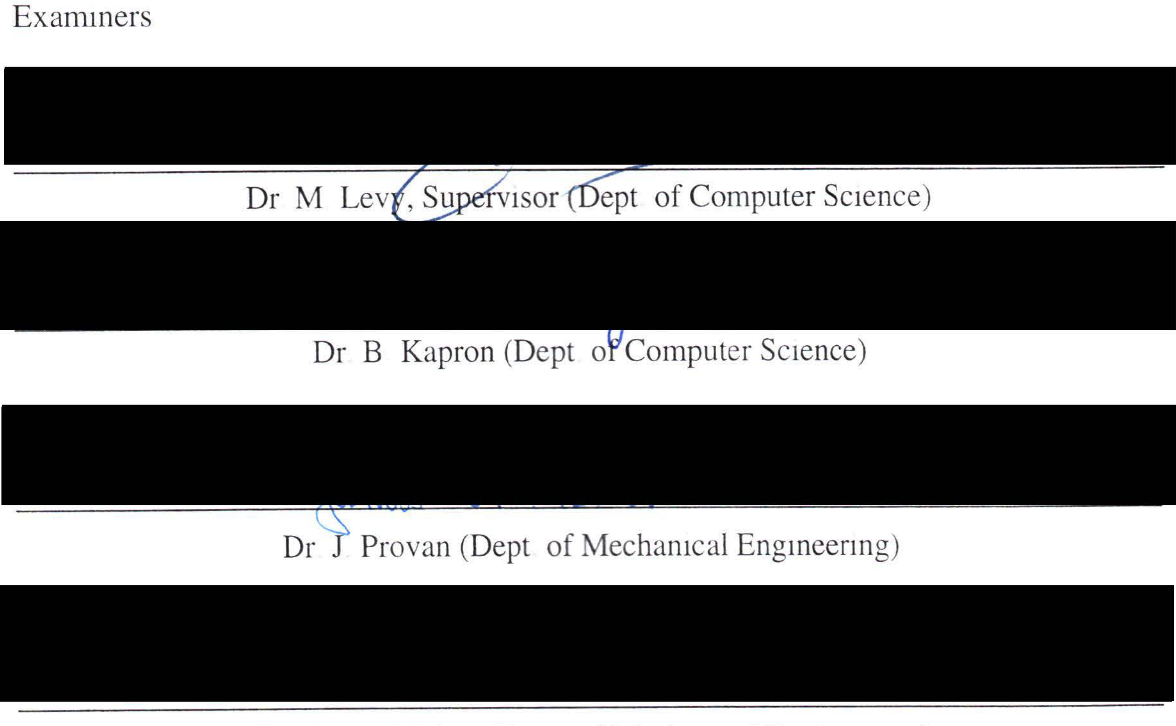
*All rights reserved This thesis may not be reproduced
in whole or in part by mimeograph or other means,
without the permission of the author*

Supervisor Dr Michael Levy

# ABSTRACT

As more and more information becomes available on the network, users become more and more eager for tools that will help them filter the information and find things of interest to them This thesis explores the potential of Prolog as a suitable programming language for writing such a tool An experimental HTTP server was implemented to test the suitability of Prolog for writing TCP/IP applications This server provides users with personalized information, keeps track of users' activities on reading news and allows centralized email message storing

Examiners



Dr M Levy, Supervisor (Dept of Computer Science)

Dr B Kapron (Dept of Computer Science)

Dr J Provan (Dept of Mechanical Engineering)

Dr Meyer Nahon (Dept of Mechanical Engineering)

**List of Tables** v

**List of Figures** vi

**Acknowledgment** viii

**1 Introduction** 1

1 1 Overview of Prolog 1
1 1 1 Basic Components of Prolog 1
1 1 2 Backtracking 2
1 1 3 Recursion 3
1 1 4 Prolog and Databases 3
1 2 The Client-Server Paradigm 4
1 3 An HTTP Server in Prolog 5
1 4 Related Work 6
1 5 The Remainder of this Thesis 7

**2 Benefits of Prolog for Client-Server Applications** 8

2 1 Intelligent Client-Server Applications 8
2 2 Prolog in Machine Learning 9
2 2 1 Relational Descriptions of Objects and Concepts 11
2 2 2 Attribute Descriptions of Objects 12
2 2 3 Reasoning 12
2 3 Prolog in Natural Language Understanding 15
2 3 1 Grammar Rules in Prolog 16
2 3 2 Defining the Meaning of Natural Language 18

**3 Concurrency** 23

3 1 DEC-10 I/O Model 23
3 2 The Levy/Rintoul Extensions 25
3 3 The catch and throw Mechanism 27
3 4 Attaining Concurrency 28
3 5 Comparison with Related Work 33
3 5 1 I C Prolog II 33
3 5 2 Delta-Prolog 35

**4 Prolog and Databases** 37

4 1 Relational Databases 37
4 1 1 The Relational Model 37
4 1 2 Relationships in the Relational Data Model 39

4.1.3 Data Processing Using Relational Languages 40
4.2 Prolog and Database Systems 45
4.2.1 Relationships between Prolog and Relational Databases 45
4.2.2 Prolog as a Query Language 47
4.3 Integrating Prolog and Relational Databases 53

**5 A Personalized HTTP Server 54**

5.1 The W3 System 54
5.1.1 The Software 54
5.1.2 The Address System 55
5.1.3 The Protocol 55
5.1.4 The Language 57
5.2 A Personalized HTTP Server 58
5.2.1 Overview of the Server 58
5.2.2 Authentication 63
5.2.3 Concurrency 66
5.2.4 Communication 67
5.2.5 Handling Requests from Clients 74
5.2.6 Related Work 80

**6 Conclusion 81**

**Glossary 83**

**Bibliography 84**

# List of Tables

Table 4 1 OFFERING relation 38

Table 4 2 EXAM relation 38

Table 4 3 Correspondence between similar concepts in Prolog and databases47

# List of Figures


| | | |
|---|---|---|
| Figure 1.1 | Overview of a database system | 4 |
| Figure 1.2 | My HTTP server acts as an intermediate layer between the HTTP client and NNTP and POP servers | 6 |
| Figure 2.1 | A backward chaining interpreter for if-then rules | 14 |
| Figure 2.2 | A forward chaining interpreter for if-then rules | 15 |
| Figure 2.3 | DCG that defines the meaning of very simple sentences | 19 |
| Figure 3.1 | The Levy/Rintoul extensions to DEC-10 I/O model | 26 |
| Figure 3.2 | A client-server program using extensions to DEC-10 I/O Model | 27 |
| Figure 3.3 | An example of catch and throw | 28 |
| Figure 3.4 | catch and throw used with listen/1 | 29 |
| Figure 3.5 | Parent process forks N child processes simultaneously | 30 |
| Figure 3.6 | Parent process waits for child process to finish before forking another process | 31 |
| Figure 3.7 | Child process creates another process before terminating itself | 32 |
| Figure 3.8 | An example server in ICP | 34 |
| Figure 3.9 | Squares example in Delta-Prolog | 35 |
| Figure 4.1 | Examples of algebraic operations | 42 |
| Figure 4.2 | Relation PARENT and PERSON | 48 |
| Figure 4.3 | The IDB relations FATHER and MOTHER | 50 |
| Figure 5.1 | An example of an HTTP request and response | 57 |
| Figure 5.2 | User's homepage | 59 |
| Figure 5.3 | User's page on NEWS | 60 |
| Figure 5.4 | User's page on NEWS articles | 61 |
| Figure 5.5 | User's page on login to MAIL server | 62 |
| Figure 5.6 | User's page on MAIL messages | 63 |
| Figure 5.7 | authenticate/1 | 64 |
| Figure 5.8 | check_identity/4 | 65 |
| Figure 5.9 | Concurrency in the server | 66 |

| | | |
|---|---|---|
| Figure 5 10 | Example server that handles file transmissions | 71 |
| Figure 5 11 | http_serve/2 | 74 |
| Figure 5 12 | process_request/3 | 75 |
| Figure 5 13 | Handling requests | 76 |
| Figure 5 14 | Prolog CGI | 77 |
| Figure 5 15 | An example of the user database file | 78 |
| Figure 5 16 | Code that generates the page in Figure 5 3 | 79 |
| Figure 5 17 | Printing the title of an article | 79 |

# Acknowledgment

Special thanks to my supervisor Dr Michael Levy for his guidance throughout the program. Thanks to Graduate Studies for providing me with the Fellowship that makes this degree possible. Thanks to my dear friends who are always there for me and who made my life in Victoria fun and enjoyable.

# Chapter 1

# Introduction

This thesis explores the features of Prolog and their use in writing client-server programs. An HTTP server was implemented to experiment the suitability of Prolog in developing client-server software.

## 1.1 Overview of Prolog

### 1.1.1 Basic Components of Prolog

Prolog is a programming language for symbolic, non-numeric computation. It is *declarative* rather than procedural in the sense that it specifies *what* to do instead of *how* to do it. It is especially well suited for solving problems that involve objects and relations between objects. The three basic statements of Prolog are facts, rules and queries. A Prolog program is a set of facts and rules that define relationships between objects. For example, the following statements are facts that define the relation *child*:

```
child(frank,rob). % defines frank is rob's child
child(owen,alan). % defines owen is alan's child
child(alan,wesley). % defines alan is wesley's child
child(alan,helen). % defines alan is helen's child
child(jim,rob). % defines jim is rob's child
```

Therefore, a deductive rule that defines the relation *grandchild* can be written based on *child*:

```
grandchild(X,Y) :-
   child(X,Z), child(Z,Y).

/* X is Y's grandchild IF there exist a Z such that X
is Z's child and Z is Y's child. */
```

A query is a question that can be asked about the relations defined by the facts and rules. For example, query *grandchild(owen, wesley)* will succeed and give the answer *yes* because there are facts that *owen* is *alan*'s child and *alan* is *wesley*'s child.

Query *grandchild(frank, wesley)* will not succeed because although there is a fact saying *frank* is *rob*'s child, there is no fact defining *rob* is *wesley*'s child.

We can also ask questions like who is *wesley*'s grandchild:

```
?- grandchild(X, wesley).
```

The answer is:

```
?- X=owen
```

When there are multiple answers to a query, the Prolog interpreter will give the answer one at a time upon request. For example, the answer of the following query

```
?- child(X, rob).
```

is

```
?- X = frank
```

Now if we enter a semicolon (;) after the answer, the interpreter will search for more answers, and it will either give another answer or say *no* if there is no more. In this case, the interpreter gives

```
?- X = jim
```

### 1.1.2 Backtracking

Let us consider the execution of Prolog queries. Take the query

```
?- grandchild(X, wesley).
```

as an example, in finding the solution, the interpreter:

1. finds the rule *grandchild/2* and substitutes *X* with *X* and *Y* with *wesley* and results in the subgoals: *child(X, Z), child(Z, wesley)*.
2. resolves the first subgoal *child(X, Z)* and matches it with the first fact *child(frank, rob)*, which results in *X* to be bound to *frank*, *Z* to *rob*.
3. resolves the second subgoal which is now *child(rob, wesley)*. It fails because there is no such a match. It then *backtracks* to find another solution to the first subgoal.
4. matches the first subgoal *child(X, Z)* with the second fact *child(owen, alan)* resulting *X* to be bound to *owen*, and *Z* bound to *alan*.
5. resolves the second subgoal which is now *child(alan, wesley)*. It succeeds this time hence finally gives the answer *X* = *owen*.

Backtracking is an intrinsic part of Prolog. As we can see from the above example, an important factor in backtracking is the ordering of the subgoals, the rules and the facts. Ordering is necessary to prevent infinite loops. It affects the speed of the searching process and good choices can have a considerable effect on the efficiency.

### 1.1.3 Recursion

Recursion plays a central role in Prolog. As with all recursive situations, there must be a rule to halt the recursion. The following is an example of a rule that is recursively defined:

```
descendant(X,Y) :- child(X,Y).
descendant(X,Y) :- child(X,Z), descendant(Z,Y).
```

### 1.1.4 Prolog and Databases

A database is a collection of information. Users can retrieve the information through some processing mechanism which provides a lookup facility (Figure 1.1).

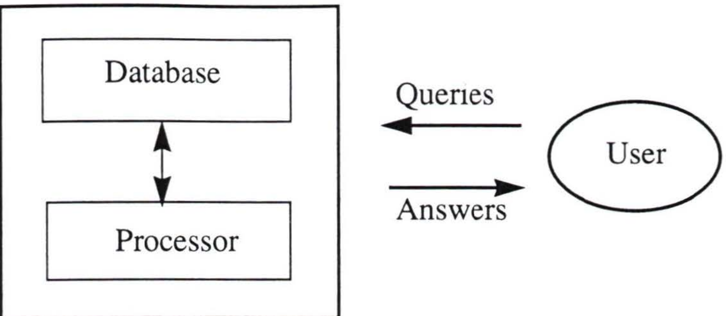



Figure 1 1 Overview of a database system

Prolog has all the built-in mechanisms for directly constructing database systems because the processing component or reasoning mechanism is a standard part of the Prolog system It has a powerful lookup facility and a straightforward logical reasoning ability Therefore, Prolog is good for implementing small deductive databases for knowledge-based systems

## 1.2 The Client-Server Paradigm

Client-server applications consist of one or more client programs and one server program running on separate machines that are connected over a network A client is a process which interacts with the user It forms requests in a predefined language for presentation to the server The client connects to the server and transmits requests to the server using a standard interprocess protocol The client also performs data analysis on the results sent back from the server and then present them to the user A server is a process, or a set of processes all of which must exist on one machine It is started before the clients are and waits passively for the clients to connect The server provides services to one or more clients by responding to clients' requests It acts either as a repository of data or knowledge or as a service provider

There are many benefits to client-server applications One benefit is application programs provide interfaces to end-users Since the server is a separate program from the

client, it is almost essential for the designer of the application to develop a clear interface between clients and servers. Another benefit is client-server applications support heterogeneity. Clients are not restricted to the same operating system or architecture as the server, as long as the host machines support a common communication protocol. Client-server applications also allow resource sharing. The client-server model has been widely used to make information easily accessible to end-users. An example of this is the W3 (World-Wide Web) client-server applications [1]. W3 can be viewed as a pool of knowledge, which allows people in remote sites to share information. A W3 client is a program that runs on a user's workstation. It allows the user to connect to W3 servers and retrieve information from the server sites. The information presented to the user can be in either hypertext, plain text, or multimedia format. The user can navigate in the network through hypertext links or specifying the URLs (Universal Resource Locators) for W3 sites. The communication protocol HTTP (HyperText Transfer Protocol) is stateless in the sense that the connection is held only for the duration of one operation, so that it does not tie up communication channels. The first widely used W3 browser was Mosaic developed by NCSA.

## 1.3 An HTTP Server in Prolog

To test the hypothesis that Prolog is suitable for distributed applications, I designed and implemented an HTTP server using the TOPIC Prolog system, augmented with TCP/IP [11]. The server acts as an intermediate layer between a Mosaic browser and the NNTP and POP servers Figure 1.2. It allows users to access NEWS and MAIL while keeping track of each user's activities. It handles different communication protocols HTTP (HyperText Transfer Protocol), NNTP (Network News Transfer Protocol), and POP (Post Office Protocol).

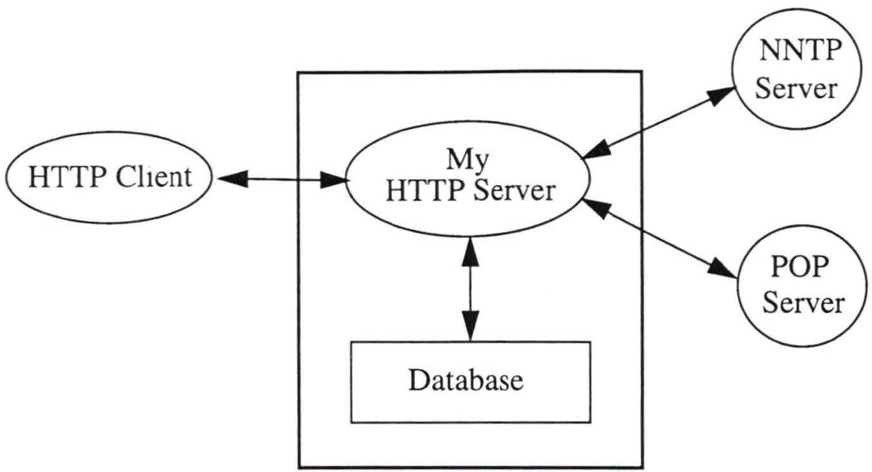



Figure 1.2 My HTTP server acts as an intermediate layer between the HTTP client and NNTP and POP servers

## 1.4 Related Work

TOPIC supports TCP/IP (Chapter 3). Although TCP/IP is not widely supported within Prolog systems, some systems do have the ability. Examples are SICStus Prolog and I. C. Prolog II. SICStus Prolog has predicates to manipulate sockets on UNIX systems. They are straight-forward interfaces to the corresponding C library routines with the same name, such as *socket/2*, *socket_bind/2*, *socket_listen/2*, and *socket_accept/2*. I. C. Prolog II supports TCP/IP in a similar way, that is, it has explicit predicates to create a socket, send a message and receive a message, and so on. TOPIC takes a different approach. It hides the details of TCP/IP from users and provides users with portable primitives. These are described in Chapter 3.

There are many existing W3 servers, written in different languages, and for different platforms. Examples are the CERN server, NCSA server and the MacHTTP server. These servers were implemented in C. There are no known HTTP servers written in Prolog.

## 1.5 The Remainder of this Thesis

Chapter Two of the thesis describes the features of Prolog that can be used to benefit the development of client-server software. Chapter Three explains how to attain concurrency using the Levy/Rintoul extension to DEC-10 I/O model. Chapter Four goes in some depth to discuss the relationship between databases and Prolog, followed by Chapter Five which gives an example of an HTTP server written in Prolog. Chapter Six concludes the thesis.

# Chapter 2

# Benefits of Prolog for Client-Server Applications

As networks grow in size, the client-server paradigm has become more and more popular. In fact, it has become so fundamental in peer-to-peer networking systems that it forms the basis for most computer communication. As a consequence, the complexity of the use of computers increases. Traditional approaches to client-server computing no longer satisfy people's need. Therefore, there is a growing need for client-server applications that exhibit *intelligent* behavior. This chapter will illustrate some of the features of Prolog that can be used to benefit the development of intelligent client-server applications.

## 2.1 *Intelligent* Client-Server Applications

Some of the existing client-server applications already indicate some shortcomings. One example is email. The existing mail programs present mail messages to users according to the order of their arrival without any processing. This brings inconvenience to users when they receive a large volume of mail messages every day. It is desirable that some intelligence be added to the mail client, such as sorting the incoming mail based on classification of the sender, the subject line or possibly the content of the message. Further more, an intelligent client should be able to *learn* from the user's behavior so as to take actions on the user's behalf, such as prioritize, delete, forward, sort and archive mail

messages. For example, should the user save a particular message after having read it, the mail client would add a description of this situation and the action taken by the user to its memory of examples. An intelligent client would observe and memorize how the user deals with different situations, so that when a new situation occurs, the client could predict the action(s) of the user, based on the examples stored in its memory, and take the action(s) for the user. Machine learning techniques can be used to implement intelligence in email clients. Prolog is a good choice of programming language for such applications.

Another example is NEWS. As more and more information becomes available on the network, users become more and more eager for intelligent tools that will help them filter the information and find articles of interest to them. With an intelligent NEWS client, a user would be able to choose areas of interests and train the client by means of examples of articles that should or should not be selected. The client would be initialized by giving it some positive and negative examples of articles to be retrieved. The client would perform a syntactic and semantic analysis of texts and remember the structure information about the article, such as the author, source, assigned indices, and so forth. Once the client is initialized, it would start recommending articles to the user. The user could give it positive or negative feedback for articles or portions of articles recommended so that the client could gather more and more knowledge about the taste of the user. The key part of such a process is the syntactic and semantic analysis of texts. To do this, we need a way to understand natural languages. Prolog is a language that is suitable for natural language understanding.

## 2.2 Prolog in Machine Learning

There are several forms of learning, ranging from "learning by being told" to "learning by discovery", with the learner's responsibility from "little" to "full". Between the two extremes lies another form of learning: learning from examples. Here the responsibility is distributed between the teacher and the learner. The teacher provides examples

for learning and the learner is supposed to make generalizations about the examples - that is, find a kind of theory that underlies the given examples.

The problem of learning concepts from examples can be formalized as the following: Let $U$ be the universal set of objects - that is, all of the objects that the learner may encounter. A concept $C$ is a subset of objects in $U$. To learn concept $C$ means to learn to recognize objects in $C$. In other words, once $C$ is learned, the system is able to recognize whether a given object $X$ is in $C$ or not, for any $X$ in $U$. An example of concepts is:

> The concept of an arch in the blocks world. The universal set $U$ is the set of all structures made of blocks in a blocks world. Arch is the subset of $U$ containing all the arch-like structures and nothing else.

For any kind of learning, we need a language for describing objects and concepts. There are two kinds of descriptions: *relational descriptions* and *attribute descriptions*. In a relational description an object is described in terms of its components and the relations between them. For example, a relational description of an arch may be as: the arch is a structure consisting of three components (two posts and a lintel), each component is a block, both posts support the lintel, the posts do not touch. In an attribute description we describe an object in terms of its global features. Such a description is a vector of attribute values. For instance, an attribute description of a particular arch may be: length = 8m, height = 6m, color = brown.

Once an object and a concept are defined, a decision rule is needed to establish whether the object belongs to the concept. Such a rule will determine whether the object matches the concept description. Matching between objects and concept descriptions can be formalized as a matching predicate or a matching function of two arguments:

```
match(Object, Concept_description).
```

### 2.2.1 Relational Descriptions of Objects and Concepts

In Prolog, it is easy to describe an object or a concept based on relations. For example, a possible representation for an object is:

```
Object = object(ListOfParts, ListOfRelations)
Example = +Object (positive example)
Example = -Object (negative example)
```

A concrete example of an arch object is:

```
+object(
        [A,B,C],
        [support(A,C),support(B,C),
         isa(A,rectangle),isa(B,rectangle),
         isa(C,rectangle)])
```

Here, A and B are the two posts and C is the lintel.

A concept description is more complex. Again, there are parts and relations. One way to describe it is to classify the relations into three categories:

1. must relations (those that have to be present),
2. desirable relations, and
3. must-not relations (those that must not be present).

Therefore, the representation of a concept is:

```
ConceptDesc = concept(Parts,Musts,Desirables,MustNots)
```

An example of the final induced description of the arch concept in general is:

```
concept(
        [part1,part2,part3],
        [support(part1,part3),support(part2,part3)],
        [isa(part1,rectangle),isa(part2,rectangle),
         isa(part3,stable_poly)],
        [touch(part1,part2)])
```

### 2.2.2 Attribute Descriptions of Objects

Attributes of objects can also be easily represented in Prolog. For example, possible attribute values for some objects from a camera image are:

size: small, large
shape: long, compact, other
holes: none, 1, 2, 3, many

Objects in the image can be represented as a set of Prolog clauses:

```
attribute(size,  [small,large]).
attribute(shape,  [long,compact,other]).
attribute(holes,  [none,1,2,3,many]).

example(screw,[size=small,shape=compact,holes=none]).
example(key,[size=large,shape=long,holes=1]).
example(pen,[size=large,shape=long,holes=none]).
example(scissors,[size=large,shape=other,holes=2]).
```

### 2.2.3 Reasoning

The process of learning from examples can be viewed as a process of building a generalized concept from studying the given positive and negative examples. It involves reasoning from given findings to a conclusion.

*If-then* rules are a natural form of expressing knowledge. They are generally conditional statements. Examples are:

- *if* condition P *then* conclusion C
- *if* condition S *then* action A

For if-then rules, there are two basic ways of reasoning: *backward chaining* and *forward chaining*, both of which are very easily implemented in Prolog.

With backward chaining, we start with a hypothesis, then we follow a chain of rules from the hypothesis to the pieces of evidence. This is Prolog's own built-style of reasoning. For example, the if-then rule:

*If* the hall is wet and the kitchen is dry *then* there is a leak in the bathroom

can be described in Prolog as

```
leak_in_bathroom :-
        hall_wet,
        kitchen_dry.
```

The observed pieces of evidence can be stated as Prolog facts

```
hall_wet.
bathroom_dry.
```

The hypothesis can now be checked by

```
?- leak_in_bathroom.

no
```

The syntax used above can be easily tailored to our taste by using Prolog operator notation. For example, we can choose "if", "then", "and" and "or" as operators, declared as

```
:- op(800,fx,if).
:- op(700,xfx,then).
:- op(300,xfy,or).
:- op(200,xfy,and).
```

Therefore, if-then rules can be written as

```
if hall_wet and kitchen_dry
then leak_in_bathroom.

if hall_wet and bathroom_dry
then problem_in_kitchen.

...
```

Observable findings can be stated as

```
fact(hall_wet).
fact(bathroom_dry).
```

Figure 2 1 shows a backward chaining interpreter for if-then rules

```
:- op(800,fx,if).
:- op(700,xfx,then).
:- op(300,xfy,or).
:- op(200,xfy,and).

is_true(P) :-
      fact(P).

is_true(P) :-
      if Condition then P, % A relevant rule
      is_true(Condition).  % whose condition is true.

is_true(P1 and P2) :-
      is_true(P1),
      is_true(P2).

is_true(P1 or P2) :-
      is_true(P1)
      ;
      is_true(P2).
```

Figure 2 1 A backward chaining interpreter for if-then rules

Unlike backward chaining which starts with a hypothesis and works backwards toward easily confirmed findings, forward chaining starts with some confirmed findings and works toward a conclusion, i e it reasons from the "if" part to the "then" part Figure 2 2 shows a forward chaining rule interpreter

```
forward :-
      new_derived_fact(P),
      !,
      write('Derived: '), write(P), nl,
      assert(fact(P)),
      forward
      ;
      write('No more facts').

new_derived_fact(Concl) :-
      if Cond then Concl,
      not fact(Concl),
      composed_fact(Cond).

composed_fact(Cond) :-
      fact(Cond).

composed_fact(Cond1 and Cond2) :-
      composed_fact(Cond1),
      composed_fact(Cond2).

composed_fact(Cond1 or Cond2) :-
      composed_fact(Cond1)
      ;
      composed_fact(Cond2).
```

Figure 2 2 A forward chaining interpreter for if-then rules

## 2.3 Prolog in Natural Language Understanding

Natural language understanding is a very complex subject It is very difficult to make a computer understand any natural language because of the variety and ambiguity of the meanings of words and the grammatical rules, both of which are essential information for understanding a language Therefore we need a representation system on the computer to provide good readability and enough flexibility to describe the dictionary and grammatical rules Prolog appears to be a computer language that is suitable to build such a system

### 2.3.1 Grammar Rules in Prolog

Most Prolog implementations provide a notational extension called DCG (Definite Clause Grammars). This makes it very easy to implement formal grammars in Prolog. A grammar stated in DCG is directly executable by Prolog as a syntax analyzer. DCG also facilitates the handling of the semantics of a language so that the meaning of a sentence can be interleaved with the syntax.

A grammar is a formal device for defining a language. For example, the following grammar in BNF notation

<*s*> ::= *a b*<br>
<*s*> ::= *a* <*s*> *b*

can generate sentences

*ab, aabb, aaabbb,* etc.

A grammar can also be used in the opposite direction, to recognize whether a given sentence belongs to some language. In this process, a sentence is disassembled into its constituents. Therefore, this process is also called *parsing*. To implement a grammar normally means to write a parser for the grammar. In Prolog, this is easily done by using DCG. The following is an example of DCG that describes the above grammar rules:

```
s --> [a],[b].
s --> [a],s,[b].
```

where terminal symbols are in square brackets hence making them Prolog lists, and non-terminal symbols are by themselves. Another example is a simple grammar that defines a move consisting of a sequence of steps:

```
move --> step.
move --> step,move.

step --> [up].
step --> [down].
```

DCG rules are automatically converted into normal Prolog clauses when they are consulted. The above examples are translated into:

```
s([a,b|Rest],Rest).
s([a|List1],Rest) :-
        s(List1,[b|Rest]).
```

and

```
move(List,Rest) :-
        step(List,Rest).

move(List1,Rest) :-
        step(List1,List2),
        move(List2,Rest).

step([up|Rest],Rest).
step([down|Rest],Rest).
```

In general, for a DCG rule

```
n --> n1,n2,...,nn.
```

if all *n1, n2, ..., nn* are non-terminals then the rule is translated into the clause:

```
n(List1,Rest) :-
        n1(List1,List2),
        n2(List2,List3),
        ...
        nn(Listn,Rest).
```

If some of *n1, n2, ..., nn* are terminals, they are directly inserted into the corresponding list. For example, the rule

```
n --> n1,[t2],n3,[t4].
```

is translated into:

```
n(List1,Rest) :-
        n1(List1,[t2|List3]),
        n2(List3,[t4|Rest]).
```

The grammars can be immediately used as recognizer of sentences. Such a recognizer expects sentences to be represented as the difference of the two lists *List1* and *Rest*. For example, *aabb* can be represented by lists [*a,a,b,b*] and [], or by [*a,a,b,b,c*] and [*c*] etc. The above DCG examples can be asked to recognize some sentences by questions:

```
?- s([a,a,b,b],[]).

yes

?- s([a,a,b],[]).

no

?- move([up,up,down],[]).

yes

?- move([up,up,left],[]).

no

?- move([up,X,up],[]).
X = up;
X = down;
no
```

### 2.3.2 Defining the Meaning of Natural Language

Prolog grammars are particularly well suited for the treatment of the meaning of a language, in particular a natural language. Arguments that are attached to non-terminal symbols of a grammar can be used to handle the meaning of sentences.

In defining the meaning of a language, the first question is how the meaning should be represented. Logic has been accepted as a good candidate for representing the meaning of natural language sentences because it allows subtle semantic issues to be dealt with. Interpretations of simple natural language sentences in logic can be constructed using the DCG notation, and encoded as Prolog terms.

For example, a natural way to express the meaning of a simple sentence "John paints" as a Prolog term is

```
paints(john)
```

Notice that *paints* here is an intransitive verb therefore the corresponding predicate only has one argument. Another example sentence is "John likes Annie". The formalized meaning of this can be:

```
likes(john,annie)
```

The verb *likes* is transitive and the corresponding predicate has two arguments. The DCG rules in Figure 2.3 defines the meaning of such simple sentences.

```
sentence(Meaning) -->
      noun_phrase(Actor),verb_phrase(Actor,Meaning).

noun_phrase(Actor) --> proper_noun(Actor).

verb_phrase(Actor,VP) -->
      intrans_verb(Actor,VP).

verb_phrase(Somebody,VP) -->
      trans_verb(Somebody,Something,VP),
      noun_phrase(Something).

intrans_verb(Actor,paints(Actor)) --> [paints].

trans_verb(Sb,Sth,likes(Sb,Sth)) --> [likes].

proper_noun(john) --> [john].
proper_noun(annie) --> [annie].
```

Figure 2.3 DCG that defines the meaning of very simple sentences

After consulting the above DCG rules, we can query about the meaning of a sentence:

```
?- sentence(M,[john,paints],[]).

M = paints(john)

?- sentence(M,[annie,likes,john],[]).

M = likes(annie,john)
```

The above example introduced some basic ideas, although it only handles the simplest sentences. When noun phrases contain determiners like "a" and "every", the meaning expressions become more complicated. The meaning of the sentence "A man paints" is definitely not *paints(man)*. The sentence really says: There exists some man that paints. In logic this is phrased as:

There exists an X such that
    X is a man and X paints.

Here, X is said to be *existentially quantified*. We can represent this by the Prolog term:

```
exists(X, man(X) and paints(X))
```

with *and* being an infix operator. To generalize the logic meaning of the determiner "a", we get:

There exists some X such that
    X has some property (e.g., *man(X)*) and
    some further assertion about X holds (e.g., *paints(X)*)

As a Prolog term, this can be represented as:

```
exists(X, Property and Assertion)
```

Similarly, the determiner "every" has its own logic meaning. For example, the logic interpretation of the sentence "Every woman dances" is:

For all X,
    if X is a woman then X dances.

which can be represented by the Prolog term:

```
all(X, woman(X) => dances(X))
```

where => being an operator. Determiner "every" thus indicates a meaning whose skeleton structure is:

```
all(X, Property => Assertion)
```

The corresponding DCG rules to achieve the desired meanings are trivial:

```
determiner(X,Prop,Assn,exists(X,Prop and Assn)) -->
        [a].
determiner(X,Prop,Assn,all(X,Prop => Assn)) -->
        [every].
```

Further more, nouns can be qualified by relative clauses. For example, "Every man that paints admires Monet". Here, "every man" is qualified by "that paints". This sentence logically means:

For all X
  if X is a man and X paints
  then X admires Monet.

This can be represented as the Prolog term:

```
all(X, man(X) and paints(X) => admires(X, monet))   (1)
```

with *and* having higher precedence than =>. Similarly, the meaning of the sentence "Annie admires every man that paints" can be represented as:

```
all(X, man(X) and paints(X) => admires(annie, X))   (2)
```

Once we extract the meanings from a natural language, we can translate the logical meanings into Prolog clauses. Then we can use these clauses to reason about the language. This in general requires some work, but in some cases such a translation is trivial. Here is an example from translating the meaning (1) and (2):

```
paints(john).
man(john).
```

```
admires(X,monet) :- man(X), paints(X).
admires(annie,X) :- man(X), paints(X).
```

Now we can ask the question "Does Annie admire anybody who admires Monet?":

```
?- admires(annie,X), admires(X,monet).

X=john
```

In this chapter, we looked at some features of Prolog for writing potential intelligent programs, which can be further used to benefit the development of intelligent client-server software. I implemented an HTTP server in Prolog which will be described in Chapter 5. This server does not have any of the fore-stated intelligence in it. It merely serves as an experiment of writing a server program using Prolog. But it could also be seen as the first step toward something better - truly intelligent client-server programs that will bring people benefits.

# Chapter 3

# Concurrency

Standard Prolog has no mechanism for inter-process communication. One way to add communication ability is to modify the existing DEC-10 I/O model. This chapter begins with discussing one way of extending the DEC-10 I/O model [11]. It then discusses how to obtain concurrency using the extensions. Finally, it compares this approach to other people's work.

## 3.1 DEC-10 I/O Model

In the DEC-10 I/O model, file manipulation predicates always refer to a file by its name. There may be more than one file open for input or output, but reading or writing attempts are always done to the *current* input or output stream. File manipulation predicates are shown below. The symbol '+' indicates that the argument after it is an input parameter. '-' indicates that the argument after it is an output parameter. '?' indicates that the argument may either be an input or an output parameter.

**see**(+*File*)

File *File* becomes the current input stream. *File* may or may not be already opened by the Prolog system. If it is not opened, the file is opened and the current input stream is set to this file. If it is already opened, the current input stream is set to this file.

**`seeing`**(?*FileName*)

*FileName* is unified with the name of the current input stream. If the current input stream is a file opened by *see/1*, *FileName* is unified with the file name. If the current input stream is user_input, it is unified with *user_input*.

**`seen`**

Close the current input stream and resets it to *user_input*.

**`tell`**(+*File*)

File *File* becomes the current output stream. *File* may or may not be already opened by the Prolog system. If it is not opened, the file is opened and the current output stream is set to this file. If it is already opened, the current output stream is set to this file.

**`telling`**(?*FileName*)

*FileName* is unified with the name of the current output stream. If the current output stream is a file opened by *tell/1*, *FileName* is unified with the file name. If the current input stream is user_output, it is unified with *user_output*.

**`told`**

Close the current output stream and resets it to *user_output*.

This model is simple but limited. But when it is enhanced with networking capabilities, it provides the basic functionality needed for inter-process communication. The above I/O primitives are implemented as library routines in the TOPIC Prolog system. Therefore, they can be modified to meet our needs without having to change the Prolog engine.

## 3.2 The Levy/Rintoul Extensions

The Levy/Rintoul extensions to the DEC-10 I/O model were introduced and implemented as part of the TOPIC project [11]. The goal of the extensions is to add inter-process communication capabilities and concurrency into Prolog for different platforms without having to modify the semantics of the language. This is achieved by introducing new primitives in the library of the Prolog system and modifying existing library routines. The Prolog engine remained untouched. Minor modifications to the DEC-10 I/O model were required to extend it for use in a networking environment. The inter-process communication protocol supported by this extension is TCP/IP. The extension is implemented to support TCP/IP on different platforms for the sake of portability. On UNIX systems, communication is done through the use of Berkeley Sockets, and on McIntoshes, it is done by using MacTCP. Yet to Prolog programmers, the interface is the same. A new channel type *port* (a Prolog structure with arity two) was introduced, the new primitive *listen/1* was added, and the meanings of the *see/1* and *tell/1* predicates were modified to allow opening of communication channels across a network. Once a connection between two processes is established, the two processes communicate using the same functions that are used to perform standard file I/O. Figure 3.1 shows the extensions.

**listen**(+*Channel*)

> *Channel* becomes the current input stream. *Channel* can only be a port address of the form *port(IPName,PortNumber)*. *Channel* must not be opened already. *listen/1* forks a new process that blocks and waits passively for a process to connect on the port number if the Prolog system is running on a multi-tasking operating system that supports process forking. An example of a port address is *port('csr.uvic.ca',2000)*.

**see**(+*Channel*)

*Channel* becomes the current input stream. *Channel* may be a file name or a port address of the form *port(IPName,PortNumber)*. If *Channel* is a file name, *see/1* acts as it did before. If *Channel* is a port address, the following action is taken: if no network connection is opened to this address, the connection is established and the current input stream is set to this stream. If an input network connection to this address is already open, the current input stream is set to this stream. If an output network connection to this address is already open, the current input stream is set to this stream and the output connection is unaffected. (Network connections are bi-directional).

**`tell`**(+*File*)

*Channel* becomes the current output stream. *Channel* may be a file name or a port address of the form *port(IPName,PortNumber)*. If *Channel* is a file name, *tell/1* acts as it did before. If *Channel* is a port address, the following action is taken: if no network connection is opened to this address, the connection is established and the current output stream is set to this stream. If an output network connection to this address is already open, the current output stream is set to this stream. If an input network connection to this address is already open, the current output stream is set to this stream and the input connection is unaffected.

Figure 3.1 The Levy/Rintoul extensions to DEC-10 I/O model

The capabilities of the enhanced model are a super-set of the capabilities of the original model. Therefore the semantics of existing programs using the DEC-10 I/O model is unchanged. An example of the use of these extensions is shown in Figure 3.2.

```
server(N) :-
      listen(port(X,N)),
      tell(port(X,N)),
      read(Fname),
      write_file(Fname),% writes file & EOF to cur output
      seen,
      told,
      server(N).

client(P,Fname) :-
      tell(P),
      see(P),
      write(''''),write(Fname),write('''.'),nl,
      tell(user_output),
      echo_file,% echoes from current input stream until EOF
      seen,
      tell(P),
      told.
```

Figure 3 2 A client-server program using extensions to DEC-10 I/O Model

This client-server program implements a simple form of remote file display without the consideration of concurrency The server is started first when the user enters *server(PortNumber)* at the Prolog prompt "?-", where *PortNumber* is the port number on which the server waits passively for clients to connect A client connects to the server by entering the query *client(Port,Filename)* in a separate Prolog session, where *Port* is the network address of the server and *Filename* is the file to be displayed Upon connection, the server reads in the file name sent by the client and sends its content to the client which then displays the content on its machine

## 3.3 The *catch* and *throw* Mechanism

The ISO standard for Prolog proposed a mechanism for handling exceptions This is the *catch* and *throw* scheme, which was implemented in TOPIC system The control structures *catch* and *throw* are provided for handling errors and other explicitly pro-

grammed exceptions. They make it possible to jump out of multiple levels of procedure calls in a single step.

The query *catch(Goal1,Arg,Goal2)* is like *call(Goal1)* except that if, at any stage during the execution of *Goal1*, there is a call to *throw(Arg)*, then execution immediately jumps back to the *catch* and proceeds to *Goal2*. Here *Arg* can be a variable or only partly instantiated; the only requirement is that the *Arg* in the *catch* must unify with the one in the *throw*, otherwise the *catch* statement fails. Thus, *Arg* can include information to tell *catch* what happened. Figure 3.3 shows an example. When entering the query

```
?- go(mom).
```

'*I am too busy.*' is printed. When entering

```
?- go(dad).
```

'*OK. Fear*' is printed.

```
empty_garbage(mom) :- write('I am too busy.'),nl.
empty_garbage(dad) :- write('OK.'),nl,throw('Fear').

go(Who) :- catch(empty_garbage(Who),Why,write(Why)).
```

Figure 3.3 An example of *catch* and *throw*

The *catch* and *throw* mechanism is implemented in TOPIC system. Therefore, every predicate is permitted to do a *throw*. This mechanism can be further used to achieve portability in attaining concurrency on different platforms.

## 3.4 Attaining Concurrency

On a UNIX system where process forking is supported, *catch* and *throw* are used in the implementation of *listen/1* in order to obtain the desired flow of control. In *listen/1*, the calling process forks a child process to wait on a port for a client to connect. The

parent process then does a *throw* (with the child process id as its argument) and it is up to the application program which *catches* it to determine to where the parent process should continue executing. The child process keeps listening on the port until a client connects. It then goes to serve the client's request. Figure 3.4 gives a simple example of a server that serves one request at a time. As soon as the user starts the serve at the Prolog prompt "?-", the parent process comes back to the top level, the Prolog prompt, "?-", and is ready to handle another query, while the child process is waiting on a port for connection. The user can issue another query *serve(Port)* to start another session to serve another request at the parent process's "?-" prompt.

```
serve(N) :-
        catch(serve_request(N),_,true).

serve_request(N) :-
        listen(port(X,N)),
        do_request(port(X,N)),
        close_connection(port(X,N)),
        halt.
```

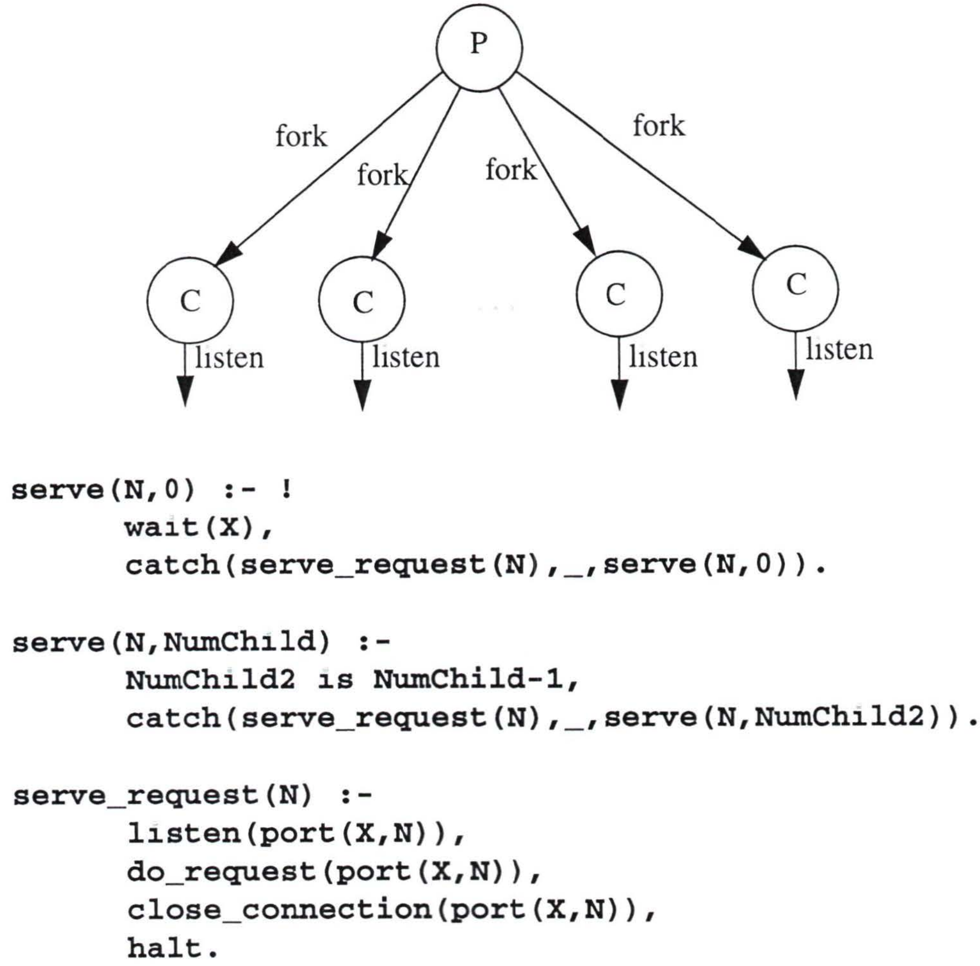
Figure 3.4 *catch* and *throw* used with *listen/1*

Can a reasonable set of concurrency models be achieved on UNIX systems by appropriately using the *catch* and *throw* (which is in *listen/1*) scheme? Figure 3.5 shows how to obtain N child processes simultaneously serving client's requests. Here the user specifies the number of connections that the program handles. After spawning the specified number of child processes, the parent process goes to wait for any child process to finish, then it spawns another child to keep the number of children unchanged. Unfortunately, there is no mechanism, using *catch* and *throw*, to detect the termination of a child process. Therefore we need to add one library routine:

**wait**(*?ProcessId*)

Waits for a child process to finish. *ProcessId* unifies with the process id of the

chıld process that fınıshes



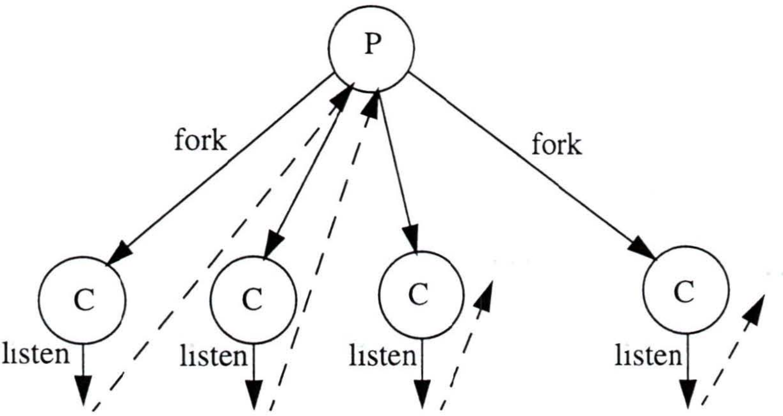

```
serve(N,0) :- !
      wait(X),
      catch(serve_request(N),_,serve(N,0)).

serve(N,NumChild) :-
      NumChild2 is NumChild-1,
      catch(serve_request(N),_,serve(N,NumChild2)).

serve_request(N) :-
      listen(port(X,N)),
      do_request(port(X,N)),
      close_connection(port(X,N)),
      halt.
```

Fıgure 3 5 Parent process forks N chıld processes sımultaneously

Now consıder the followıng model After the parent process forks a chıld process to serve a clıent, ıt blocks ıtself and waıts for the chıld process to fınısh before forkıng another process Thıs model ıs useful when exclusıve access to the resource ıs requıred It ıs shown ın Fıgure 3 6



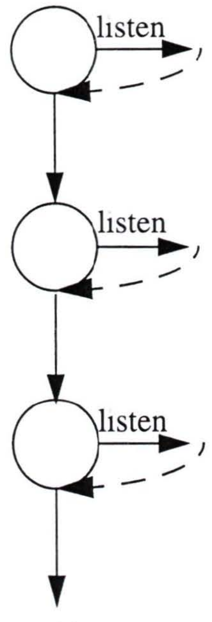

```
serve(N) :-
      catch(serve_request(N),Pid,parent(N,Pid)).

parent(N,Pid) :-
      wait(Pid),
      serve(N).

serve_request(N) :-
      listen(port(X,N)),
      do_request(port(X,N)),
      close_connecction(port(X,N)),
      halt.
```

Figure 3.6 Parent process waits for child process to finish before forking another process

An alternative approach to Figure 3.6 is Figure 3.7 Here, instead of having the parent waiting for a child to finish before spawning another one, the parent process forks a child process and then terminates itself After the child process finishes serving the client, it then forks another process and terminates itself The cycle goes on




```
serve(N) :-
        catch(serve_request(N),_,halt).

serve_request(N) :-
        listen(port(X,N)),
        do_request(port(X,N)),
        close_connection(port(X,N)),
        catch(serve_request(N),_,halt).
```

Figure 3 7 Child process creates another process before terminating itself

On a system that does not support multi-processing, such as on a Macintosh, *listen/1* is implemented without doing any forking or *throw* The calling process simply listens on a given port for connection, and upon connection, it serves the request from the connecting machine From the Prolog programmer's point of view, *listen/1* can still be used with a *catch*, except now the parent branch of the *catch* is never executed For example, the program in Figure 3 7 can be run on a Macintosh without any modification to achieve the same goal What is different now from it running on a UNIX machine is, here *halt* is never executed because *listen/1* does not do a *throw* anymore Portability is

achieved.

The Levy/Rintoul extensions to the DEC-10 I/O model introduced communication ability and concurrency into Prolog and therefore made writing distributed applications possible. Chapter 5 gives an example of an HTTP server implemented in Prolog running on a UNIX system.

# 3.5 Comparison with Related Work

## 3.5.1 I. C. Prolog II

I.C. Prolog II [3] (ICP for short) is an implementation of Prolog that is suited for distributed applications. This section compares the Levy/Rintoul approach with the ICP approach to show the similarities and differences between the two.

ICP contains primitives that support concurrency as well as three mechanisms for performing inter-process communication: pipes, TCP/IP sockets and mail-boxes. ICP's support for multiple threads is provided by a built-in primitive *fork/1* with the single argument being the predicate to execute in the new process. Each ICP process is a distinct WAM-like structure consisting of a STACK area and a set of WAM registers.

Communication between two threads that are running in the same ICP process can be done using pipes. A pipe is a uni-directional communication channel which has two ports, one for writing to and the other for reading from. Communication between different ICP processes (possibly on different machines) or between ICP and other processes must be done using TCP/IP primitives or mail-boxes. Figure 3.8 is an example of a server using TCP/IP primitives for communication.

```
concurrent_server(Port) :-
      tcp_server(Port, Socket),
      multi_serve(Socket).

multi_serve(Socket) :-
      tcp_accept(Socket, New),
      fork(service(New)),
      multi_serve(Socket).
```

Figure 3.8 An example server in ICP

In Figure 3.8, *tcp_server/2* is the primitive to create a socket and bind it to a port. Call to *tcp_accept/2* suspends until a connection request from a client is received whereupon a new socket will be created specifically for that connection. *fork/1* creates a new thread with the single argument being the goal to be executed in the new thread. *fork/1* always succeeds immediately regardless of the success or failure of the forked goal so as to allow concurrency. Once a connection is established, the writing and reading attempts are done by calling primitives *tcp_send(+Socket, +Message)* and *tcp_recv(+Socket, -Message)*.

Obviously, there are similarities between the Levy/Rintoul approach and that of ICP. Both implementations of communication primitives are based on TCP. ICP made its interface to TCP rather specific, while the Levy/Rintoul extensions were implemented as a simple extension to the DEC-10 I/O model. Both implementations support concurrency required for client-server applications. ICP's multi-threading capability is explicit through the use of the *fork/1* primitive. Levy/Rintoul's is implicit with the *fork* occurring at the time a connection from a client takes place. The reason for not having an explicit *fork* is because not every system supports it. It would seem unnecessary to have a set of primitives for a particular system such as UNIX. Instead, it would be more appropriate to hide the lower level detail and provide Prolog programmers with portable primitives that can be run on different platforms.

### 3.5.2 Delta-Prolog

Delta-Prolog [10] is a distributed logic programming language based on Monteiro's Distributed Logic which extends Horn Clause Logic in two ways (1) by distinguishing between sequential and parallel composition of goals and (2) by introducing the time related notion of event which provides both for process communication and synchronization It is a super-set of Prolog

Events are represented using the notion of "event goals" denoted by G ! E or G ? E, where ! and ? are binary predicate symbols, G is any term, and E is an atom A selectable goal G ! E may be reduced iff a complementary goal G' ? E may be selected such that G and G' are unifiable G ! E and G' ? E execute in separate processes, possibly on different computers G ! E blocks until the complementary G' ? E within a different process is executed Communication between the two processes occur when G and G' are unified The "?" and "!" operators provide a convenient and powerful way of synchronizing processes and communicating between processes

Figure 3 9 shows a simple example of a Delta-Prolog program which computes squares according to the formula $K^2 = (K-1)^2+(2K-1)$ for K>0

```
(process 1)

     squares :- write(0), nl, sq(0).
     sq(Q) :- I ? mail,
           R is Q+I, write(R), nl,
           sq(R).

(process 2)

     odds :- odd(1).
     odd(I) :- I ! mail,
           J is I+2, write(I), nl,
           odd(J).
```

Figure 3 9 Squares example in Delta-Prolog

Two processes, *squares* and *odds*, are launched simultaneously on two different terminals. When *squares* begins, it waits at the goal *I ? mail* in the *sq* predicate until *odds* executes the *I ! mail* goal in the *odd* predicate. When this happens, the *I* in the *sq* predicate is unified with 1 and the *squares* process continues with a recursive call to *sq/1*, where it again waits for *odds* to execute the *I ! mail* goal.

Delta-Prolog provides a way for inter-process communication. Yet there are important differences between it and TOPIC. Firstly, the TOPIC extensions are done on top of TCP/IP. Therefore it is possible for a TOPIC Prolog program to communicate with existing TCP/IP applications written in other programming languages. While in Delta-Prolog, communication has to take place between two Delta-Prolog programs. Secondly, the TOPIC extensions are done by enlarging the library of the Prolog system without touching the Prolog engine. In Delta-Prolog, the semantics of the Prolog language is modified to incorporate the extensions, consequently inventing a new language.

# Chapter 4

# Prolog and Databases

A database is often a key part of a server. In my experimental HTTP server, authorization information as well as client access history are maintained in databases. In this chapter, we explore the correlation between Prolog and databases. In the next chapter, we will see, through an example, how Prolog can be used to benefit client-server development in terms of managing small databases.

## 4.1 Relational Databases

A *data model* is an abstract representation of a collection of data about entities, their attributes, events, activities, and their associations. The purpose of a data model is twofold: first, to represent data, and second, to be understandable. If a data model accurately and completely represents required data and is understandable, then it can be used in applications. The *Relational Data Model*, the *Network* and the *Hierarchical* models are the three data models currently supported by most *Database Management Systems (DBMS)* [9]. We will only talk about the relational data model here because that is what Prolog is related to.

### 4.1.1 The Relational Model

In the relational data model, a *relation* is viewed as a two-dimensional table with rows for the entities and columns for the attributes. Two examples of relations are shown

in Table 4.1 and Table 4.2

| COURSE | YEAR | DEPARTMENT | INSTRUCTOR |
|---|---|---|---|
| Databases | 1992 | Computer Science | Brown |
| Data Structures | 1991 | Computer Science | Black |
| Geometry | 1991 | Mathematics | Ross |
| Chemistry | 1993 | Chemistry | Smith |
| Chemistry | 1994 | Chemistry | Smith |

Table 4.1 OFFERING relation

| STUDENT | COURSE | YEAR | GRADE |
|---|---|---|---|
| Davis | Geometry | 1991 | A |
| Fraiser | Databases | 1992 | B |
| Myers | Chemistry | 1993 | A |
| Jones | Chemistry | 1993 | C |
| Turner | Data Structures | 1991 | A |

Table 4.2 EXAM relation

Each row of a relation is called a *tuple*. The number of attribute values that a tuple has is called the *degree* of the relation. OFFERING is an example of a tuple with degree 4. The number of tuples in a relation is its *cardinality*.

A *domain* is the set of possible values for an attribute. For example, the domain for INSTRUCTOR in the OFFERING relation is a set of alphabetic character strings restricted to people's names. The domain for GRADE in the EXAM relation is A, B, C, D, F.

An attribute (or attributes in combination) for which no more than one tuple may have the same (combined) value is called a *key*. A key can be used to uniquely identify the tuples of the relation. It should be minimal: it should not contain attributes which are not strictly required for a unique identification of the tuples of a relation. One relation

can have multiple keys, but only one (or one combination) is chosen. In the above examples, we assume that each course is taught each year by a single professor; thus, the pair of attributes COURSE and YEAR is the key of OFFERING relation. Each professor can teach the same course over many years, and multiple courses in the same year; hence, pairs INSTRUCTOR, COURSE and INSTRUCTOR, YEAR are not keys. The pair STUDENT, COURSE is the key of the relation EXAM, since we assume that each student takes an exam in a specific course at most once.

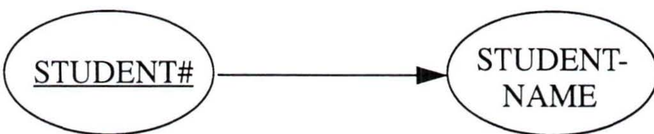
### 4.1.2 Relationships in the Relational Data Model

Relations can be abstractly represented using a shorthand notation. The two relations in Table 4.1. and Table 4.2. can be written in this notation as the following, with the key underlined.

OFFERING(<u>COURSE</u>, <u>YEAR</u>, DEPARTMENT, INSTRUCTOR)

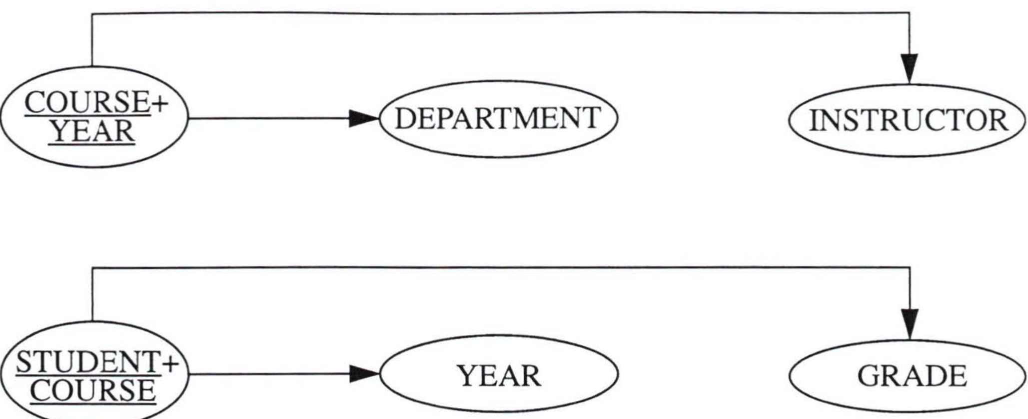
EXAM(<u>STUDENT</u>, <u>COURSE</u>, YEAR, GRADE)

The relationship between attributes is called *functional dependency*. Attribute B is functionally dependent on attribute A if at each point in time, each value of A has only one value of B associated with it. For example,




illustrates that STUDENT-NAME is dependent on STUDENT#.

The key in a relation uniquely identifies the tuple. Therefore it uniquely identifies each of the remaining attributes in the relation. The OFFERING and EXAM relations can be illustrated using such a dependency diagram (called a bubble diagram) as follows.




These bubble diagrams indicate that each of the OFFERING relation nonkey attributes depends only on the concatenated key of COURSE and YEAR, and the nonkey attributes of EXAM depend only on the concatenated key STUDENT and COURSE.

We summarize that a relation has the following properties:

1. Each column contains values about the same attribute, and each table cell value must be simple (a single value).
2. Each column has a distinct name (attribute name), and the order of columns is immaterial.
3. Each row is distinct; that is, one row cannot duplicate another row for selected key attribute columns.
4. The sequence of the rows is immaterial.
5. All nonkey attributes should be fully dependent on the whole key.
6. Each nonkey attribute should be dependent only on the relation's key, not on any other nonkey.

### 4.1.3 Data Processing Using Relational Languages

The basic component of the relational data model is a relation. The two classes of special-purpose relational languages that manipulate relations are *relational algebra* and

*relational calculus*. A significant feature of relational languages is that the result of any operation is also a relation. Thus, the result can then be further manipulated by the same operations as base relations.

Relational algebra manipulates one or two relations as operands and produces a new relation as the result. The operations of relational algebra are: selection, projection, join, Cartesian product, union and difference.

- Selection: This operation retrieves all tuples of a specified relation that satisfy a certain condition and constructs a new relation that contains the selected tuples. It constructs a new table by taking a horizontal subset of an existing table. That is, it selects those whole rows that satisfy a stated condition (Figure 4.1 (a)).

- Projection: This operation forms a vertical subset of an existing table by extracting specified columns (attributes) from all tuples to form a new table. The cardinality of the result can change due to projection, because duplicate tuples are removed (Figure 4.1 (b)).

- Join: The join operation combines the data from two relations based on values for common attributes. The result is also a relation (Figure 4.1 (c)).

- Cartesian product: The Cartesian product of two relations A and B is a new relation whose attributes are the concatenation of all attributes of A and B, and whose tuples are formed by all the possible concatenations of tuples of A and B (Figure 4.1 (d)).

- Union: This operation merges two tables that have corresponding columns with identical domains into one table with duplicate tuples eliminated (Figure 4.1 (e)).

- Difference This also applies to two tables with identical domains The difference of two tables, A and B, is a third table C, which contains the tuples that are in A but not in B (Figure 4 1 (f))

Initial relations

R

| A1 | A2 | A3 |
|---|---|---|
| a | b | c |
| f | d | h |
| f | e | h |

S

| B1 | B2 |
|---|---|
| d | e |
| g | h |
| f | m |

T

| B1 | B2 |
|---|---|
| b | e |
| d | e |
| g | m |

Operations

(a) SELECT R WHERE A1=f GIVING N

| A1 | A2 | A3 |
|---|---|---|
| f | d | h |
| f | e | h |

(b) PROJECT R OVER (A1,A3) GIVING N

| A1 | A3 |
|---|---|
| a | c |
| f | h |

(c) JOIN R OVER A2 AND S OVER B2 GIVING N

| A1 | A2 | A3 | B1 | B2 |
|---|---|---|---|---|
| f | e | h | d | e |

(d) PRODUCT R AND S GIVING N

| A1 | A2 | A3 | B1 | B2 |
|---|---|---|---|---|
| a | b | c | d | e |
| a | b | c | g | h |
| a | b | c | f | m |
| f | d | h | d | e |
| f | d | h | g | h |
| f | d | h | f | m |
| f | e | h | d | e |
| f | e | h | g | h |
| f | e | h | f | m |

(e) UNION S WITH T GIVING N

| B1 | B2 |
|---|---|
| d | e |
| g | h |
| f | m |
| b | e |
| g | m |

(f) SUBTRACT T FROM S GIVING N

| B1 | B2 |
|---|---|
| g | h |
| f | m |

Figure 4 1 Examples of algebraic operations

Relational calculus is the second major category of relational languages. It manipulates relations implicitly by specifying conditions that can involve attributes from several relations. Unlike relational algebra whose operations apply to whole relations, relational calculus can apply to attributes of relations. Commands in relational calculus systems specify in some syntax which attributes to manipulate, from what relations, and for what tuples. There are two fundamental differences between relational algebra and relational calculus: (1) calculus combines the SELECT and PROJECT commands and the binary operations into one RETRIEVE (or similar) statement that lists the attribute names to appear in the result and uses a WHERE clause to specify the selection criteria, and (2) calculus also uses the WHERE clause to specify the inter-relation associations used for implicitly JOINing relations in the RETRIEVE command. The JOIN operator of relational algebra is a binary operator. Thus, a table that is the combination of *n* relations must be generated in *n*-1 JOINs, whereas the RETRIEVE command can JOIN numerous tables implicitly at one time. For example, example (c) written in a relational calculus type of command would be

```
RETRIEVE (R.A1,R.A2,R.A3,S.B1,S.B2) INTO N
WHERE R.A2=S.B2
```

An example of a relational calculus language is SQL. The most typical statement of the language is *query block*. An SQL block has the form:

```
SELECT <attribute_list>
FROM <relations>
WHERE <condition>
```

Query blocks can also be nested by including entire SQL blocks within the WHERE clause. The following is a simple example of block nesting:

```
SELECT <attribute_list>
FROM <relations>
WHERE <attributes> =
     SELECT <attribute_list>
```

```
    FROM <relations>
    WHERE <condition>
```

In the examples of Table 4.1 and Table 4.2, if we want to find out all the students who received an A in 1991, the SQL query would be:

```
SELECT STUDENT
FROM EXAM
WHERE GRADE = A AND YEAR = 1991.
```

The answer to this query is a relation having two tuples:

| STUDENT |
|---|
| Davis |
| Turner |

If we want to know the names and grades of all the students who have taken an exam in one of the courses administered by Computer Science department, the SQL query would be:

```
SELECT STUDENT, GRADE
FROM EXAM
WHERE <COURSE, YEAR> =
     SELECT COURSE, YEAR
     FROM OFFERING
     WHERE DEPARTMENT = Computer Science
```

The answer to this query is:

| STUDENT | GRADE |
|---|---|
| Fraiser | B |
| Turner | A |

A *view* is a relation which is not stored explicitly in the database, but is defined through a query language expression. Once a view is defined, it can be used for data manipulation as if it were a normal database relation. However, a view is a virtual relation. It is not maintained as real data. Instead, it is constructed automatically by the DBMS as needed. The major purpose of a view is to simplify query commands.

## 4.2 Prolog and Database Systems

### 4.2.1 Relationships between Prolog and Relational Databases

Logic programming and database languages have evolved in parallel throughout the seventies. Prolog was invented as a simplification of more general theorem proving techniques to provide efficiency and programmability. Similarly, the relational data model was invented as a simplification of complex hierarchical and network models, to enable set-oriented nonprocedural data manipulation. Important studies on the relationships between logic programming and relational databases have been conducted since the late seventies. Studies show that logic programming and database systems have several features in common:

1. *Databases*. Logic programming systems manage small, single-user, main-memory databases, which consist of deduction rules and factual information, called *deductive databases*. Databases systems deal with large, shared, mass-memory data collections, and provide the technology to support efficient retrieval and reliable update of the persistent data.
2. *Queries*. A query denotes the process through which relevant information is extracted from the database. In logic programming, a query (or goal) is answered by building chains of deductions, which combine facts and rules, in order to prove or refute the validity of an initial statement. In database systems, a query (expressed through a special-purpose data manipulation language) is processed by determining the most efficient access path in memory to large data collections, in order to extract relevant information.
3. *Constraints*. Constraints specify correct conditions for databases. Constraint validation is the process through which the correctness of the database is preserved, by preventing incorrect data being stored in the database. In logic programming, constraints are expressed through general-purpose rules, which are activated whenever the database is modified. In database

systems, only a few constraints are typically expressed using the data definition language.

Logic programming offers a greater power for expressing queries and constraints as compared to that offered by data definition and manipulation languages of database systems. It also allows more sophisticated reasoning about the database content. On the other hand, logic programing systems do not provide the technology for managing large, shared, persistent, and reliable data collections. To be more specific, here are some of the features that Prolog has that essentially enrich the expressive power of classical query languages based on relational algebra or calculus:

- *Intensional database specification*: rules in Prolog provide powerful means of definition and evaluation of derived relations ("view" in database systems). The set of derived relations defined through Prolog rules is called the *intensional database (IDB)*.

- *Recursion*: predicate or rules in Prolog can be recursively defined. Consequently, the expressive power of Prolog as a query language is strictly greater than the expressive power of conventional query languages.

- *Incomplete knowledge*: it is possible to mention null values in the rules and facts of the program, by means of the anonymous variable "-".

- *Total order in the fact and rule base*: the prolog programmer has the possibility of ordering facts and rules in the most efficient way with respect to computation efficiency. The Prolog engine is order sensitive.

- *Inference control*: Prolog allows the programmer to enhance efficiency of programs by introducing some procedural features, such as *cut*.

- *Negative information*: Prolog permits negative information to be expressed by using *not*.

The first two features are typical to any logic programming languages, while the remaining ones are typical to Prolog. Table 4.3. shows the similar concepts between Prolog and databases.

| Database Concepts | Prolog Concepts |
|---|---|
| Relation | Predicate |
| Attribute | Predicate argument |
| Tuple | Fact |
| View | Rule |
| Query | Goal |
| Constraint | Goal (returning an expected truth value) |

Table 4.3. Correspondence between similar concepts in Prolog and databases

### 4.2.2 Prolog as a Query Language

Prolog lends itself naturally to the specification and creation of databases. In fact, it offers additional facilities to those found in classic database systems, in particular, the ability to make deductions. This is why a Prolog database is called a *deductive database*. Prolog offers a unified framework for representing data in a database, for formulating requests or questions, and for defining "views" and integrity constraints. The facts form the extensional component and the rules form the intensional component.

A deductive database consists of facts and rules. Rules are used for two purposes: to express additional knowledge, deductible from existing facts, and to control the knowledge and its evolution. In the latter case these rules are called *integrity constraints*, which will be explained later. Let us have a look at how Prolog can be used as a query language.

Consider a relational database with two relations:

PARENT(PARENT, CHILD) and PERSON(NAME, AGE, SEX).

Each tuple of the PARENT relation has two attributes which define a parent-child relationship; each tuple of PERSON relation defines the name, age and sex of a person. We

assume that each person has a different name. The content of the database is shown in Figure 4.2.

PARENT

| PARENT | CHILD |
|---|---|
| john | jeff |
| jeff | margaret |
| margaret | annie |
| john | anthony |
| anthony | bill |
| anthony | janet |
| mary | jeff |
| claire | bill |
| janet | paul |

PERSON

| NAME | AGE | SEX |
|---|---|---|
| paul | 7 | male |
| john | 78 | male |
| jeff | 55 | male |
| margaret | 32 | female |
| annie | 4 | female |
| anthony | 58 | male |
| bill | 24 | male |
| janet | 27 | female |
| mary | 75 | female |
| claire | 45 | female |

Figure 4.2 Relation PARENT and PERSON

In Prolog, the two relations can be represented by two predicates *parent* and *person*, each of which defines a set of facts:

```
parent(john,jeff).
parent(jeff,margaret).
parent(margaret,annie).
parent(john,anthony).
...
```

```
person(paul,7,male).
person(john,78,male).
person(jeff,55,male).
person(margaret,32,female).
...
```

After loading the facts into the Prolog system, we then can express simple queries. The query *Who are the children of John?* is expressed by the following Prolog goal:

```
?- parent(john,X).
```

The answer is *X={jeff, anthony}*. In fact, this is not the form of answer Prolog interpreter would give. The answer is given as follows: after executing the goal, the variable *X* is first set equal to *jeff*, if the user asks for more answers, then the variable *X* is set equal to *anthony*, if the user asks for more answers again, then the search fails, and the interpreter

gives *no*. Note that by asking the query in this way, Prolog returns the result one tuple at a time, instead of returning the set of all result tuples.

The query *Who are the parents of Jeff?* is expressed as follows:

```
?- parent(X,jeff).
```

In the same way as the Prolog interpreter searches answers for the above example, the set of all answers is: *X={john,mary}*.

We can also asks query like *Is john jeff's parent?* by expressing

```
?- parent(john,jeff).
```

The interpreter will answer *yes* because the tuple *<john,jeff>* is in the database. Otherwise, we would expect *no* as an answer.

Rules can be used to build an *Intensional Database (IDB)* from the *Extensional Database (EDB)*. The EDB simply consists of facts, in our example it includes the relations PARENT and PERSON. The IDB is built from the EDB by applying rules which define its content, rather than by explicitly storing its tuples. In the following, we build an IDB which includes the relations FATHER, MOTHER, GRANDPARENT, SIBLING, UNCLE, AUNT, ANCESTOR, and COUSIN. Intuitively, all these relationships among persons can be built from the two EDB relations PARENT and PERSON.

We defines the relations FATHER and MOTHER, by simply indicating that a father is a male parent and a mother is a female parent.

```
father(X,Y) :- person(X,_,male), parent(X,Y).
mother(X,Y) :- person(X,_,female), parent(X,Y).
```

As a result, we can deduce the IDB shown in Figure 4.3.

FATHER

| FATHER | CHILD |
|---|---|
| john | jeff |
| jeff | margaret |
| john | anthony |
| anthony | bill |
| anthony | janet |

MOTHER

| MOTHER | CHILD |
|---|---|
| margaret | annie |
| mary | jeff |
| claire | bill |
| janet | paul |

Figure 4.3 The IDB relations FATHER and MOTHER

The two rules FATHER and MOTHER above can be regarded as view definitions, i.e., the resulting tuples are not stored. They are built from the tuples of PARENT and PERSON by the Prolog system.

The IDB can be queried as well. For instance, we can query: *Who is the mother of Jeff?* as follows:

```
?- mother(X,jeff).
```

After executing the query, *X* is set to be *mary*. Similarly, we can define the IDB relations GRANDPARENT, SIBLING, UNCLE, and AUNT:

```
grandparent(X,Z) :- parent(X,Y), parent(Y,Z).
sibling(X,Y) :- parent(Z,Z), parent(Z,Y), not(X=Y).
uncle(X,Y) :-
     person(X,_,male), sibling(X,Z), parent(Z,Y).
aunt(X,Y) :-
     person(X,_,female), sibling(X,Z), parent(Z,Y).
```

Complex queries to the EDB and IDB can be formulated by building new rules which combine EDB and IDB predicates, and then presenting goals for those rules. More complex IDB relations can be built from recursive rules, i.e. rules whose head occurs in the rule body. Here are two examples:

```
ancestor(X,Y) :- parent(X,Y).
```

```
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).

cousin(X,Y) :-
      parent(X1,X), parent(Y1,Y), sibling(X1,Y1).
cousin(X,Y) :-
      parent(X1,X), parent(Y1,Y), cousin(X1,Y1).
```

Rules can also express integrity constraints. In the PARENT relation, one of the constraint is:

a) *SelfParent constraint: a person cannot be his (her) own parent.*

In Prolog, this can be written as:

```
incorrectdb :- parent(X,X).
```

This constraint formulation enables us to inquire about the correctness of the database. The Prolog query

```
?- incorrectdb.
```

gives the answer. If no individual X exists to satisfy the body of the rule, then the answer to the query would be *no*. If there is an X which satisfies the goal, which means the database is incorrect, the answer to the query would be *yes*.

More examples of constraints are:

b) *OneMother: Each person has just one mother.*
c) *PersonParent: Each parent is a person.*
d) *PersonChild: Each child is a person.*
e) *AcyclicAncestor: A person cannot be his (her) own ancestor.*

These constraints are written in Prolog as follows:

```
b)   incorrectdb :- mother(X,Z), mother(Y,Z), not(X=Y).
c)   incorrectdb :- parent(X,_), not(person(X,_,_)).
d)   incorrectdb :- parent(_,Y), not(person(Y,_,_)).
e)   incorrectdb :- ancestor(X,X).
```

Constraint evaluation can be used either to preserve the integrity of an initially

correct database, or to determine (and then eliminate) the sources of inconsistency

Let us consider the first kind of application of constraints, namely, preserving the integrity of a correct database. This is needed because the database is constantly updated. A transaction is committed if and only if it does not produce a final database state that violates any constraint. Efficient methods have been designed for testing the correctness of the final state of a transaction. These methods assume the database to be initially correct, and test integrity constraints on the part of the database that has been modified by the transaction.

The second kind of application of constraints is concerned with restoring a valid database state. To serve this purpose, we need to modify the above Prolog constraint formulation to generate more helpful answers such as

```
a)    incorrectdb(selfparent,[X]) :- parent(X,X).
b)    incorrectdb(onemother,[X,Y,Z]) :-
          mother(X,Z), mother(Y,Z), not(X=Y).
c)    incorrectdb(personparent,[X]) :-
          parent(X,_), not(person(X,_,_)).
d)    incorrectdb(personchild,[Y]) :-
          parent(_,Y), not(person(Y,_,_)).
e)    incorrecctdb(acyclicancestor,[X]) :- ancestor(X,X).
```

In this formulation, the head of the rule has two arguments. The first argument is the name of the constraint, and the second one is the list of variables that violate the constraint. This constraint formulation enables us to inquire about the causes of incorrectness of the database. For instance, Prolog goal:

```
?- incorrectdb(X,Y).
```

gives the answer *no* if there exists no constraint X has been violated, in which case, the database is correct. If instead, one such constraint exists, then variable X and Y are bound to the constraint name and the list of values which cause the invalidity. For example, the answer *X=personparent, Y=[karen]* reveals that *karen* who belongs to the rela-

tion PARENT does not belong to the relation PERSON. This should be fixed by adding a tuple for *karen* to the relation PERSON.

However, the answer to the above query might not be sufficient if we want to restore the correctness of an IDB. IDB relations are generally not stored explicitly; they are defined by rules, and their value depends on the EDB relations which appear in these rules. Thus, violation to IDB constraints such as b) and e) should be compensated by actions applied to the underlying EDB relations.

## 4.3 Integrating Prolog and Relational Databases

Although we have seen that Prolog can act as a powerful database language, it has its drawbacks when used in this context, such as *tuple-at-a-time processing* and *order sensitivity and procedurality*. Research has been conducted on how to integrate the functionalities of logic programming and database systems. One approach is to develop new languages such as *Datalog* [4] which eliminate the drawbacks of Prolog, yet still preserve the functionality of logic programming. Datalog is specifically designed for use as a database language. Another approach is the *coupling* approach [4]. In this approach, an interface is built between current available Prolog systems and database systems, which preserve their individuality. The interface provides the procedures required for bringing data from the mass-memory database into the main-memory logic programming execution environment.

# Chapter 5

# A Personalized HTTP Server

## 5.1 The W3 System

The World-Wide Web (W3) is a system that combines a stateless client-server protocol (HTTP) with a hypertext language (HTML). W3 has a body of software, and a set of protocols and conventions. It uses hypertext and multimedia techniques to make the Web easy for anyone to browse and contribute to.

### 5.1.1 The Software

W3 software consists of client applications and servers. When a client is launched, it displays an object, normally a document with text and images. Some of the phrases or images are highlighted. Clicking the mouse on the highlighted area (called an *anchor*) causes the client program to retrieve another object from any computer on the network which is running a server. The retrieved object is normally also in a hypertext format, so the process of navigation continues. A W3 server's main task is simply to deliver requested documents. In addition it may support remote execution of requests (CGI's) and secure data change. A W3 server responds to an incoming TCP connection and provides a service to the caller, namely the client. Examples of client programs are NCSA Mosaic for X-Windows, Netscape, and MacWeb. Examples of server programs are CERN server and NCSA HTTP servers.

### 5.1.2 The Address System

Universal Resource Locators[1] (URLs) are strings used as addresses of objects on the Web. The following is an example:

http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/Docs/d2-intro.html

URLs are *universal* in the sense that they encode members of the universal set of network addresses. The prefix *http* in the above example indicates the protocol to be used, and defines the interpretation of the rest of the string. When HTTP protocol is used, the string contains the internet address of the server machine to be contacted and a substring to be passed to the server. URL syntax also allows objects to be addressed using the other common network information protocols in use today (FTP, NNTP, Gopher, etc.), and will allow extension when new protocols are developed.

URLs are central to the W3 architecture. The fact that it is easy to address an object anywhere on the Internet is essential for the system to scale, and for the information space to be independent of the network and server topology.

### 5.1.3 The Protocol

HTTP - HyperText Transfer Protocol, is the protocol used in W3. Rather than a protocol for transferring hypertext data, it is a protocol for transferring information with the efficiency necessary for making hypertext jumps. HTTP is an Internet protocol. It is similar in its readable, text-based style to FTP and NNTP that have been used to transfer files and news on the Internet. However, unlike these protocols, HTTP is stateless. That is, it runs over a TCP connection that is held only for the duration of one operation. Because practical information systems require functionality like search, front-end update

1 There is another term URI (Universal Resource Identifier) which is the name for a generic W3 identifier. The URI specification simply defines the syntax for encoding arbitrary naming or addressing schemes, while the URL specification defines the encoding for specific access protocols.

and annotation, this protocol not only allows fast information retrieval from different remote locations, but also allows an open-ended set of methods to be used, such as creating a new object, text searching, and updating existing documents at the remote end. A transaction consists of the following steps:

1. Connection: The establishment of a connection by the client to the server. The TCP/IP port 80 is the default port. Other non-reserved ports may be specified in the URL.
2. Request: The sending of a request message to the server by the client.
3. Response: The sending of a response to the client by the server.
4. Close: The closing of the connection by either or both parties.

A HTTP request from the client starts with a method followed by the URI of the object to be fetched, followed by optional header fields and data. A HTTP response contains a status line followed by an optional header about the object (*metainformation*) and the object itself. The format of the header is an extension of the MIME (Multipurpose Internet Mail Extensions) format. This is to facilitate the integration of hypermedia mail, news, and information access. Unlike in email, transfer in binary and nonstandard but mutually agreed document formats is possible. Figure 5.1 shows an example of a request and a response. More detail about HTTP can be found in [14].

Request:

```
GET /foo/bar HTTP/1.0
Referer: http://home.mcom.com/home/welcome.html
User-Agent: Mozilla/0.9 Beta (Windows)
From: linjiang
Accept: *
Accept: image/gif
Accept: image/jpeg
Authorization: Basic TWljaGFlbCBMZXZ5OldoYXQgYSBDcm9jaw==
```

Response:

```
HTTP/1.0 200 Document follows
MIME-Version: 1.0
Server: TOPIC Experiment 1.0
Date: Wed, 29 Mar 95 01:37:20 GMT
Content-Type: text/html

<h1>Prolog Experimental Server</h1>
<strong>This is just a test</strong>
<p>She sells sea shell by the sea shore.<p>
```

Figure 5.1 An example of an HTTP request and response

### 5.1.4 The Language

HTML (HyerText Markup Language) is the common basic language used in W3 to interchange hypertext. It was designed sufficiently simple so as to be easily produced by both people and programs, while adhering to the SGML (Standard Generalized Markup Language) standard. It is a collection of styles (indicated by markup tags) that define the various components of a W3 document. It includes simple structure elements, such as several levels of headings, bulleted lists, menus and compact lists, all of which are useful when presenting choices, and in on-line documents.

HTML is defined to be a language of communication, which actually *flows* over the network. There is no requirement that files have to be stored in HTML format. Serv-

ers may store files in other formats and then generate HTML on the fly with each request.

# 5.2 A Personalized HTTP Server

I implemented a HTTP server (referred to as "the server") which is compatible with any TCP/IP based HTTP clients. The purpose of the server is not only to allow users to browse through homepages over the network like other HTTP servers do, but also to provide each user a personalized home page with information specific to that user. It is implemented in TOPIC Prolog which is hooked with some library routines written in C. It takes advantage of the conciseness of Prolog in handling cases and the simplicity in building small databases, and the efficiency of C in doing the lower level tasks.

## 5.2.1 Overview of the Server

All users of the server must be registered at the server side. The client information is stored in a Prolog database. When a user connects to the server from a HTTP client for the first time, the server asks the client to authenticate itself (Section 5.2.2). The client then prompts the user for a username and password. After the server has checked the authority of the client and approved it, it then sends out a home page for that user (Figure 5.2).

The home page basically contains two options for users to choose. A user can either choose NEWS to read news articles, or choose MAIL to read mail messages. If the user clicks on NEWS, the server will send over a list of newsgroups that the user has been reading and a dialog box which allows the user to subscribe a new newsgroup (Figure 5.3). By either clicking on an existing newsgroup or subscribing a new group, the user can get a list of articles that *haven't been read before* on the next page (Figure 5.4). The user can read an article by clicking on it. Next time when the user reads the same newsgroup, this article will not be shown in the list since the user has read it already.

If the user clicks on MAIL on the homepage, a new page will appear prompting

the user for username and password for a chosen mail server (Figure 5 5) After clicking on OK, if login succeeded, the user will get the next page which contains a list of new mail messages and a hypertext link to the old mail messages (Figure 5 6) The user can read a message by clicking on it The server creates a mailbox for each user Every time when a user reads his or her mail, the new mail messages are transferred to the HTTP server machine from the mail server, and are stored in the user's mailbox This allows a user to read mail messages from different mail servers, yet it stores the messages centrally

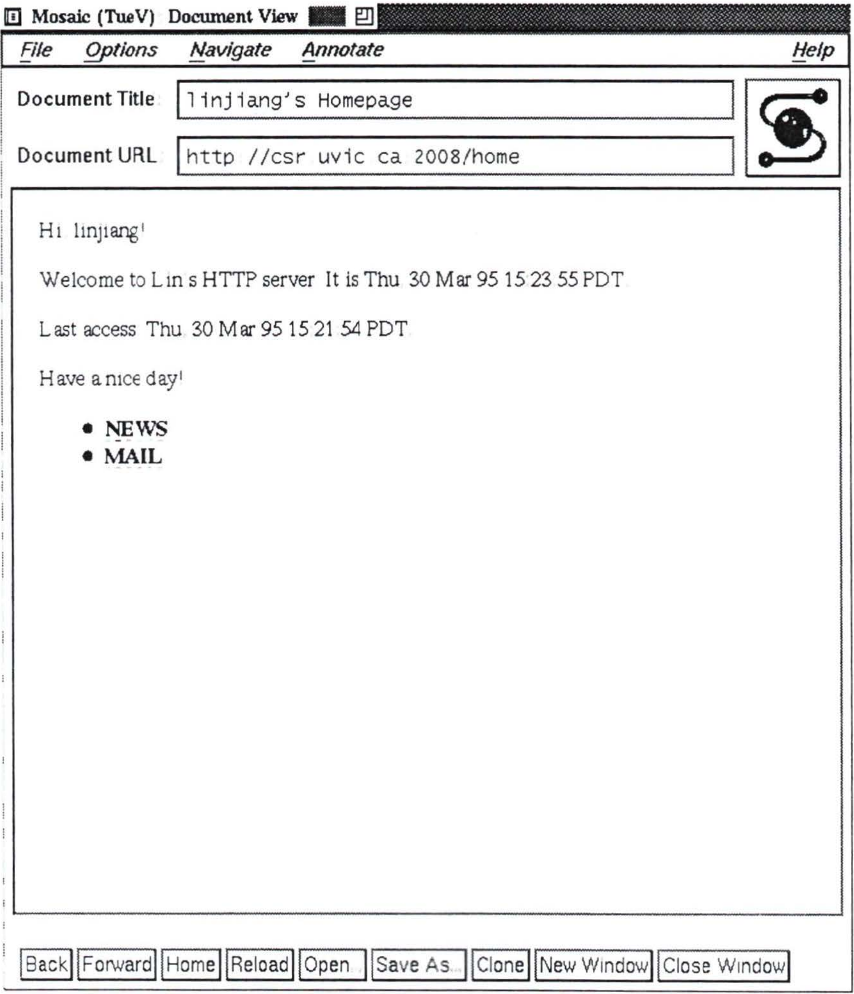



Figure 5 2 User's homepage

Mosaic (TueV): Document View

File Options Navigate Annotate Help

Document Title: News

Document URL: http://csr.uvic.ca:2008/news

**Newsgroups currently subscribed:**

- alt.evil
- alt.food
- alt.magic
- comp.lang.c
- comp.lang.prolog

---

**Subscribe a newsgroup:**

**Press this button when ready:** OK

Back | Forward | Home | Reload | Open | Save As | Clone | New Window | Close Window

Figure 5.3 User's page on NEWS

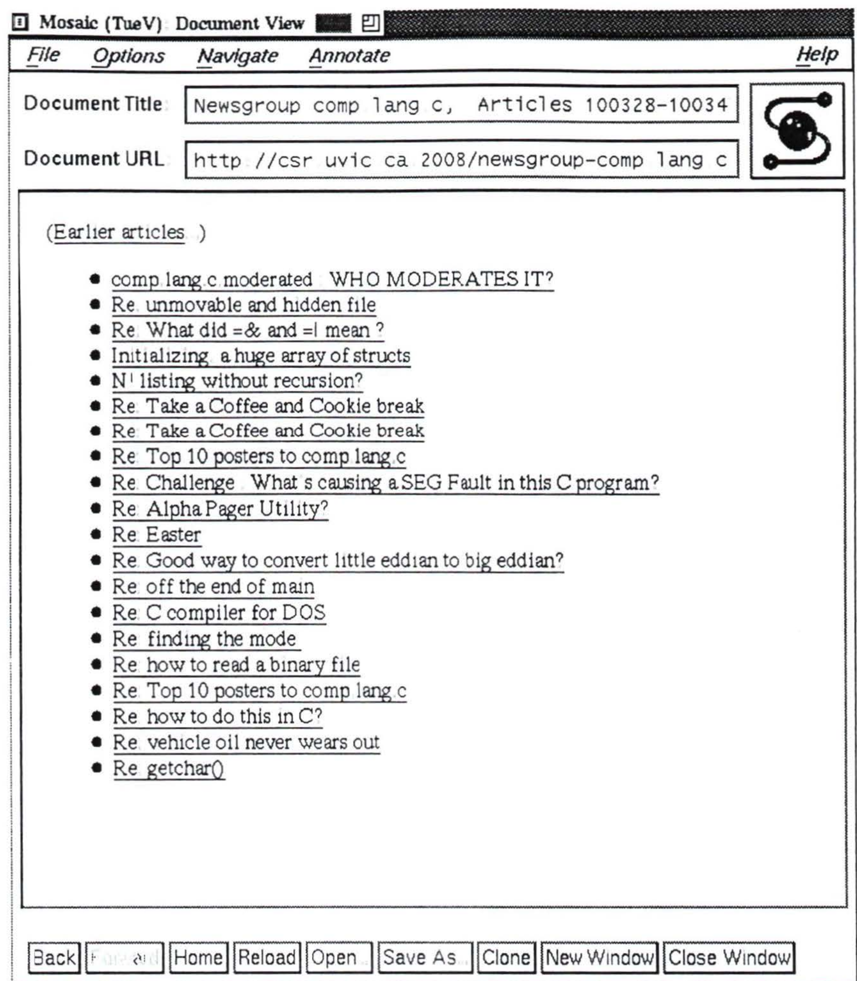



Figure 5 4 User's page on NEWS articles

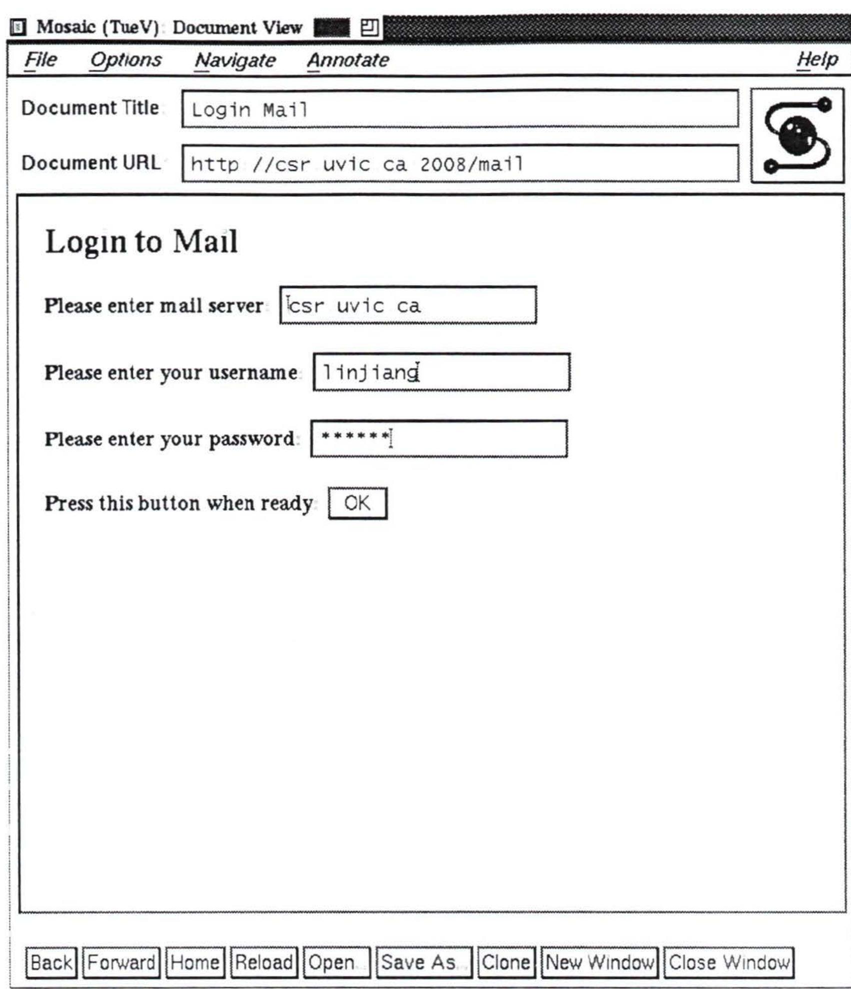



Figure 5 5 User's page on login to MAIL server

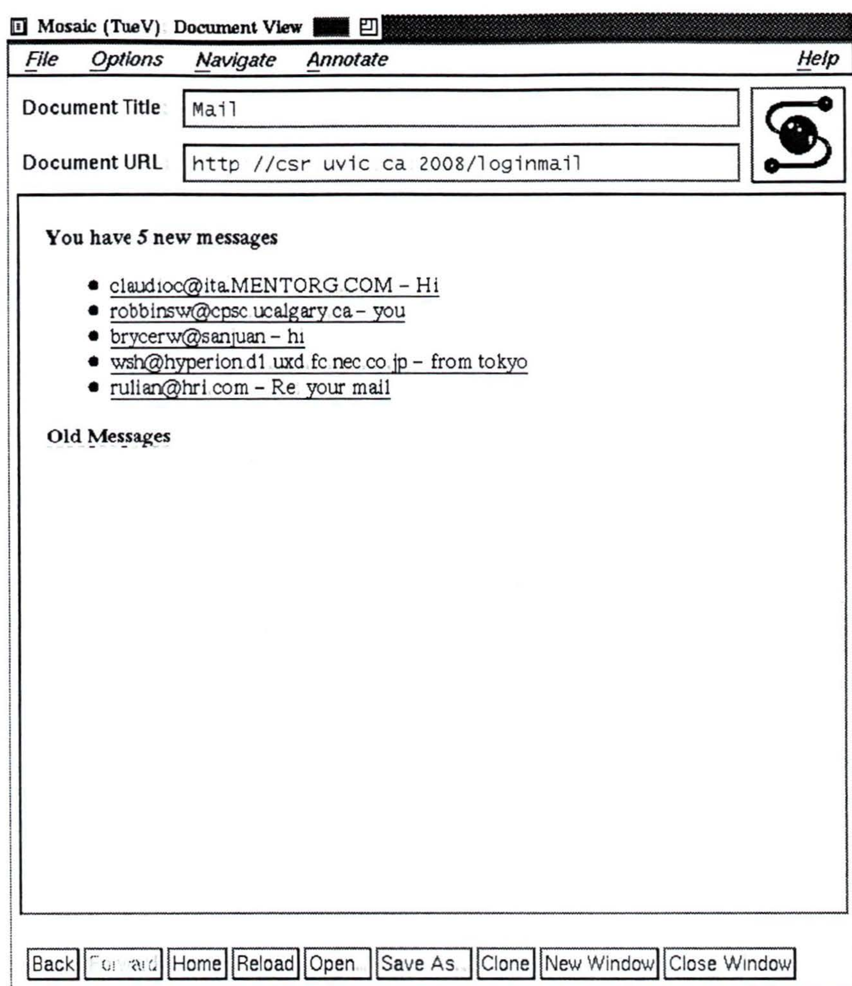



Figure 5 6 User's page on MAIL messages

## 5.2.2 Authentication

The protection scheme I implemented is the Basic Protection Scheme The authentication process happens when the user tries to connect to the server for the first time It consists of the following steps

1 Server sends an Unauthorized status

2 Client authenticates itself

3 Server checks authentication and authorization

4 If Step 3 is successful, document is sent normally by the server

#### 5.2.2.1 Server Sends an Unauthorized Status

In the server, all requests received from the client must contain the *Authorization* field. If the server receives a request that does not have this field, it sends an *Unauthorized 401* status code, and a set of *WWW-Authenticate* fields containing valid authentication schemes and their scheme-specific parameters. The reply from the server is as follows:

```
HTTP/1.0 401 Unauthorized -- authentication failed
WWW-Authenticate: Basic realm="Subscribers"
```

where realm specifies the username-password file. In Prolog, this is done by setting the current output channel to the HTTP client, and then writing the above content to the current output channel (Figure 5.7).

```
authenticate(Hc) :-
      tell(Hc),
      write('HTTP/1.0 401 Unauthorized'), nl,
      write('WWW-Authenticate:  Basic  realm="Subscrib-
            ers"'),nl,nl.
```

Figure 5.7 authenticate/1

#### 5.2.2.2 Client Authenticates Itself

After receiving the Unauthorized status code, the browser prompts for username and password, and constructs a string containing those two separated by a colon:

```
username:password
```

This string is then encoded into printable characters, and sent along with the next request in the Authorization field as follows:

```
Authorization: Basic encoded_string
```

The string *Basic* indicates that the scheme used is the Basic scheme. The encoding

is done as defined in RFC 1421 The encoding scheme has a symbol table which is 64 in size Each element of the table is a printable character Every 6 bits of the original string is encoded into one of these characters, using the value of the 6 bits as the index of the table The character '=' is used for padding

### 5.2.2.3 Server Checks Authentication and Authorization

When the server receives a full request with the *Authorization* field, it decodes the string in this field The decoding process is the reverse of the encoding process After retrieving the username and password fields, it compares them to the ones in the password file If they are the same as those in the file, the server goes on to handle the request Otherwise, it asks the client to authenticate itself again This process is easily implemented using Prolog

On the server machine, there is a password file which is in pure Prolog form It contains multiple definitions of the fact *password*, which has two arguments, username and password The following is an example

```
password(linjiang, apple).
password(hwang, orange).
...
```

The file is consulted by the server when it is started (Section 5 2 3) The authorization checking is done as in Figure 5 8

```
%check_identity(User,Pass,Whattoget,HTTPclientIPaddr)
check_identity(Usr,Pwd,What,Hc) :-
      password(Usr,Pwd), !,
      do_request(What,Hc).

check_identity(_,_,_,Hc) :- % password failed
      authenticate(Hc).
```

Figure 5 8 check_identity/4

### 5.2.3 Concurrency

Because the protocol is stateless, the server handles each request independently, and the TCP connection is only open for the duration of processing that one request. The concurrency model used in this server is one that satisfies these needs. Basically, the parent process spawns a child process upon a connection request received from the client. The child process is responsible for handling the request, and the parent process goes into sleep and waits for the child process to finish. After the child process finishes handling the request, it closes the connection and terminates itself, and control goes back to the parent process, which again listens to the same port for another connection.

In TOPIC, the predicate *listen/1* allows the parent process to listen on a port for connection, and upon connection, it spawns a child process to handle the request. The Prolog code is shown in Figure 5.9.

```
serve(N) :-
        Ns = port('news.uvic.ca',119),
        see(Ns)[1],
        nntp_read(_),
        consult(pswrd),
        catch(http_serve(N,Ns),Pid,parent(N,Ns,Pid)).

parent(Hc,Ns,Pid) :-
        wait(Pid),
        catch(http_serve(Hc,Ns),Pid1,
                parent(Hc,Ns,Pid1)).
```

Figure 5.9 Concurrency in the server

In Figure 5.9, N is the port number on which the serve will be listening. *Consult(pswrd)* is to consult the password file that is used for authentication (Section 5.2.2). The catch-and-throw mechanism was explained in Chapter 3. Predicate *http_serve/2* is

1. This is to connect to the local NNTP server. The NNTP connection is kept open throughout the program instead of being established at each request because it takes about 3 seconds to connect to the NNTP server.

the predicate that handles the request from the HTTP client, which will be explained in Section 5.2.5.

### 5.2.4 Communication

The server communicates with three parties: the HTTP client, the NNTP server and the POP server. In a sense, the server is also a client to the NEWS server and mail server. In some cases, it interprets the HTTP requests into news or mail requests and send them to the news server or mail server (Section 5.2.5). Therefore it understands the three different protocols. The low level communication is based on UNIX sockets. Basically, the server creates a socket and binds it to the party's Internet address. The socket created can then be used as a usual I/O channel. The details are of course encapsulated within the predicates *listen/1*, *see/1* and *tell/1*, using a port. The difference between *listen/1* and *see/1* is that *listen/1* does a passive wait on a port for connection from a remote source, while *see/1* either connects to a remote port that is waiting for a connection and establishes an input channel, or it brings back an existing channel as the current input channel if the connection is already established. The *tell/1* is equivalent to *see/1* for output. For example, to provide HTTP services to clients, the server listens on a port by calling *listen/1*:

```
listen(port(X, 80)),
...
```

Upon connection, *X* contains the IP address of the remote machine that issued the connection request.

To connect to the local NEWS server which provides services on port 119, do a

```
see(port('news.uvic.ca', 119)),
...
```

Now any reading attempts take input from this channel. If later on, writing to the channel is desired, we can do a

```
tell(port('news.uvic.ca', 119)),
...
```

to set the output channel. Now any writing attempts write things out to the NEWS server.

To connect to the local POP server which listens on port 109, do a

```
see(port('csr.uvic.ca', 109)),
...
```

The three protocols are all readable, text-based protocols, which means the content can be read or written from or to an established socket connection as plain text being read or written from or to a file.

### 5.2.4.1 Communicating with HTTP Clients

The server communicates with the clients using HTTP. In order to read the requests sent by the client and interpret them into a form that is easy for Prolog to process, the predicate *http_read/1* is implemented. It is written in C and is added to the library. Predicate *http_read/1* takes a variable as the parameter, and returns a list of fields as the result in this parameter upon finishing. Basically, it parses the HTTP request to generate Prolog terms to represent it. The following is an example of a HTTP request:

```
GET /test HTTP/1.0
User-Agent: Mozilla/0.9 beta (X11; SunOS 4.1.2 sun4m)
From: linjiang
Accept: *
Accept: image/gif
Accept: image/x-xbitmap
Accept: image/jpeg
```

The request is sent with a first line containing the method to be applied to the object requested, in this case, the method is *GET* and the object is the file *test*, and the protocol version in use, followed by further information encoded in the RFC822 header style. *User-Agent*, *From* and *Accept* are all header fields. *User-Agent* gives the software program used by the original client. This is for statistical purposes and the tracing of pro-

tocol violations, *From* gives the name of the requesting user This field may be used for login purposes and an insecure form of access protection, *Accept* field contains a representation scheme which will be accepted in the response to this request If this request is read in using *http_read(L)*, the result is

```
L=[
  get(test,'HTTP/1.0'),
  user_agent('Mozilla/0.9   beta   (X11;   SunOS   4.1.2
               sun4m)'),
  from(linjiang),
  accept('*'),
  accept('image/gif'),
  accept('image/x-xbitmap'),
  accept('image/jpeg')
  ]
```

The method and each header field are turned into Prolog structures which are put into a list

To send a response to the client, the server usually sends a header first, which contains information about the object to be sent (metainformation), followed by the content of the object The object can be plain text, hypertext, or data in multimedia form The object header contains optional fields that describe the nature of the object Example fields are *Content-Type*, which describes the type of the data to be sent, *Content-Length*, which describes the length of that data, and *Date*, which is the creation date of object For example, the following is a header for sending a page in hypertext format

```
HTTP/1.0 200 Document follows
MIME-Version: 1.0
Server: TOPIC Experiment 1.0
Date: Sat, 25 Feb 95 01:14:40 GMT
Content-Type: text/html
Content-Length: 1269
```

The status code *200* after the http version indicates that the request is fulfilled The body section is the object returned by the request, which is in *html* form indicated by

the *Content-Type* field The transfer is terminated either by client reading *Content-Length* bytes of data or by the close of connection if *Content-Length* field is not presented An example of a simple server that handles transmission of files is shown in Figure 5 10

In Figure 5 10, *file_stat*, *sys_time* and *dumpfile* are predicates implemented in C *File_stat* gets the statistics about a file, *sys_time* gets the current system time, and *dumpfile* dumps a file into the current output channel This example is not "fool-proof", but it serves the purpose of demonstrating how the communication works between the server and the HTTP client

```
serve(N) :-
      catch(http_serve(N),Pid,parent(N,Pid)).

parent(Hc,Pid) :-
      wait(Pid),
      catch(http_serve(Hc),Pid1,parent(Hc,Pid1)).

http_serve(Hc) :-
      listen(port(X,N)), % wait for connection
      http_read(L), % read in request
      process_request(L, port(X,N)),
      seen. % close connection

process_request([],P).
process_request(get(X,_)|L,P) :- !,
      file_stat(X,stat(Fsize,Ftime,Ftype)),
      sys_time('GMT',Systime),
      tell(P),
      write_header(Systime,Fsize,Ftime,Ftype),
      dumpfile(X),
      process_request(L,P).
process_request([_|L],P) :- process_request(L,P).

write_header(Systime,Fsize,Ftime,Ftype) :-
      write('HTTP/1.0 200 Document follows'), nl,
      write('MIME-Version: 1.0'),nl,
      write('Server: TOPIC Experiment 1.0'),nl,
      write('Date: '), write(Systime), nl,
      write('Content-Type: '), write(Ftype),nl,
      write('Content-Length: '),write(Fsize),nl,
      write('Last-Modified: '), write(Ftime),nl,nl.
```

Figure 5 10 Example server that handles file transmissions

#### 5 2 4 2 Communicating with NNTP Server

The server communicates with the NEWS server using Network News Transfer Protocol (NNTP) NNTP is a protocol for the distribution, inquiry, retrieval, and posting of news articles It is designed so that news articles are stored in a central database allowing a subscriber to select only those items he wishes to read Like those in HTTP, requests and

responses of NNTP are also composed of characters from the ASCII character set. Requests consist of a keyword, in some cases may be followed by a parameter. For example, to get information about a specific newsgroup is:

```
GROUP group-name
```

and to get an article by its article number is:

```
ARTICLE nnn
```

Each request line must be terminated by a CR-LF (Carriage Return - Line Feed) pair.

Responses consist of a status line, which begins with a three digit numeric code, and in some cases some text, which is terminated by a single line containing only a period (.). For example, the response for

```
GROUP comp.lang.prolog
```

is

```
211 42 9067 9774 comp.lang.prolog
```

*211* is the status code indicating success, *42* is the estimated number of articles in the group, *9067* is the first article number in the group, *9774* is the last article number in the group, and *comp.lang.prolog* is the name of the group. The response for

```
ARTICLE 9774
```

is

```
220    9774    <1995Feb23.044046.3974@cs.sfu.ca>    Article
retrieved; head and body follow.
```

followed by the content of the article.

To read the response sent by the NNTP server, I implemented the predicate *nntp_read/1* which reads in the status line and parses it into Prolog terms that are easy for the program to process. Predicate *nntp/1* takes a variable as a parameter and returns the

result in it. The result is a list of items extracted from the status line that are useful for the program. For example, the result for *nntp_read(L)* on the response of *GROUP comp.lang.prolog* is

```
L=[211,42,9067,9774,'comp.lang.prolog']
```

and the result for the response of *ARTICLE 9774* is

```
L=[220,9774,'<1995Feb23.044046.3974@cs.sfu.ca>'].
```

#### 5.2.4.3 Communicating with POP Server

The server communicates with the mail server using Post Office Protocol (POP), which is intended to permit a workstation to dynamically access a maildrop on a server host in a useful fashion. Usually, this means that POP is used to allow a workstation to retrieve mail that the server is holding for it. Like NNTP, POP requests also consist of a keyword possibly followed by an argument. All commands are terminated by a CR-LF pair. For example, after logging in to the POP server, a user can retrieve a message by issuing the command:

```
RETR message-number.
```

To get the number of messages in the maildrop and the total size in bytes, use the command *STAT*.

Response sent back by the server either start with *+OK* upon success, or *-ERR* upon error, followed by more information. Possible responses to a *RETR* command are

```
+OK message follows
-ERR no such message
```

If successful, the body of the message is sent right after the *+OK* line, terminated by a period (.) on a line of its own. An example of a positive response to *STAT* is

```
+OK 2 320
```

To parse the responses to generate useful information for the Prolog program, *pop_read/1* is implemented. Because the responses do not start with status codes as NNTP responses do and neither do they follow any patterns, *pop_read/1* simply parses the first line of the response into words and puts them into a list which is returned in the argument. It is the application program's responsibility to retrieve the information it needs from the list, knowing the position of the item in the list. For example, the above responses to *RETR* and *STAT* are read in and parsed into *['+OK', message, follows]* and *['+OK', '2', '320']* by calling *pop_read/1*, respectively. Note that the numbers are strings instead of integers. Conversion from string to integer is necessary if arithmetic is to be operated on these numbers.

## 5.2.5 Handling Requests from Clients

Because HTTP is stateless, requests from the client are independent of each other. Therefore, the server serves each request independently without knowledge of previous transactions. Connection is established for each request and is closed after the response. The top-level code for handling each request is shown in Figure 5.11.

```
http_serve(N,Ns) :-
      Hc = port(X,N),
      listen(Hc),      % wait for connection
      http_read(L),
      extract_method(L,Coms), % extract method and
         % Authorization field (if presented)
      process_request(Coms,Hc,Ns),
      close_connection(Hc,Ns),
      halt.
```

Figure 5.11 http_serve/2

*N* is the port number on which the server listens. *Ns* is the channel to the news server which is already open (Section 5.2.3). Predicate *extract_method/2* extracts the method with its parameter and the *Authorization* field (if presented) from the list returned by

*http_read/1*. Predicate *process_request* processes the request based on whether the *Authorization* field is presented or not. It is shown in Figure 5.12. *check_identity* without the last parameter was shown in Figure 5.8.

```
% Contains only the method
process_request([What],Hc,Ns) :-
      authenticate(Hc).

% Contains method and Authorization field
process_request([What,Auth],Hc,Ns) :-
      decode(Auth,Decoded),
      strtok(Decoded,":",Usr,Pwd),
      check_identity(Usr,Pwd,What,Hc,Ns).
```

Figure 5.12 process_request/3

#### 5.2.5.1 The Tagging Technique

When clicking on a hypertext link at the client side, the name of the object that is attached to the method sent by the client is the anchor of that link defined by html. For example, the html for defining the anchor for NEWS in Figure 5.2 is

```
<a href="news"><strong>NEWS</strong></a>
```

Clicking on NEWS results in the following request being sent to the server:

```
GET /news HTTP1.0
```

Therefore, a simple way to determine what request the client sends is by tagging the *href* field of the anchor with a keyword followed by any useful information that the server may need to process the request. For example, the anchor for a newsgroup contains the keyword *newsgroup* followed by a hyphen (-), followed by the name of the group. The anchor for each news article contains the keyword *newsarticle*, followed by a hyphen (-), followed by the message-id of the article. Therefore, by parsing the object name sent with the *GET* method, the server can obtain the keyword and any useful information fol-

lowing it and handle the request accordingly. The Prolog code is shown in Figure 5.13. The conciseness of Prolog makes case handling easy and intuitive.

```
do_request(What,Usr,Hc,Ns) :-
        strtok(What,"-:?",Key,Name),
        get_item(Key,Name,Usr,Hc,Ns).

get_item(home,_,From,Hc,Ns) :- !,
...

get_item(news,_,From,Hc,Ns) :- !,
...

get_item(gotonewsgroup,Group,From,Hc,Ns) :- !,
...

get_item(newsgroup,Group,From,Hc,Ns) :- !,
...

get_item(newsarticle,Article,From,Hc,Ns) :- !,
...

get_item(articles,Range,From,Hc,Ns) :- !,
...

get_item(mail,_,From,Hc,Ns) :- !,
...

get_item(loginmail,SvrUsrPwd,From,Hc,Ns) :- !,
...

get_item(mailmsg,Msgnum,From,Hc,Ns) :- !,
...

get_item(oldmail,Msgnum,From,Hc,Ns) :- !,
...
```

Figure 5.13 Handling requests

### 5.2.5.2 Prolog CGI

An alternative to *do_request* in Figure 5.13 in handling requests is to use the idea

behind CGI (Common Gateway Interface) CGI is an interface for running external programs, or gateways, under an information server Gateways are programs which handle information requests and return the appropriate document or generate a document on the fly With CGI, a server can serve information which is not in a form readable by the client, and act as a gateway between the two to produce something which clients can use Gateways can be used for a variety of purposes, the most common one being the handling of FORM requests for HTTP When the server receives a request, it executes a program to which it passes the information from the request as the parameters This idea is easily implemented in Prolog by dynamically consulting a program and then calling a predicate defined in the program passing it useful parameters Figure 5 14 demonstrates how Figure 5 13 is done by using this idea The program that handles a specific request has the keyword as its file name, and each program has the predicate *get_item* defined in it

```
do_request(What,Usr,Hc,Ns) :-
        strtok(What,"-:?",Key,Name),
        consult(Key),
        get_item(Name,Usr,Hc,Ns). % defined in file "Key"
```

Figure 5 14 Prolog CGI

### 5 2.5 3 Personal Databases

One of the nice features about the server is it remembers what each user does by building a small database for each user The server keeps two files for each user, one for mail, in mailbox format, and one for news, in Prolog format The mail database file contains the user's mail messages, and the news database file contains the news groups and news articles the user has read Figure 5 15 gives an example of the news database file

```
group('alt.magic').
group('alt.food').
group('alt.evil').
group('comp.lang.prolog').
article('3eo251$1m9@darkstar.UCSC.EDU').
article('davew-0801951501180001@147.197.201.110').
article('3f754k$efg@taz.ramp.com').
article('prolog/faq-1-790250403@cs.sfu.ca').
article('3fd0ka$r1v@newsbf02.news.aol.com').
article('3fsh21$jrq@ixnews3.ix.netcom.com').
article('3fvgnq$pv9@crl7.crl.com').
...
```

Figure 5.15 An example of the user database file

This file is built by adding a fact each time the user subscribes a news group or reads a news article. The server obtains knowledge about a user simply by consulting the file in that user's name. The following code shows how *group/1* is used to generate the page shown in Figure 5.3.

```
get_item(news,_,From,Hc,Ns) :- !,
      tell(user_output),
      load_database(From), % consult the database file
      setof(X,group(X),L),
      tell(Hc),
      write_header('text/html'),
      write('<strong>Newsgroups    currently    sub-
            scribed:</strong><p>'),
      write('<menu>'),
      put_group(L),
      write('</menu><p>'),
      dumpfile('newsform.html').

put_group([]).
put_group([Name|L]) :-
      write('<li><a href="newsgroup-'),
      write(Name), write('">'),
      write(Name), write('</a>'),
      put_group(L).
```

Figure 5.16 Code that generates the page in Figure 5.3

The predicate *article/1* is used as a condition when listing the titles of news articles, as shown in Figure 5.17. Here, we are only interested in the articles that the user has not read.

```
print_if_unread(Id,_,_,_) :- article(Id), !.

print_if_unread(Id,Title,Hc,Ns) :- % article failed
      tell(Hc),
      write('<li><a href="newsarticle-'),
      write(Id),write('">'),
      write(Title),write('</a>'),
      tell(Ns).
```

Figure 5.17 Printing the title of an article

### 5.2.6 Related Work

There are many existing W3 servers, written in different languages, and for different platforms. But none of them has the feature of "personalizing" things. Examples of general purposed servers are:

- CERN server: A full featured hypertext server, with access authorization, research tools, etc. which can be used as a regular HTTP server. This server is also used as a basis for many other types of servers and gateways. Platforms: unix, VMS.
- NCSA server: A HTTP/1.0 compatible server for making hypertext and other documents available to Web browsers. Platform: unix.
- MacHTTP: A server for Macs participating in W3. It allows you to serve hypertext documents to other W3 users from your MacIntosh. Platform: MacIntosh.

# Chapter 6

# Conclusion

In this thesis, we have investigated the features of Prolog that would benefit the development of client-server applications. Some of the features of Prolog that made it a good language for Artificial Intelligence could also apply to the development of intelligent client-server software. Prolog is also good for building deductive databases. To test the hypothesis that Prolog is suitable for writing client-server applications, I implemented an HTTP server in the TOPIC system.

The TOPIC system is a WAM-based Prolog interpreter enriched with interprocess communication and multi-tasking abilities. With these essential features added in, the HTTP server I implemented is able to provide users with news and email services. The feature that made this server different from other existing ones is it provides personalized information to each user. It takes advantage of Prolog for building small databases and builds databases for users on the server's machine. It keeps track of users' activities on reading news and allows centralized email message storing.

This server is an experiment to see how good Prolog is in writing TCP/IP servers. The results are very promising. Firstly, the execution speed is competitive with existing HTTP servers on related transactions. For example, there is no perceptive time difference in getting a page of article titles in a newsgroup in this server and the standard NCSA server, despite the fact that there are additional computations going on in this server. Secondly, it again proved that Prolog is a very succinct and concise programming language

A good example is the handling of authentication. Thirdly, concurrency was easily expressed in code that is portable with no modification to non-UNIX platforms. Fourthly, the built-in database facility of Prolog was useful for a variety of "knowledge" representation tasks, such as maintaining personal information, newsgroup information, and so on. One drawback of TOPIC is the lack of buffered reads and pattern matching. Therefore buffering and regular expression facilities can be added in in future work. They will make it easier to read and parse protocol messages and HTML files.

My experiment can be seen as the first step towards an intelligent client/server software. I have shown that Prolog can easily be extended to support applications that use the TCP/IP suite protocols. The next logical step is to use Prolog to design intelligent modules for typical applications. As an example, it would be worthwhile to use a natural language recognizer to tackle the problems of mail prioritizing and news article filtering. Because TCP/IP is incorporated directly into TOPIC Prolog, using intelligent Prolog applications will be straight forward.

Although various commercial Prolog systems can be called from C or C++ programs, there are various obstacles to doing this. These include lack of portability, lack of a standard foreign language interface, and the difficulty of transferring data structures between the two languages. No such difficulties occur with the use of the TOPIC extensions.

# Glossary

**CGI (Common Gateway Interface)**

An interface for running external programs, or gateways, under an information server.

**DCG (Definite Clause Grammar)**

A special grammar rule notation in Prolog.

**EDB (Extensional DataBase)**

A Prolog database that consists of facts.

**HTML (HyperText Markup Language)**

The language used to create document in the World Wide Web system.

**HTTP (HyperText Transfer Protocol)**

The protocol used in the World Wide Web system. It is a protocol for transferring information with the efficiency necessary for making hypertext jumps. It is an Internet protocol and it is stateless.

**IDB (Intensional DataBase)**

A Prolog database that consists of rules.

**MIME (Multipurpose Internet Mail Extensions)**

Defines mechanisms for specifying and describing the format of Internet message bodies.

**NNTP (Network News Transfer Protocol)**

An Internet protocol for the distribution, inquiry, retrieval, and posting of NEWS articles.

**POP (Post Office Protocol)**

An Internet protocol for accessing maildrops.

**URL (Universal Resource Locator)**

The address system used in the World Wide Web system. It defines the encoding for specific access protocols.

# Bibliography

[1] T Berners-Lee, R Cailliau, A Luotonen, H F Nielsen, A Secret, *The World-Wide Web*, Communications of the ACM, pp 76-82, Vol 37, No 8, August 1994

[2] I Bratko, Prolog - *Programming for Artificial Intelligence, Second Edition*, Addison Wesley, 1990

[3] D Chu, K Clark, *I C Prolog II a Multi-threaded Prolog System*, Technical Report, Imperial College of Science, Technology and Medicine, May 1993

[4] S Ceri, G Gottlob, L Tanca, *Logic Programming and Databases*, Surveys in Computer Science, Springer-Verlag, 1990

[5] D Crookes, *Introduction to Programming in Prolog*, Prentice Hall, 1988

[6] D E Comer, D L Stevens, *Internetworking with TCP/IP, Vol III*, Prentice Hall, 1994

[7] B Kantor, P Lapsley, *Network News Transfer Protocol, RFC977*, 1986

[8] P Maes, *Agents that Reduce Work and Information Overload*, Communications of the ACM, pp31-40, Vol 37, No 7, July, 1994

[9] F R McFadden, J A Hoffer, *Data Base Management*, Benjamin/Cummings, 1988

[10] L M Pereira, R Nasr, *Delta-Prolog A Distributed Logic Programming Language*, Proceedings of the International conference on Fifth Generation Computer Systems, 1984

[11] K Rintoul, *Remote Predicates for Prolog A Basis for Declarative Client-Server Applications*, M Sc Thesis, University of Victoria, 1994

[12] P Saint-Dizier, *An Introduction to Programming in Prolog*, Springer-Verlag, 1990

[13] A Sinha, *Client-Server Computing*, Communications of the ACM, pp 77-98, Vol 35, No 7, July 1992

[14] *HTTP A protocol for networked information*, http //info cern ch/hypertext/WWW/Protocols/HTTP/HTTP2 html

[15] *A Beginner's Guide to HTML*, http://www.ncsa.uiuc.edu/demoweb/html-primer.html

# VITA

Surname: Jiang Given Name: Lin

Place of Birth: Beijing, China Date of Birth: January 28, 1969

Educational Institutions Attended:

| | |
|---|---|
| Beijing Computer Institute | 1987 - 1990 |
| Brandon University | 1991 - 1993 |
| University of Victoria | 1993 - 1995 |

Degrees Awarded:

| | | |
|---|---|---|
| B.Sc., Greatest Distinction | Brandon University | 1993 |

Honours and Awards:

| | |
|---|---|
| Brandon University | Faculty Association Scholarship |
| | Arthur and Abbie Vining Memorial Graduate Scholarship |
| | Silver Medal in Computer Science |
| University of Victoria | UVic Fellowship |
| | ASI Graduate Scholarship |

# PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me It is understood that copying or publication of this thesis for financial gain shall not be allow without my written permission

**Using Prolog for Internet Protocol Applications**

Author ____________

Lin Jiang

Date June 5, 1995