XML-BASED AGENT SCRIPTS AND INFERENCE MECHANISMS

Guili Sun, B.A., M.A.

Thesis Prepared for the Degree of

MASTER OF SC IENCE

UNIVERSITY OF NORTH TEXAS

August 2003

APPROVED:

Paul Tarau, Major Professor
Rada Mihalcea, Committee Member
Karl Steiner, Committee Member
Krishna M. Kavi, Chair of the Department of
Computer Science
C. Neal Tate, Dean of the Robert B. Toulouse
School of Graduate Studies

Sun, Guili, XML-Based Agent Scripts and Inference Mechanisms. Master of Science (Computer Science), August 2003, 49 pages, 4 figures, 29 references, 27 titles.

Natural language understanding has been a persistent challenge to researchers in various computer science fields, in a number of applications ranging from user support systems to entertainment and online teaching. A long term goal of the Artificial Intelligence field is to implement mechanisms that enable computers to emulate human dialogue. The recently developed ALICEbots, virtual agents with underlying AIML scripts, by A.L.I.C.E. foundation, use AIML scripts - a subset of XML - as the underlying pattern database for question answering. Their goal is to enable pattern-based, stimulus-response knowledge content to be served, received and processed over the Web, or offline, in the manner similar to HTML and XML. In this thesis, we describe a system that converts the AIML scripts to Prolog clauses and reuses them as part of a knowledge processor. The inference mechanism developed in this thesis is able to successfully match the input pattern with our clauses database even if words are missing. We also emulate the pattern deduction algorithm of the original logic deduction mechanism. Our rules, compatible with Semantic Web standards, bring structure to the meaningful content of Web pages and support interactive content retrieval using natural language.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Paul Tarau, for his immense help since the very beginning of my thesis work. His inspiration, encouragement, and carefulness helped me at every step of my process. I would also like to thank Dr. Rada Mihalcea for her comments on my thesis. Dr. Karl Steiner provided valuable feedback as well. The Natural Language Processing (NLP) group also gave me tremendous input. Last but not least, I would like to thank my husband for his support.

CONTENTS

ACKNOWLEDGEMENTS ii

1 Introduction 1

1.1 Chat Agents and Agent Programming 2

1.2 JINNI 3

1.3 XML 4

1.4 Prolog 5

2 Literature Review 7

2.1 ELIZA 7

2.2 ALICEbot 9

2.3 Semantic Web 11

2.4 Open Mind Project 13

2.5 VISTA Project 15

2.6 AIML 16

2.6.1 General Overview 16

2.6.2 Program D 18

2.6.3 AIML Pattern Mapping 19

2.6.4 Logical Deduction in AIML 20

3 Implementation 25

3.1 System Architecture 25

3.2 Database Generation 26
3.2.1 AIML Parser 26
3.2.2 Structure of Prolog Clauses 29
3.2.3 Lossless Information Conversion 35
3.3 Prolog Database Creation 36
3.4 Information Deduction 40
3.5 Running Scripts 43
3.6 Application with Agents 44

4 Conclusion 45

BIBLIOGRAPHY 47

LIST OF FIGURES

3.1 System Architecture 26

3.2 Database Generator 27

3.3 Tree Representation of Prolog clause of AIML scripts 30

3.4 Runtime System 40

CHAPTER 1

Introduction

Ever since computers were invented, natural language understanding has been a persistent challenge to researchers in various computer science fields, in a number of applications ranging from user support systems to entertainment and online teaching. A long term goal of Artificial Intelligence (AI) field is to implement mechanisms to enable computers to emulate human dialogue. From an early version of human-computer interaction in the 1960s, called Eliza[1], who was able to carry on a conversation with the end-user through string substitution and keyword-based canned responses, to recently developed ALICEbots[2], virtual agents with underlying AIML scripts, by A.L.I.C.E. foundations, we have seen dramatic improvements in AI chat bots. Claire[3], the virtual service representative of SprintPCS, has facial expression and can listen and respond to human speech audibly. Ultimately, we would like to have the perfect virtual agents[4] to perform various tasks for us in our daily life, as in the movie "Artificial Intelligence". The goal is clearly set, but we have a long way to reach it. In this thesis, we focus on improving the searching capability of chat agents, who have underlying XML compliant AIML query database, by converting the AIML scripts to equivalent but faster Prolog terms.

1.1 Chat Agents and Agent Programming

A computer agent is basically a program that can collect relevant information, and process certain tasks in the background. In particular, agent-based programming has become a dominant and promising paradigm which may help realize AI through distributed problem solving [5].

A chat agent is also called a chat bot, or simply bot. It is a computer program that simulates human conversation, or chat, through artificial intelligence. It is a software tool for digging through data. Typically a chat bot will communicate with a real person. It has wide applications, for instance, a welcome host in an online tour, an e-commerce service representative, a customer service representative to handle regular situation. Using chat agents could enhance web applications with a visible interactive personality, with human-like interaction between users and the system. The ability of making a decision on knowledge base has great development in entertainment and in distributed education.

Agents enable software developers to incorporate a new form of user interaction, called conversational interfaces, and develop conversational agents. This leverages natural aspects of human social communications. In addition to input from keyboard and mouse, newly developed agents can also recognize sound patterns and perform a certain task or respond back using synthesized speech, recorded audio or text. The conversational interface approach added special features to the already existing type chatting approach using graphical user interfaces. It brings computer to the next level of being more human like.

Agents, as digging tools, have great potential in data mining field. Data mining requires a series of searches, and agents can save labor as they can perform search diligently, and even better, consolidate the result in more human readable formats. They could have storing abilities, and speed up later searches. When AI will be advanced enough, agents will be able to make decisions using the vast knowledge base they collected. Typically, agent technology has applications in online teaching, accepting voice commands and accomplishing specific tasks, walking through a decision process, story narration and querying information to build virtual environments. The most important merit of agents is they do not feel tired of their job and never easily get emotional.

1.2 JINNI

JINNI[6], short for Java INference engine and Networked Interactor, is a lightweight, multi-threaded, logic programming language. It is designed to be used as a flexible scripting tool for gluing together knowledge processing components and Java objects in distributed applications. JINNI threads are coordinated through blackboards that are local to each process[7]. The synchronization mechanisms between local and remote thread are built on top of a Linda-style coordination framework. The associative search is implemented through unification based pattern matching. Threads are controlled with tiny interpreters following a scripting language based on a subset of Prolog. This particular feature makes JINNI a convenient development platform for distributed AI, and in particular, for building intelligent autonomous

agent applications.

1.3 XML

Thanks to the World Wide Web, the information exchange these days is much quicker, easier and flexible. To make electronic document distribution faster and easier to an international audience, to address the requirements of commercial Web publishing, and to enable the further expansion of Web technology into new domains of distributed document processing, the World Wide Web Consortium has developed an Extensible Markup Language (XML)[8] for applications that require functionality beyond the current Hypertext Markup Language (HTML). The flexibility of XML lies in that one can define his own tags, and use a Document Type Definition (DTD) or an XML Schema to describe the data. With a DTD or XML Schema, XML is designed to be self-descriptive.

There are certain applications that cannot be accomplished by HTML, but an be achieved with XML. These applications can be divided into four broad categories [9]:

- Communication between two or more heterogeneous databases
- Distribution of a significant proportion of the processing load from the Web server to the Web client
- Presenting different views of the same data to different users
- Possibility of the intelligent Web agents attempting to tailor information discovery to the needs of individual users.

XML derives from a philosophy stating that data belong to its creators and that content providers are best served by a data format that does not bind them to particular script languages, authoring tools, and delivery engines. Instead, data format should provide a standardized, vendor-independent, level playing field upon which different authoring and delivery tools may freely compete. Consequently, the flexible but structured nature of XML documents leads to a variety of ways of implementing XML effectively. AIML, the scripts used by ALICEbots, are so created to meet the special needs of chatting bots in various fields.

1.4 Prolog

Prolog, which stands for PROgramming in LOGic, is the most widely available language in logic programming paradigm. Since it deals with symbolic or non-numeric computation, Prolog is frequently used in AI field where manipulation of symbols and inference about them is common. Prolog consists of some facts and certain rules. When a Prolog program is executed, some queries are made against the facts and rules. If they follow the rules, they are proved to be true, otherwise false. Some of the features of Prolog are "logical variables", which means that they behave like mathematical variables, a powerful pattern-matching facility called unification, a backtracking strategy to search for proofs, uniform data structures, and interchangeable input and output. Therefore, an important application area of Prolog is parsing. Actually Prolog was created to express grammar rules with logic and to formalize the process of parsing [10]. The most common logic grammar formalism is definite clause grammars

(DCG). In our implementation, we apply DCG to generate the querying database.

CHAPTER 2

Literature Review

2.1 ELIZA

Eliza is a computer program for the study of natural language communication between man and machine [1]. It is believed to be one of the first public displays of artificial intelligence. Eliza parodies a psychotherapist by answering questions with questions. Input sentences are analyzed on the basis of decomposition rules triggered by key words appearing in the input text. Then responses are constructed by reassembly rules associated with selected decomposition rules. When in conversation with Eliza, the user types in some statements or sets of statements in natural language patterns, using normal punctuation and sentence structures except for question mark. When the control is transferred to Eliza after the user hits the carriage return, Eliza analyzes the user's statement and generates some response and types it out. Following is an excerpt of a typical conversation with Eliza:

```
U: Men are all alike.

E: In what way?

U: they're always bugging use about something or other.

E: Can you think of a specific example?

U: Well, my boyfriend makes me come here.

E: Your boyfriend made you come here.
```

```
U: He says I'm depressed much of the time.

E: I'm sorry to hear you are depressed.

...
```

When an input is read, it is inspected for the presence of a keyword. When such a word is found, the sentence is transformed according to a rule defined for this keyword. Next, the text is composed and printed out to the user. These keywords and their associated transformation rules constitute the script for a particular class of conversation. It is noteworthy that the script of ELIZA is data, not part of the program itself. This means ELIZA is not restricted to a particular set of recognition patterns or responses, and not even to any specific language.

However, there are some fundamental technical problems with ELIZA [11]. The central issue is text manipulation, which regards the transformation rule associated with certain keywords. The pattern matching used by ELIZA provides no way for ELIZA to represent or recognize the grammatical structures, i.e. the syntax, of a sentence. The answer to a pair of sentences, one expressed in active voice and the other in passive voice, but with identical meanings will be different. Besides, ELIZA has no knowledge of anything it talks about. It simply has a set of responses. This lack of initiative, knowledge, and common sense in ELIZA makes a conversation with her increasingly irritating with the program's aimlessness.

2.2 ALICEbot

ALICEbots are the latest creation of artificial intelligence by A.L.I.C.E. foundation. They are capable of imitating human brain with greater speed and accuracy. The original bot was created by Dr. Richard Wallace to approximate life. By analyzing human chats, Dr. Wallace noticed that most conversation utterances start with one of the 2,000 words. Then Alice was created to start a conversation with one of the 2,000 words, obeying what language theorists call Zipf's Law – a very small number of words make up for most of what we say [12]. After being taught 40,000 responses, Alice was able to answer 95 percent of what people said to her.

Following is a conversation transcript with Alice, who was powered by ProgramD with around 25,000 categories:

```
ALICE> Hello there user and thanks for connecting!
user > Hello, Alice.
ALICE> Hi there!
user > What do you know about the weather in Denton?
ALICE> I do not know what weather in Denton is.
       The ALICE chat robot can follow many things, like our discussion
       about weather in Denton. But I did not get that.
ALICE> Can you tell me any gossip?
user > The University is shut down for two days because of winter storm
       warning.
ALICE> Do you like your school?
```

```
user > yes, sure
ALICE> What's your favorite thing at school? OK let's do it. Oh, you are
       a poet.
user > What is artificial intelligence?
ALICE> Artificial intelligence is the branch of engineering and science
       devoted to constructing machines that think. I like AI a lot too.
       Where do your interests lie?
user > My interest is in AI field, as well. I am doing programming with
       Prolog for AI applications.
ALICE> Have you told many people that your interest is in AI field as
       well? It's good to meet someone who is programming with Prolog
       for AI applications.
user > Nice to talk with you.
ALICE> Thank you for your kindness.
user > bye
ALICE> See you later user.
```

We can see that this chatbot is capable of getting control of the topic and leading the conversation by asking user questions. Alice chat robot is based on experiments specified by Alan M. Turing[13] in 1950. It uses a sophisticated pattern-matching case-statement technology to create a very convincing illusion of natural conversation. In other words, Alice uses case based reasoning[14], which by comparison makes Eliza keyword searching quite simple. Alice is capable of learning and storing information

from the user, which means it can spread gossip told to it by the end user. Therefore, the conversation is very lively and impersonates human-like behavior. However, it is still a machine, and the model of learning is supervised by its botmaster, who monitors the robot's conversations and creates new AIML content to make the responses more appropriate, accurate, believable or more human-like. This process is called targeting, with several algorithms for detecting new patterns in dialog being developed[15].

2.3 Semantic Web

According to World Wide Web Consortium (W3C), the semantic web is the abstract representation of data on the World Wide Web, based on Resource Description Framework (RDF) and other to-be-defined standards[16]. It extends the current web and re-defines the meaning of information available on the web to enable computers and people to work cooperatively [17]. Most of the Web's content available today is designed for humans to read, but not for computer programs to manipulate meaningfully. When we search the Internet, we can retrieve the relevant information based on a key word. However, we must sift through the returned WebPages to find the good ones that actually meet our intended meaning. Chances are 80 percent of the returned pages are irrelevant, because the search engine does not check the meaning of the keyword in the context of a particular page. It simply returns the web page for human to read, which is really time consuming.

We would, however, like to have a powerful tool that can build related information for us. If we browse the web and find an interesting conference we would attend,

what we would like to have instantly by clicking a button are the time and place and links to other documents including the pages of other people who would attend the conference. When we decide to attend this conference, by clicking the register button, our own calendar would record the time and date of the conference with links to other useful information such as flight schedule, event description, etc. Instead of doing all these jobs manually by us, we would like to have all these information retrieved and sorted by some computer agents. Currently, web data can be shared by applications using different XML DTDs or schemas, databases could be linked and data could be wrapped with SOAP (Simple Object Access Protocol) to communicate.

The use of Semantic Web will bring structure to the meaningful content of Web pages. The Semantic Web is not a separate web but an extension to the current one. For the semantic web to function, computers must have access to structured collections of information and sets of inference rules that they can use to conduct automated reasoning. Artificial Intelligence workers have studied such systems before the Web was developed. Knowledge representation is still currently in its primitive form. Searching will yield irrelevant information because word sense needs to be disambiguated[18]. Some good demonstrations exist, but it has not yet changed the world. It contains the seeds of important applications, but to realize its full potential it must be linked into a single global system. A software agent will roam from page to page, carrying out the complicated tasks for users. It will link data from the Web and use it more effectively for new discovery, automation, and integration across various applications.

2.4 Open Mind Project

Despite years of research in artificial intelligence, computers still lack any knowledge and are incapable of understanding the common sense of life [19]. A simple rule like "Every person is younger than the person's mother" is taken for granted by humans, but needs to be taught to computers in order to figure out family relationships. Computer scientists have been trying very hard to find ways to teach computers all the common sense knowledge, but they have not been very successful. For one reason, there are so many things to be learned, and for another, there is not a good project to adequately collect information.

Sponsored by MIT Media laboratory, Open Mind Commonsense Project seeks all resources available online, given the fact the web is popular enough to reach a large population, to teach computers how to describe and reason about the world, especially about people and their goals, activities, and interests. All these informations will take the Internet from its current state as a giant repository of web pages, to a new state where it will be able to think about all the knowledge it contains to make it a living entity that could be deployed in real life for various purpose mentioned in Chapter 1.

Computers are smart when we feed them software to do complex things like playing chess, or designing airplane engines. However, we cannot make computers communicate with people in the way human interacts with each other. The main problem is that computers know nothing about people. To give computers common sense we must program them with knowledge about many different areas. The artificial intelligence is about how to make systems that are abundant with many types of knowledge

and many ways of thinking about different things, i.e., the capacity of commonsense reasoning. A large database is indispensable, but we need to give computers many different ways of using that knowledge to think, giving computers methods of reasoning, planning, explaining, predicting, and all the other things we human do.

With abundant data in the database, at the Media Lab, researchers are exploring several kinds of applications. They developed systems that could reason what the user really wants when the user types a piece of information. For instance, when the user types "my cat is sick" into the search engine, the system reasons, and gives back the answer of "Search for a veterinarian in your area". This answer is based on the following facts:

People care about their pets:

- People want their pets to be healthy
- My cat is my pet
- I want my cat to be healthy
- A veterinarian heals sick pets
- A veterinarian makes sick pets healthy
- I want to call a veterinarian
- A veterinarian is a local service

Therefore: Search for a veterinarian in the user's area

With large database available, sense disambiguation is an important issue. There have been several researches done in this area[20]. Achievements in building large unambiguous corpus gives natural language processing much more power in searching for accurate information.

2.5 VISTA Project

The VISTA project (Virtual Interactive Story Telling Agents) is part of University of North Texas Digital Storytelling Project [21]. The virtual agents interact with users through natural language query/answer patterns derived from the analysis of narrative content. Such interactions allow a user to learn about the content of a story by asking the questions he/she personally needs or wants to ask. VISTAs are coded as a combination of AIML scripts [2, 15] to support AI applications and rules in a Prolog knowledge base. The components of the system includes a web client with a video/audio agent interface, an HTTP and media server, a story database, AIML Programs, and Prolog Knowledge Base Programs executed by the JINNI systems[6, 22].

In VISTA project, the query/answering process about a given story is modeled as a combination of story specific AIML query/answer patterns, a generic AIML pattern library, and a set of JINNI 2002 classes implementing the underlying storytelling ontology that emerges from classifying the stories by themes, motifs, genres, and other indexing schemes. Agents use two orthogonal techniques to answer questions. The first technique uses transcripts from human chat sessions as analogical sources

to replicate what humans do directly; the second technique is inferential/deductive. It tries to identify, at least partially, what the focus of interest is in the question and consults the story classification hierarchy and related dialogue patterns to handle unknown situations. It uses the lexical knowledge base together with an advanced rule based inference mechanism for understanding stories[23].

2.6 AIML

2.6.1 General Overview

AIML[24] is a subset of XML. AIML stands for Artificial Intelligence Markup Language and was developed by ALICEbot community between 1995 and 2000 to enable people to input knowledge into chat-bots. The syntax of AIML is compliant with XML. Its goal is to enable pattern-based, stimulus-response knowledge content to be served, received and processed on the Web and offline, in the manner that is currently possible with HTML and XML[25].

AIML describes a class of data objects called AIML objects and partially describes the behavior of computer programs that process them. There are two basic units in AIML, *topics* and *categories*, either parsed or unparsed. Parsed data consist of characters, some being character data, while other being AIML elements. AIML elements encapsulate the stimulus-response knowledge contained in the document. Character data within these elements is sometimes parsed by an AIML interpreter, while sometimes left unparsed for later processing by a responder[25].

In AIML, the tags most important to us are <aiml> <category> <pattern>, and

<template>. <aiml> is the root tag that marks the beginning of an AIML document. Correspondingly, </aiml> marks the ending of such document. A unit of knowledge forms a category and is denoted within the <category> </category> tag set. Each category consists of an input question (the <pattern> part), an output answer(the <template> part), and an optional context(the <that> part) or the previous utterance the bot says. The AIML pattern consists of words (letters and numerals only), spaces, and the wildcard symbols _ and *. This pattern language is case invariant. The question that users input will be compared to the content inside <pattern> </pattern> tags. If there is a match, the contents of <template> will be retrieved as a response to the question asked. More generally, this reply is transformed into a mini computer program which can save data, activate other programs, give conditional responses, and recursively call the pattern matcher to retrieve responses from other categories. Following is a sample AIML script:

```
<category>
<pattern>YES</pattern>
<that>FOOBAR</that>
<template>You said yes</template>
</category>
```

in which the robot says “Foobar” and if the user answers with “yes”, then the robot will respond with “You said yes”.

2.6.2 Program D

ALICEbot engine is implemented in many languages like Java, C, C++, Perl, Lisp, and PHP scripts. The first edition of A.L.I.C.E. was implemented in 1995 using SETL, an unpopular language or unknown language based on set theory and mathematical logic. It later migrated to the platform independent Java language in 1998. Codenamed "Program A", this implementation of A.L.I.C.E. and AIML was implemented in pre-Java 2 and was very popular among the research community. Launched in 1999, Program B brought a breakthrough in A.L.I.C.E free software development. More than 300 developers contributed to this version. AIML was transitioned to a fully XML-compliant grammar, and this version won the Loebner Prize, an annual Turing Test, in January 2000. Program C was the C/C++ implementation of AIML released in 2000.

Program D was recoded by Jon Baer[2] based on Program B, a pre-Java 2 implementation. Many features of Java 2, like Swing and Collections, were added into this new version. It still uses AIML scripts as underlying pattern database of ALICE-bots for question answering. Though AIML can handle well individual patterns, it has limitations in generalization and inference capabilities. Due to the limitations of AIML, we decide to implement this engine in Prolog, i.e. extend AIML-based pattern processing with a logic-based engine and deploy it with JINNI[26]. However, there are good features of AIML that we try to emulate in our implementation, like the pattern deduction algorithm used in Program D.

2.6.3 AIML Pattern Mapping

AIML pattern mapping is done via Graphmaster technique [15]. The Graphmaster has a collection of nodes called Nodemapper. Pictorially it is a hierarchy of Nodes, each being either a root, a leaf, or both. The root of Graphmaster has about 2000 branches, each accounting for the first word of all the patterns. A <template> tag is attached to each leaf node.

The search for an answer to a question is as follows. To match an input of the user to a pattern, there are three basic steps. The underscore "_" has the highest priority, followed by atomic word match, and then by the wildcard "*". For example, if the first word of the input string is "X", first of all, we check if the Nodemapper contains the key "_". If we find such a node, we traverse the subtree of this node and find the subsequent words following "X". If there is a match, we return the <template> attached to the last branch. If not, we backtrack to the root and put back the words of input string one by one. Next, we look for the branch that contains exactly the word "X" and apply the same algorithm to find the match. If no match is found with the atomic word branch, we try the third path, the wildcard path. For an empty/null input, if the Nodemapper contains the <template> key, then a match is found. If the root Nodemapper has a wildcard "*" node and it points to a leaf, then we guarantee there is a match for any input string.

The above-described matching algorithm is a highly restricted version of backtracking. The patterns need not to be ordered strictly though if only the underscore branch comes before any word branch and the wildcard branch comes after any word

branch.

2.6.4 Logical Deduction in AIML

AIML has some simple reasoning capabilities. It doesn't use any logic engine as Prolog, and the results are obtained with AIML only. The question pattern being considered is the "What" questions. For example:

```
What does a bird have?
What do birds have?
What does a raven do?
What else does a bird have?
What else do ravens do? A bird has a beak, a tail, lungs, eyes, wings,
feathers, and is cold-blooded.
```

The first step is to construct an ISA hierarchy in AIML [27]. The entries look like this:

```
<category>
<pattern>ISA RAVEN</pattern>
<template>A Bird.</template>
</category>

<category>
<pattern>ISA BIRD</pattern>
<template>An animal.</template>
```

```
</category>
```

From the database of ISA's, there are many answers to a simple "What" question. Therefore, AIML uses the <random> tag to group these answers:

```
<category>
<pattern>ISA CHICKEN</pattern>
<template>
<random>
<li>A Bird.</li>
<li>A Food.</li>
</random>
</template>
</category>
```

As with any database, there is a default case for ISA relation where an answer will be provided even if the first argument cannot be identified:

```
<category>
<pattern>ISA * </pattern>
<template>Unknown</template>
</category>
```

To build a knowledge base for "What" questions, the next step would be creating Has and Does relations. Similar to ISA relation, the <random> tag is used to group many facts together:

```
<category>
<pattern>WHAT DOES A BIRD HAVE</pattern>
<template>
<random>
<li>Lungs.</li>
<li>An eye.</li>
<li>A beak.</li>
<li>A tail.</li>
<li>A wing.</li>
<li>A feather.</li>
<li>Cold blood.</li>
</random>
</template>
</category>
```

For the default case, the symbol "UNKNOWN" appears in the patterns. Here, the art sense of the writer comes into play.

```
<category>
<pattern>WHAT DOES UNKNOWN HAVE</pattern>
<template>
<random>
<li>Imagine no possessions.</li>
<li>I don't know</li>
```

```
<li>The same as everyone else?</li>
</random>
</template>
</category>

<category>
<pattern>WHAT DOES UNKNOWN DO</pattern>
<template>
<random>
<li>Exist.</li>
<li>I don't know.</li>
<li>The same as everyone else?</li>
</random>
</template>
</category>
```

To reduce symbols, we first transform a variety of grammatical forms into simpler inputs. The tag <srai> is designed to recursively find the next pattern. Given the following knowledge base, we will be able to transform "WHAT DOES A X DO", "WHAT DOES AN X DO" and "WHAT DO X DO" into single canonical form like "WHAT DOES X DO".

```
<category>
<pattern>WHAT DO * DO</pattern>
```

```
<template><srai>WHAT DOES A <star/> DO</srai></template>
</category>

<category>
<pattern>WHAT DOES A * DO</pattern>
<template><srai>WHAT DOES <star/> DO</srai></template>
</category>
```

The inferential abilities of AIML are realized through the <srai> tags. Basically, we will reduce "WHAT DOES X DO" to "WHAT DOES Y DO" if there is ISA relation between X and Y. Similarly we can reduce "WHAT DOES X DO" to "WHAT DOES X HAVE". With the same technology, AIML is capable of reducing plural forms to singular forms. For each "WHAT" question, therefore, we would be able to find an answer to it. Ultimately, there is an "WHAT DOES AN UNKNOWN HAVE" category as the default case to halt the recursion.

CHAPTER 3

Implementation

3.1 System Architecture

The AIML scripts developed by Dr. Richard Wallace serve as the query database for various ALICEbots. They contain rich information about our daily life, and rely on recursive methods to find an answer to a given question. With targeting mechanism, the bot is capable of acquiring new knowledge. The Loebner winner Program D, introduced in Section 2.6.2, uses AIML scripts to carry out the query/answer tasks. In our project, we reuse AIML scripts but in a Prolog based knowledge processor. The whole project could be separated into two phases: Database Generation Phase and Runtime Phase.

In the database generation phase, the original XML compliant AIML scripts are sent to event driven SAX parser. With JINNI engine, patterns of questions and answers are extracted to a Prolog QA pattern database. This phase is implemented by a modified XML-Prolog conversion package and it supports bi-directional conversion between AIML and Prolog clauses.

In the runtime phase, the QA patterns in Prolog clauses serve as the underlying database and provide canned and inferential answers to incoming queries. It is achieved by a Prolog inference engine that searches the Prolog QA database recursively in a finite amount of time for the matched answer and presents it to the user

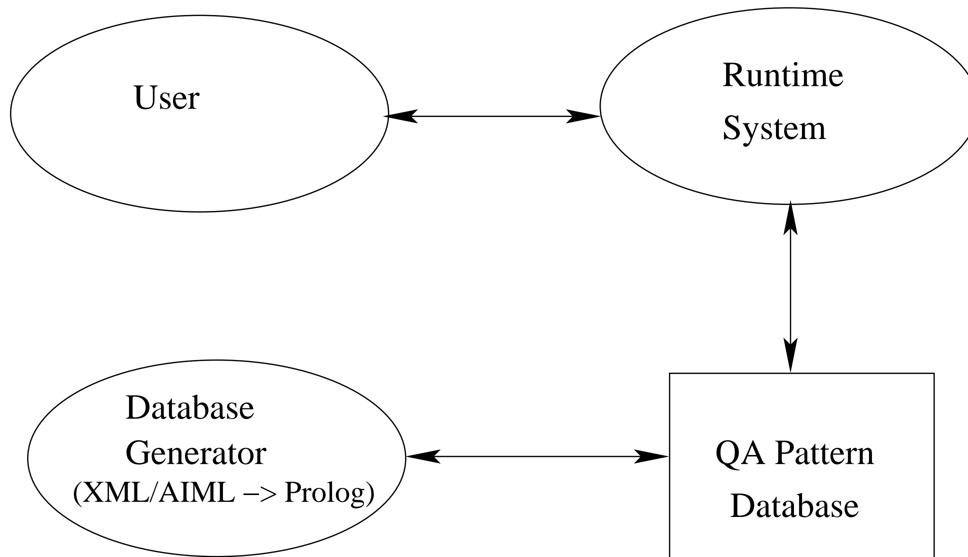

Figure 3.1: System Architecture

via a GUI. This match engine uses standard DCGs and works with JINNI 2003 or BinProlog.

When the user inputs a question where there is not a ready answer, we will prompt the user for a possible solution and insert dynamically into the database. This learning capacity enhanced the power of our search engine greatly.

3.2 Database Generation

3.2.1 AIML Parser

In Figure 3.2, the AIML/XML file is the origin of file conversion. There are 34 standard AIML files:

```
std-65percent.aiml   std-gossip.aiml       std-religion.aiml
```

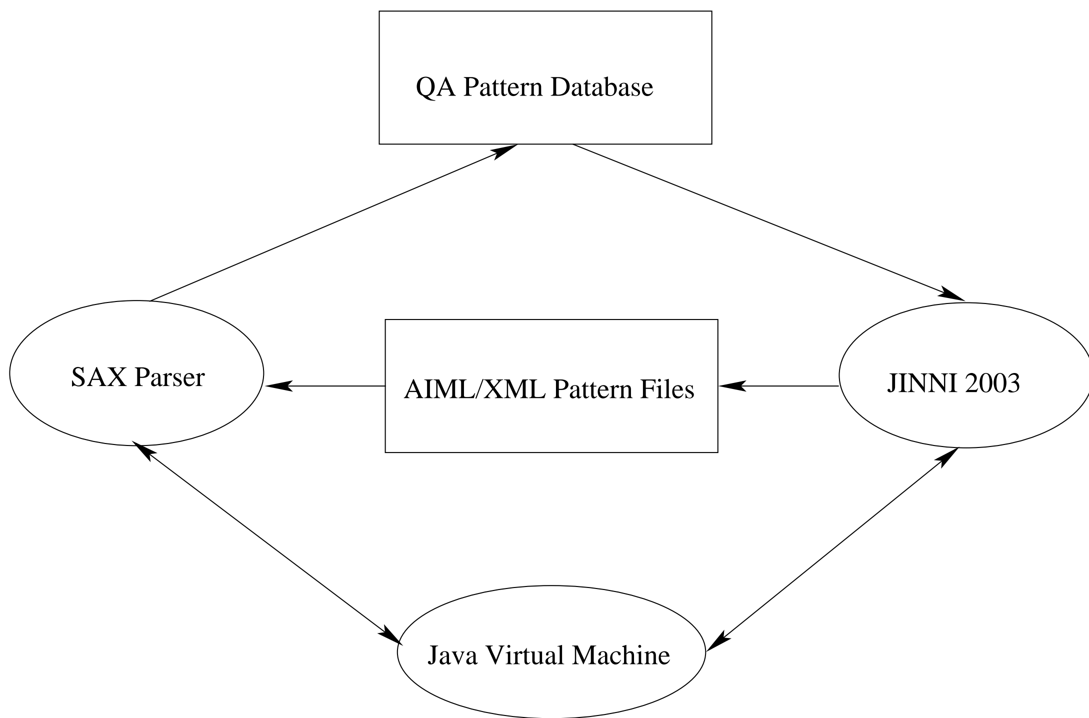

Figure 3.2: Database Generator

`std-atomic.aiml`
`std-botmaster.aiml`
`std-brain.aiml`
`std-connect.aiml`
`std-dictionary.aiml`
`std-disconnect.aiml`
`std-dont.aiml`
`std-errors.aiml`
`std-gender.aiml`
`std-geography.aiml`
`std-german.aiml`
`std-hello.aiml`
`std-inactivity.aiml`
`std-inventions.aiml`
`std-knowledge.aiml`
`std-lizards.aiml`
`std-login.aiml`
`std-numbers.aiml`
`std-personality.aiml`
`std-pickup.aiml`
`std-politics.aiml`
`std-profile.aiml`
`std-robot.aiml`
`std-sales.aiml`
`std-sextalk.aiml`
`std-sports.aiml`
`std-srai.aiml`
`std-suffixes.aiml`
`std-that.aiml`
`std-turing.aiml`
`std-yesno.aiml`

Each of these files specifies a special feature for ALICEbots. For instance, the file std-gender.aiml lists the common English names that ALICEbots could encounter. This file finds the user's name and determines the gender for a given name. Its information will be used in the (set name gender male) category. The file std-profile.aiml tries to learn specific things about an individual user and the file std-knowledge.aiml contains the common knowledge and is an important resource in conversation. When we start our chat bot, all these files are loaded into the brain of our bot.

An AIML to Prolog parser is written in Java to convert AIML files to Prolog based clauses. Well documented XML processors in literature include Simple API for XML (SAX) parser and Document Object Model (DOM) parser from org.w3c.dom and org.xml.sax [28]. However, they take different approaches. DOM parser creates a

tree of nodes when a Document object is created based on an XML file. To access the leaf knowledge, we need to traverse the whole tree. SAX parser, on the other hand, is event-driven. The information of an XML file is accessed as a sequence of events. But we need to create our own custom object model and a listener class listening to SAX events.

Given the fact that only a small subset of database needs to be searched and processed during information query, we developed an event-driven parser based on JINNI 2003's Java based SAX parser. Using an event driven parser avoids large XML trees in memory. Having a cache for external Prolog facts ensures that other large data is only brought into memory as needed. In our model, we use SAX parser to parse iteratively the AIML files and create Prolog clauses which could be ported to JINNI.

We have AIMLContentHandler and AIMLContentPrinter classes to perform this conversion. The AIMLContentHandler class implements org.xml.sax.ContentHandler. The important methods we override are startDocument(), endDocument(), startElement(), endElement(), and characters().

3.2.2 Structure of Prolog Clauses

The original AIML files have a list of categories within the <aiml> tags. Ecah category could have its own attribute and a list of data. Therefore, when creating Prolog clauses for AIML scripts, we use recursively the form '$ai'(tagname, [AttributeList],

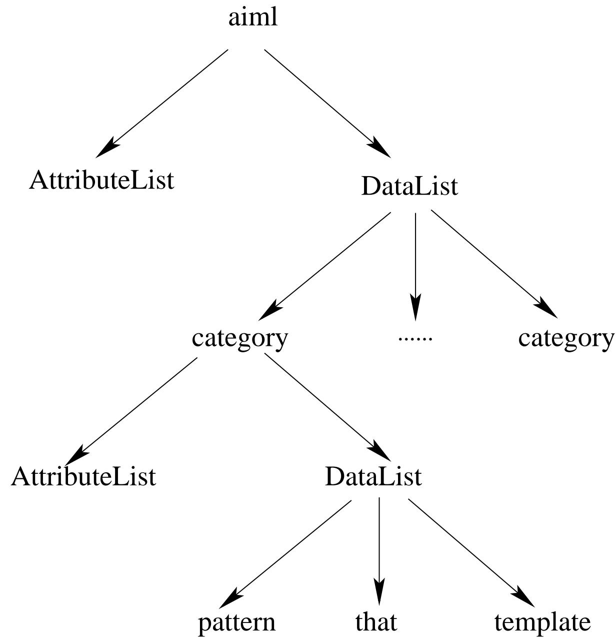

Figure 3.3: Tree Representation of Prolog clause of AIML scripts

[DataList]) as shown in Figure 3.3 to represent the original information. Any DataList could contain another '$ai'(tagname,[AttributeList], [DataList]) as its data. If the attribute list is empty, empty square brackets [] are used to represent empty attribute. Similarly for empty data list, there is a pair of empty brackets for it. For string data, we use the form '$s'("Data") because if a string is represented with single quotes, it will be loaded in memory, and we can run out of memory space fairly quickly. On the contrast, list representations for strings are faster and more convenient for conversion. The outer most tagname is always aiml and each source file is converted to a single file consisting of one Prolog clause.

In the end, the original AIML tree representation is expressed with recursive '$ai'(tagname, [AttributeList], [DataList]) form. It enables the Prolog database to

have a uniform structure and could be parsed conveniently in QA extraction. Given a source file of:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<aiml version="1.0">
<!-- Free software (c) 2001 ALICE AI Foundation -->
<!-- This program is open source code released under -->
<!-- the terms of the GNU General Public License -->
<!-- as published by the Free Software Foundation. -->

<meta name="author" content="Jon Baer"/>
<meta name="language" content="en"/>

<category>
<pattern>BOT HOW MUCH IS *</pattern>
<template>
<random>
<li>The answer is <javascript><star/></javascript>.</li>
<li><javascript><star/></javascript> I think.</li>
<li>I think it's <javascript><star/></javascript></li>
<li>Let me check, it's <javascript><star/></javascript></li>
</random>
</template>
```

```
</category>

<category>
<pattern>EVALUATE *</pattern>
<template>
<javascript><star/></javascript>
</template>
</category>

<category>
<pattern>WHAT IS NATURAL LANGUAGE</pattern>
<template>
Natural language is what artificial intelligences speak.
</template>
</category>

<category>
<pattern>SHOW ME A WINDOW</pattern>
<template>
<display target="sized" height="400" width="400" status="1">
http://www.alicebot.net
</display>
```

```
OK.
</template>
</category>

<category>
<pattern>YES</pattern>
<that>FOOBAR</that>
<template>You said yes</template>
</category>

</aiml>
```

The converted aiml clause would be like this:

```
'$ai'(aiml, ['version' = '1.0'],[
'$ai'(meta, ['name' = 'author','content' = 'Jon Baer'],[]),
'$ai'(meta, ['name' = 'language','content' = 'en'],[]),
'$ai'(category, [],['$ai'(pattern, [],['$s'("BOT HOW MUCH IS *")]),
'$ai'(template, [],[
'$ai'(random, [], [
'$ai'(li, [],['$s'("The answer is"),
'$ai'(javascript, [],['$ai'(star, [],[])]), '$s'(".")]),
'$ai'(li, [],['$ai'(javascript, [],[
'$ai'(star, [],[])]), '$s'("I think.")]),
```

```
'$ai'(li, [],['$s'("I think it's"), '$ai'(javascript, [],[
'$ai'(star, [],[])])]),
'$ai'(li, [],['$s'("Let me check, it's"),
'$ai'(javascript, [],['$ai'(star, [],[])])])])])]),
'$ai'(category, [],['$ai'(pattern, [],['$s'("EVALUATE *")]),
'$ai'(template, [],['$ai'(javascript, [],['$ai'(star, [],[])])])]),
'$ai'(category, [],['$ai'(pattern, [],['$s'("WHAT IS NATURAL LANGUAGE")]),
'$ai'(template, [],['$s'("Natural language is what
'Artificial Intelligences' speak.")])]),
'$ai'(category, [],['$ai'(pattern, [],['$s'("SHOW ME A WINDOW")]),
'$ai'(template, [],['$ai'(display, ['target' = 'sized',
'height' = '400','width' = '400','status' = '1'],[
'$s'("http://www.alicebot.net")]), '$s'("OK.")])]),
'$ai'(category, [],['$ai'(pattern, [],['$s'("YES")]),
'$ai'(that, [],['$s'("FOOBAR")]),
'$ai'(template, [],['$s'("You said yes")])])]).
```

For simplicity, we omit the comment lines. They do not affect the integrity of the original source file. The information can be stored externally and put back when needed.

3.2.3 Lossless Information Conversion

To make sure there is no information loss during AIML to Prolog conversion, the converted Prolog files are tested by a modified JINNI XML Parser. There are seven classes in this package: Main, XMLConverter, GenericJinniXMLHandler, FullXMLHandler, SimpleXMLTermHandler, XMLTermBuilder, and XMLTermHandler. The Main class loads the Prolog-based AIML file and starts the usual JINNI top level. In XMLConverter, the prolog terms is converted to an XML representation. The GenericJinniXMLHandler class acts as a no-action adaptor for various SAX parser based Handlers. It calls the parser, sends end notification, and properly terminates the Prolog side processing. The FullXMLHandler inherits from GenericJinniXMLHandler and sends events to Prolog for each member of the DefaultHandler SAX adaptor. It ensures all Java functionalities are being done in Prolog as well. SimpleXMLTermHandler only overrides a few methods needed in parsing XML represented Prolog terms. The XMLTermBuilder builds Prolog terms directly for fast processing. It creates the illusion of having a DOM parser. The complete "syntax tree" it builds is a set of Prolog terms. Similarly XMLTermHandler focuses on handling XML represented Prolog terms.

To test the integrity of converted AIML files in Prolog form, first of all, we run the modified XMLConverter class to change the Prolog file back to AIML format. We use a reflection-based interface for Prolog-to-XML converter, which takes a file containing Prolog clauses and builds an XML representation in another file. Because we use a well-formed Prolog clause, converting it back to XML representation with

JINNI Engine means recognizing three objects, namely Fun, String, and Integer of JINNI. JINNI Fun class implements external representations of Prolog compound terms. It is a functor of the form Symbol/Arity. Our newly created Prolog clauses have recursively '$ai'(tagname, [AttributeList],[DataList]) form and because we use '$ai' and '$s' to denote tagname and string data, the tagname is conveniently taken as a functor name when we convert it back to XML representation.

The new AIML file is converted for a second time to Prolog format. Comparing this version against the first version reveals no difference between two output files. Therefore, we conclude that this three way conversion, i.e. AIML to Prolog, Prolog to AIML, and AIML to Prolog again, ensures no information loss at all.

3.3 Prolog Database Creation

After aiml clauses are converted to Prolog format, the next step we perform is the creation of a query database in the form of qa(Q,A). Each qa corresponds to a category in the original AIML file, where the Q corresponds to <pattern> content and the A is what <template> contains.

The first issue of database creation is reading all Prolog files of aiml scripts and extract the qa's into a qadb file. BinProlog has a predicate called dir2files that search iteratively through a directory and read the content into a database. The codes we have for this purpose is:

```
go1 :-
  retractall(qa(_,_)),
```

```
dir2files(ai_files, FileList),
forall(member(X, FileList),
       (namecat(ai_files,'/',X, File),
        term_of(File, Term),
        process(Term))),
write(qa([[who, is, 'X',.]],
      [[i, know,a,lot,but,i,do,not,know,anything,about,'X',.]])),
       println('.'),
write(qa([[what, is, 'X',.]],
      [[i,know,a,lot,but,i,can,not,answer,'X',.]])),
       println('.'),
write(qa([[my, name, is, 'X',.]],
      [[hello, 'X',what,can,i,do,for,you,?,.]])),
       println('.'),
told,
listing(qa).
```

In this part, we clean up the qa database for the first time and append the facts read from each file to qa list. When this process is finished, all qa facts are in memory, ready for query. We insert a few default answers for who and what questions as well.

When processing aiml term, we fully utilize the feature present with the aiml clause, which contains recursive '$ai'(tagname, [AttributeList], [DataList]) information. With the pictorial representation as illustrated in Figure 3.3, we can tell there

is a clear iteration of the form '$ai'(tagname, [AttributeList], [DataList]).

Therefore, after reading into memory one file at a time, the next step we do is scanning the attribute list as well the data list from the outmost tag aiml. Then we recursively scan all categories. We also insert a default answer to handle situations where there is not a ready answer in the database. The code that does this part of job is:

```
process('$ai'(aiml, AttrList, Children)) :-
  scan_attr_list(AttrList),
  scan_all_categories(Children, _).
```

The top level scanning we do on the aiml term is finding all categories included in a file. Other tags parallel to category is ignored. Since there is more than one category in a single aiml file, our scanning method recursively goes through all categories until the file comes to an end. For each category, we insert into our qadb file the content of pattern() as Q, and the content of template() as A in pairs. Another important tag is that(), which records the previous utterance of the user. The codes that perform these functions are:

```
scan_all_categories([], []).
scan_all_categories([C|Cs], [qa(Q,A)|QAs]) :-
  scan_category(C, Q, A),
  tell_at_end('qadb.pl'), write(qa(Q,A)), println('.'),
  scan_all_categories(Cs, QAs),
  assert(qa(Q,A)), !.
```

```
scan_category(C,Q,A) :-
  ( C = '$ai'(category, _, Children)
              -> process_category(Children, Q, A)
   ; Q=[[hello,'X',.]], A=[[hi,thanks,for,calling,me,'X',.]],!
  ).

process_category([], [], []).
process_category([C|Cs], Q, A) :-
  (C = '$ai'(pattern, Attr, Child)
        -> get_Q(Attr, Child, Q), process_category(Cs, Q, A)
  ;C = '$ai'(template, Attr, Child)
        -> get_A(Attr, Child, A), !
        % once get the answer, no need to proceed
  ;C = '$ai'(that, Attr, Child) -> process_that(Attr, Child)
  ;process_category(Cs, Q, A)
  ).
```

When inserting into qadb file, Prolog has special features that require attention. Any word that starts with a capital letter is considered a variable. Therefore, we must be very careful about it. Sometimes, though, we purposefully insert a variable X for later inference during querying database.

3.4 Information Deduction

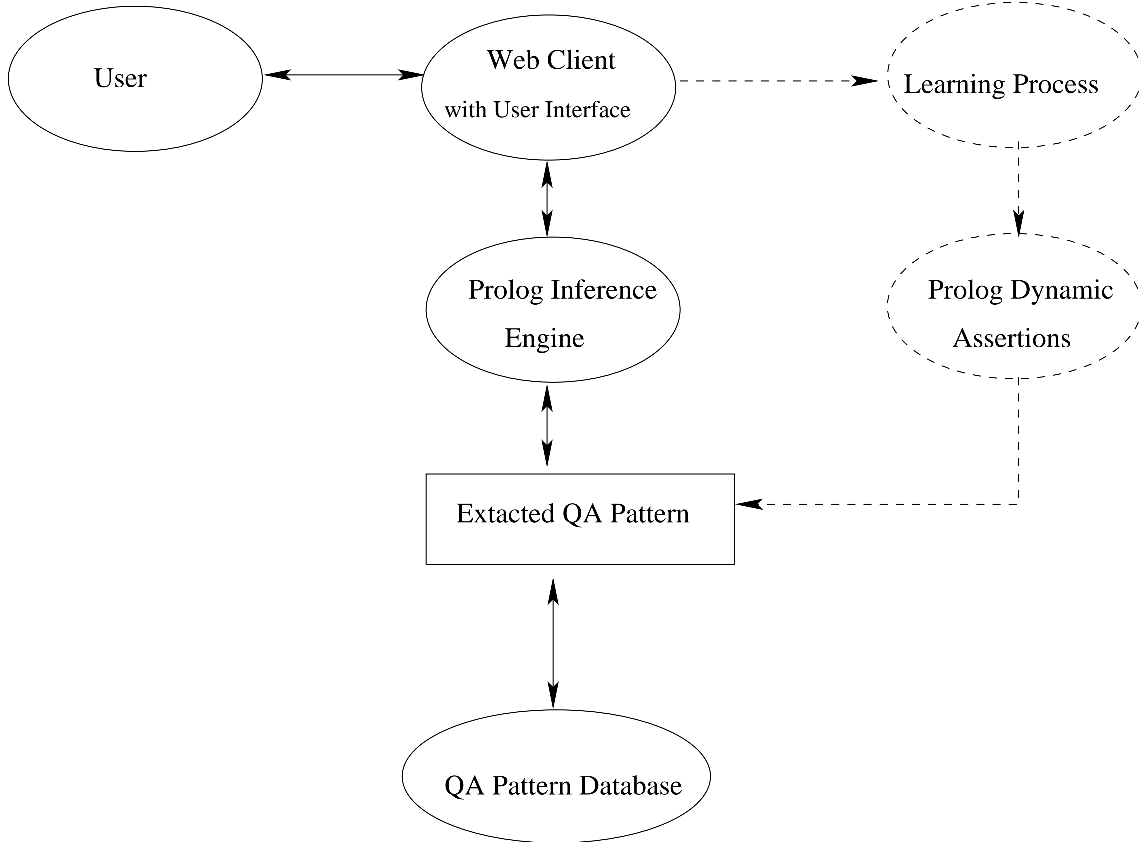

Figure 3.4: Runtime System

As in Figure 3.4, the user interacts with a web client whose function is simply displaying a graphical user interface (GUI), reading in user inputs and displaying system output. We call our bot Emily, and she is capable of providing useful information about various aspects.

First of all, the input string is converted into a list of words. If the input list consists of words like “bye”, “quit”, or “exit”, Emily will stop running and exit.

Otherwise, the search engine searches the database for the most appropriate answer. The codes we have for the initiation of Emily is here:

```
go :-
  nl,print('EMILY > '), println('Hello, user'),
  repeat,
    print('User > '),
    read_line(Input),
    name(Input,Cs),
    to_lower_chars(Cs, Ls),
    to_words(Ls, Ws),
    if_any((member(bye, Ws);
            member(quit, Ws);
            member(exit, Ws)),
           (nl, print('EMILY >'),
            println('See you'),
            abort),
           (qtoa(Ws, Answer),
            nl, print('EMILY > '),
            write_words(Answer), nl
            )),
     Ws = bye.
```

There might be more than one answer to a question. The original AIML files uses

<random> tag to select from a list of answers randomly. We preserved this feature by choosing one element from the answer list and write that out to the user.

```
qtoa(Input,Answer):-
  qa(Q,Ans),
  match_pattern(Q, Input), !,
  length(Ans, L),
  if_any(L > 1,
       (random(X), I is X mod L, N is I + 1,
        findnth(Ans, N, Ans2),
        flatten(Ans2, Answer)),
        flatten(Ans, Answer)
        ).
```

The internal pattern processing engine has a pattern matching library. The predicates works on tokens as well as char, integer, even clause streams. The match_pattern predicate words with JINNI 2000 or BinProlog. It uses standard DCG because DCG allows designing pattern processor more easily in that DCG handles patterns elegantly without complex use of append operations. Initially the complete list of tokens is given as the hidden input argument for the DCG processor.

```
match_pattern(Pattern,Input):-
  match_pattern(Pattern,Input,_Before,_After).


match_pattern(Pattern,Input,Before,After):-
```

```
  nonvar(Pattern),nonvar(Input),
  [Before|Pattern]=NewPattern, % adds initial collector variable
  !,
  match_internal(NewPattern,Input,After),
  !. % matches the first occurrence of the pattern
```

This match_internal(+Pattern,+InputList,-LeftOverList) matches pattern against input list and returns what is left over in -LeftOverList part. So after the sequence of patterns has been found, the LeftOverList contains tokens left unprocessed. In our pattern processor, a DCG rule is invoked as a (+InputList, -LeftOverList). The DCG transformation maps a grammar rule like

```
a --> b,c,d.
```

to

```
a(S1,S4):-b(S1,S2),c(S2,S3),d(S3,S4).
```

Forcing the LeftOverList to be empty brakets [] might trigger extensive backtracking.

3.5 Running Scripts

As described in previous sections, we first convert all AIML source scripts to corresponding Prolog format and load Prolog clauses into memory when our chatbot Emily is started. Following is a short script of a dialogue with Emily.

```
?- go.
```

```
EMILY> Hello, user
User > Why is the sky blue.
EMILY> The sky is blue because of the atmospheric optical effect known
as raleigh scattering.
User > Do you know about the recent happenings in Denton?
EMILY> I know a lot but I can not tell you about the recent happenings
in Denton.
User > What is natural language?
EMILY> Natural language is what artificial intelligences speak.
User > Who is Yoko Ono.
EMILY> Miss yoko ono is the widow of John Lennon.
User > Who is Reagan.
EMILY> The best president the united states had in the 20th century.
```

3.6 Application with Agents

As described in 1.1, a chat agent can simulate human conversation. It can be used to incorporate animated characters into software application and web pages. Microsoft provides an agent[29] with Text-to-Speech (TTS) and Speech-to-Text (STT) functions and the TTS engine is supported in its Internet Explorer on Windows 2000 and Windows XP. This agent technology is borrowed here and linked to our database. Agent Genie, the parrot, asks questions and Agent Merlin, the funny man, answers them.

CHAPTER 4

Conclusion

The goal of natural language processing in the world of conversational agents is to enable a computer to emulate human languages so that people can talk to computers as though they are addressing another person. The early ELIZA program carries on conversation with people, and provides canned answers based on keywords. ALICEbots, the chat agents developed by A.L.I.C.E. foundation, use AIML to power up their conversational abilities and use animations with realistic facial expression. They can listen to human speech and speak as well.

As part of this thesis, an AIML parser is developed in Java using the event-driven SAX parser. This parser converts AIML scripts to Prolog clauses. These clauses are reversible to the original AIML form without losing any information. With the correctness of this conversion, we guarantee the database created is as accurate and informative as the original AIML scripts.

Secondly, a Prolog question-answer (QA) database is generated by analyzing the converted AIML patterns. With this functionality, we can translate a large corpus of AIML based QA patterns into a format ready for knowledge processing.

To enhance the searching capability of our chat agent, a Prolog based pattern processing engine is developed supporting a subset of the converted AIML patterns. The recursive search is at its preliminary stage. When linked to Microsoft agent technology, the search agent could interact with user via Text-to-Speech (TTS).

This technology has broad applications ranging from entertainment to online teaching. Our patterns have been integrated in the VISTA (Virtual Story Telling Agents) project as a way to enhance their functionality by reusing existing libraries of AIML patterns for common sense reasoning and improved conversational abilities.

BIBLIOGRAPHY

[1] J. Weizenbaum, Computer Program for the Study of Natural Language Communication Between Man and Machine. Available at http://i5.nyu.edu/ mm64/x52.9265/january1966.html.

[2] A.L.I.C.E. AI Foundation. Artificial Intelligence Markup Language (AIML). Technical report, A.L.I.C.E. AI foundation, 2002. Available at http://alice.sunlitsurf.com/TR/2001/WD-aiml/.

[3] Claire, Virtual Representative of SprintPCS. Available at http://www.sprintpcs.com.

[4] Artificial Intelligence. Movie. 2001 Warner Bros. Directed by Steven Spielberg. Available at http://aimovie.warnerbros.com/.

[5] G. A. Agha and N. Jamali. Concurrent Programming for Distributed Artificial Intelligence. Multiagent Systems: A modern Approach to DAI, Ed. Gerhard Weiss, Chapter 12, pp.505-531, MIT Press, 1999. Available at http://yangtze.cs.uiuc.edu/Papers/Agents.html.

[6] BinNet Corporation. Jinni 2002 A High Performance Java and .NET based Prolog for Object and Agent Oriented Internet Programming. Technical report, BinNet Corp., 2002. Available at http://www.binnetcorp.com/download/jinnidemo/JinniUserGuide.html.

[7] P. Tarau. Fluents: A refactoring of Prolog for Uniform Reflection and INteroperation with External Objects. In Proceedings of CL 2000, Jul 2000. Ed. John Lloyd. London.

[8] Extensible Markup Language. Technical Report. Available at http://www.w3c.org/XML/.

[9] J. Bosak. XML, Java and the Future of the Web. Technical report. Sun Microsystems. Available at http://www.ibiblio.org/pub/sun-info/standards/xml/why/xmlapps.htm.

[10] L. Sterling and E. Shapiro. The Art of Prolog: Advanced Programming Techniques, 2nd edition, MIT Press, Mar 1994.

[11] M. Sharples, D. Hogg, C. Hutchison, S. Torrance and D. Young. Computers and Thought: A practical Introduction to Artificial Intelligence, Oct 1996. Available at http://www.cs.bham.ac.uk/research/poplog/computers-and-thought/index.html.

[12] C. Thompson, Approximating Life, New York Times. Available at http://www.nytimes.com/2002/07/07/magazine/07WALLACE.html.

[13] A. Hodges. Alan Turing and the Turing Test. Available at http://www.turing.org.uk/publications/testbook.html.

[14] W. Pieters. Case-based Techniques for Conversational Agents. Master's thesis. University of Twente, Jun 2002, Available at http://wwwhome.cs.utwente.nl/ pieters/report.pdf.

[15] Richard Wallace. AIML Pattern Matching Simplified. Technical report, A.L.I.C.E. AI foundation, 2002. Available at http://www.alicebot.org/documentation/matching.html.

[16] Resource Description Framework (RDF) Model and Syntax Specification. Technical report. Available at http://www.w3.org/TR/REC-rdf-syntax/.

[17] T. Berners-Lee, J Hendler and O Lassila. The Semantic Web. Scientific American, May 2001. Available at http://www.scientificamerican.com/article.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21&catID=2.

[18] R. Mihalcea and D.I. Moldovan. Word Sense Disambiguation Based on Semantic Density, in Proceedings of COLING-ACL 1998 Workshop on Usage of WordNet in Natural Language Processing Systems , pp.16-22, Montreal, Canada, August 1998.

[19] Push Singh, The Open Mind Common Sense Project. Available at http://www.kurzweilai.net/articles/art0371.html?printable=1.

[20] T. Chklovski and R. Mihalcea. Building a Sense Tagged Corpus with Open Mind Word Expert. In Proceedings of the Workshop on "Word Sense Disambiguation: Recent Successes and Future Directions", ACL 2002 pp. 116-122.

[21] E. Figa and P. Tarau. The VISTA Project: An Agent Architecture for Virtual Interactive Storytelling. In TIDSE 2003, Eds N. Braun and U. Spierling, Darmstadt, Germany, Mar 2003.

[22] P. Tarau and V Dahl. A Logic Programming Infrastructure for Internet Programming. In Artificial Intelligence Today – Recent Trends and Developments. Eds. M. J. Wooldridge and M. Veloso. Springer, LNAI 1600. ISBN 3-540-66428-9. pp.431-456.

[23] E. Figa and P. Tarau. Story Traces and Projections: Exploring the Patterns of Storytelling. Available at http://logic.csci.unt.edu/tarau/research/2002/figa_tarau_wnet_tidse.pdf.

[24] AIML Tutorial, Technical report, A.L.I.C.E. AI foundation, 2002. Available at http://www.pandorabots.com/pandora/pics/wallaceaimltutorial.html.

[25] AIML Specification. Technical report. Available at http://alicebot.org/TR/2001/WD-aiml-1.0.1-20011025-006.html.

[26] P. Tarau. Inference and Computation Mobility with Jinni. In The Logic Programming Paradigm: a 25 Year Perspective, Eds. K.R. Apt and V.W. Marek and M. Truszczynski, 1999, Springer,ISBN 3-540-65463-1, pp.33-48.

[27] Richard Wallace. Simple Logical Deductions in AIML. Available at http://www.aiml.info/modules.php?name=Content&pa=showpage&pid=1.

[28] XML Standard API. Available at http://xml.apache.org/xerces2-j/javadocs/api/index.html.

[29] Microsoft Agent. Available at http://www.microsoft.com/msagent/default.asp.