Multi-Agent Systems for Internet Information Retrieval using Natural Language Processing

CS-98-24

Vlado Kešelj

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
e-mail: vkeselj@uwaterloo.ca

September 21, 1998

Abstract

In the context of the vast and still rapidly expanding Internet, the problem of Internet information retrieval becomes more and more important. Although the most popular at the moment, the keyword-based search engine is just one component in a complex software mosaic that needs to be further developed in order to provide a more efficient and scalable solution.

This report will argue that the multi-agent approach is a viable methodology for this task. The main issue is how the natural language processing could be used in it, as well as why it should be used. Two implementations and their theoretical foundations are presented: One is the natural language parser generator NL PAGE, which produces parsers in C or Java; and, the other one is the communication part of the multi-agent framework MIN. The higher levels of the framework are also discussed and a simple implementation of a multi-agent system is presented.

Acknowledgements

I wish to thank my supervisors, Professor Graeme Hirst and Professor Forbes Burkowski, for their guidance and patience throughout my program; and, especially, for their time and effort during the final phase in which this thesis was created.

I also wish to thank to the second readers of the thesis, Professor Fahiem Bacchus and Professor Chrysanne DiMarco, for their efforts and valuable comments. I wish to extend my gratitude to Professor Frank Tompa for providing my financial support, and for giving me his comment and suggestions, and some references relevant to my work.

Last, but not least, I wish to thank my wife Tanja and my son Stefan for their continuous support, encouragement, and sacrifices in the course of my graduate studies.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Internet has vast potential for further development, new applications, and new solutions. One of the challenging problems that needs an improved solution is the problem of Internet Information Retrieval (InIR).[1]

In this chapter, we will start with the description of the problem of InIR in Section 1.1. Section 1.2 is an introduction to multi-agent systems; Section 1.3 is an introduction to natural language processing; Section 1.4 gives an overview of related work; and, Section 1.5 presents an overview of the remaining chapters.

[1]All abbreviations used in this document are expanded the first time they are used. They are listed in the Appendix A along with their expansions and the page numbers of their first occurrences.

1.1 Description of the problem

The environment of our problem is the Internet, seen as a large, distributed, and heterogeneous source of information. Although this approach is somewhat narrow compared to the Internet with its full functionality, the multi-agent (MA) part of the methodology that is going to be presented can be directly applied in solving any massively distributed Internet problem. The presented demonstration system MIN (Multi-agent system for Internet information retrieval using Natural language processing) will illustrate this approach at the implementation level of detail.

What does the Internet information space consist of? Mostly, on-line WWW (World Wide Web) documents; but, we can easily see that it is more. The Internet, from the user perspective, is perceived through a set of applications based on the point-to-point communication links that are provided by the TCP/IP (Transmission Control Protocol/Internet Protocol) protocol stack. For example,

- Many applications frequently end up with a full-sized InIR problem, where we want to find a relevant document, relevant item, or, generally, a relevant point in the information space consisting of Telnet sites, news groups, news group postings, FTP (File Transfer Protocol) sites, Gopher documents, and WWW documents (pages, movies, audio files, code, radio broadcasts).

- E-mail: How can we find out if someone has an e-mail address, and how can we find that address? Finding interesting mailing lists is a still better example.

- The r-utilities (remote utilities), such as remote login (`rlogin`), remote shell (`rsh`), and remote copy (`rcp`): One might be surprised that this kind of application has anything to do with InIR. But even in this case, we may want to find out the IP quadruple address that corresponds to a domain name, or vice versa. This is already solved through the domain name system; but, it is interesting to notice that this is a problem that can be classified as an InIR problem and could be treated in a more general context.

We now introduce a couple of terms that are going to be used throughout the document. The *Internet information space* is the abstraction of the Internet we are dealing with. We will denote it as the *InI-space.* The InI-space is a set consisting of a finite number of elements which are called *resources.* They are also referred to as the *Internet resources,* the *points* in the InI-space, or the *elements* of the InI-space. The space is dynamically changing, so, strictly speaking, it is actually a sequence of sets where each set corresponds to a certain moment in time. Informally, we simply say that InI-space is a "dynamic set," since we assume that it is in any case an intractable task to determine the complete content of that space at any moment. We will assume that the resources are exactly those items that are addressable by URL's (Uniform/Universal Resource Locators).

The basic problem of InIR is the following: A *user's query* is given. It can be in different forms but we assume that the most direct form is the natural language (NL) form. The task is to find a *relevant subset* of the InI-space, i.e. a subset of the *relevant resources* for that query. The relevance relation is semantic, high-level, and even subjective; so we can only approximate it with a computational relation. This basic problem can be extended to a more general form. First, we want to *rank* the relevant subset, i.e. to give the *relevance order* of its elements with the most relevant resource being the maximal element, the second-most relevant appearing next, and so on. We also want to be able to give answers that can be derived from the relevant subsets; e.g., to find the number of elements in the relevant subset.

Two additional conditions are important to keep in mind when dealing with the InIR problem. First, InI-space is changing at a fast rate; and, second, a large number of users are doing InIR in parallel.

The most popular tools currently used for InIR are the search engines[2]. They use the classical IR (Information Retrieval) approach, which consists of gathering site descriptions associated with their URL's into an index, and, afterwards, matching user queries to relevant descriptions. The relevant descriptions are ranked and mapped to a list of URL's, which is returned to the user, who can now manually browse the sites. The matching algorithm

[2]Appendix C contains the URL's of some popular search engines, all-in-one pages, and meta search engines.

is always keyword-based, which leads to certain drawbacks.

Search engines are very useful InIR systems and we are not trying to replace them, but they have certain inherent limitations and the concept needs to be extended. They can be the basis for further development of some new InIR systems.

Let us summarize some significant disadvantages of the classical approach used in the search engines:

1. *Rigidity*: The strategy assumes a large, unitary retrieval system with a hard-wired interface to the Internet. There is an inherent risk of becoming obsolete in such a rapidly changing environment. For example, HTML (Hypertext Markup Language) is continually being enriched with new features, and one can never know what new communication form will be developed tomorrow.

2. *Expensive total indexing*: With classical IR systems, the process of incorporating incoming information into a database (e.g., updating inverted files, or signature files) is expensive, but it is not a significant drawback if it is a relatively rare operation. However, if we take into account that the InI-space is dynamically changing and that the links are unreliable, then frequent updates of the database are necessary, making this problem a major issue.

3. *Expensive one-to-all communications*: A centralized IR system would require frequent updates, which means frequent Internet communications. Every centralized system on the Internet used by many users is a bottleneck, and, as a consequence, there is a growing number of search engines. Each search engine updates its index by making one-to-all connections. Hence, the same data is retrieved many times without effective reuse. This creates a substantial and unnecessary network load. If there are N Internet resources and M search engines, then to keep their index databases up to date, they would need to make $M \cdot N$ connections in relatively short periods of time, which is far from being an effective solution for a large M.

   An unwanted situation also arises concerning the processing cost: if different users make the same or similar queries, their results cannot be effectively reused because of the lack of communication among retrieval clients and information servers that are not directly related.

4. *Hidden information*: A large amount of information is hidden inside archives (databases), such as FTP sites and various search-engine interfaces. Since we do not have access to complete information in those databases, we cannot maintain a keyword list for the hidden information. Consequently, if a user wants a document that is possibly in an archive we need a more sophisticated way to recognize that archive as relevant than keyword matching. We can use only the front-end page of the archive, a readme file, or something similar, which are only semantically connected to the user's query.

5. *Keyword barrier*: Using ordinary keyword matching, one can easily retrieve an unmanageable number of Internet pointers about a popular topic. This can happen if we are looking for pointers to Web pages, or, especially, if we are retrieving news articles from Usenet. A more advanced technique such as conceptual matching is required to get around this problem. This limitation is sometimes called the *keyword barrier* [Mau91a].

There are alternative approaches to get around these problems.

One trivial approach is the use of *all-in-one* pages. All-in-one pages simply provide an easy way for the user to query various search engines in parallel. This method directly depends on the search engines, so we cannot say that it brings any essential improvements. An interesting issue here is what the search-engine owners have to say about this kind of access. Because, their interest lies in the advertising which is effective only during a direct user access. One can expect that they would obstruct this kind of indirect access by periodically changing the user interface, for example.

A similar strategy and the same kind of problems are evident in another popular approach—*meta-search engines*. They also query multiple search engines, but they do other more sophisticated subsequent processing—such as filtering, ranking, and combining—and then present results to the user.

Other options include multi-agent systems and use of natural language processing. A synthesis of these two approaches is the key concept pursued in this thesis.

1.2 Multi-agent systems

The development of MAS's (Multi-agent Systems) is based on work in two areas—artificial intelligence (AI) and distributed systems.

Let us try to explain why use of AI methods is natural in InIR. Currently, the Internet can be imagined as a flat low-level structure based on the point-to-point communication links activated with considerable, manual (human) participation. A human is presumably an intelligent entity and so the Internet content and its navigation mechanisms are based on this *intelligence assumption.* In this way, it has been possible to get such a huge information space, practically integrating the whole planet, in such a short time. But now, since such an intelligence-assuming environment has been created, it requires AI techniques to manage it. Probably the most obvious example is a simple Web page. If we want to automatically use its content in a fashion more sophisticated than collecting keywords, or collecting links embedded in it for further navigation, then the most flexible, robust, and appropriate way to do this is to try to understand some of its content and to reason about it. This necessarily leads to the realm of AI.

Agents. The notion of agent has become both a crucial term in AI, and a frequent buzzword having a wide range of definitions. Nevertheless, there are some common characteristics which roughly describe what an agent can be.

In their book *Introduction to Artificial Intelligence—a Modern Approach* [RN95], Russell and Norvig based the whole problem of AI around the notion of agent. They say, on page 31:

> An **agent** is anything that can be viewed as **perceiving** its environment through **sensors** and **acting** upon that environment through **effectors.**

This looks like a very general definition and a lot of things end up being agents if we apply it. But on page 33 of the same book, we concur with the following sentence:

> The notion of an agent is meant to be a tool for analyzing systems, not an absolute characterization that divides the world into agents and non-agents.

Considering general resources about agents, we can mention two more references: Stan Franklin and Art Graesser's paper [FG96] gives various definitions of agents and proposes the beginning of a taxonomy of autonomous agents. In "Agent-Oriented Programming" [Sho93], Shoham presents a different approach, introducing *agent-oriented programming* (AOP) as a specialization of object-oriented programming (OOP).

So, what is the concept of an agent that we want to adopt before going into the task of specializing it for our specific problem? We can address various levels of generality, and emphasize various characteristics of an agent. The starting position is that an agent is any entity inhabiting an environment, sensing that environment and acting within it. This is called the *reactive* property of an agent [FG96]. It is a quite general definition, and many things can be called agents: humans, thermostats, computer programs, and so on. We can agree that humans are classified as agents; actually, they are the ideal agents—which is a good reason to avoid giving a precise computational definition of an agent. Conversely, it is not very useful to see a thermostat or every computer program as an agent.

As Russell and Norvig's quote suggests, the issue is not to distinguish precisely between agents and non-agents, but to (imprecisely) decide when it is advantageous to see something as an agent. We can decide that by looking at the typical properties that agents possess. The following description should be qualified by adding the word "usually" in each sentence:

- Agents are relatively autonomous, working on behalf of someone else—another agent or a human.
- Their environment is complex and they can sense the results of their actions.
- The mapping percept-to-action is neither trivial nor simple, but the sequence of perceptions and actions is done in order to achieve a high-level goal.

- Agents are complex and high-level systems.
- We talk about the states of an agent in terms of mental categories; so, an agent has goals, beliefs, commitments, choices, knowledge, and so on.
- Agents are flexible and able to learn.
- Agents are temporally continuous; i.e., their state is persistent over a long period of time.
- Because of the complexity of its environment, an agent usually does not know all about it but only about one part of it which is closest—its immediate neighborhood. This implicitly declares the environment to be a space and associates the agent with its position in that space. Then, we can talk about the agent's mobility. An agent can be *mobile* or *immobile.* An agent is mobile if it can autonomously change its location, otherwise it is immobile. If an agent is mobile, then we can discuss its *mobility of state*—an agent's state is mobile if it does not change during a move from one location to another.

These are general characteristics of an agent. We are interested in a more specific type of agent—*Internet agents.* Internet agents are computational software agents—which distinguishes them from biological and robotic agents [FG96]. Their environment is the Internet and they communicate with other agents and users. They are not mobile. Our agents can also be classified as *information gathering agents* because of their IR goals.

As mentioned, Shoham [Sho93] proposed a new programming paradigm—agent-oriented programming (AOP). We are going to rely on some of his ideas concerning the relation between AOP and OOP, but it is more appropriate to discuss them after multi-agent systems are introduced.

Multi-agent systems (MAS's) If two or more agents can communicate among themselves, then we can consider a MAS. A group of agents constitutes a MAS only if it is designed in that way. For example, if we have two chess-playing agents playing chess, then it is not very useful to call them a MAS. A set of two or more agents is a MAS only if they work cooperatively in solving their tasks. A MAS can be seen from the outside as a single agent composed of less complex agents, which are then called the *sub-agents.*

The paradigm of AOP is closely related to OOP. In OOP the data and algorithms encapsulated as objects. Programming involves passing messages between objects, i.e. by invoking the object methods. In this approach, a programmer is provided with a powerful way of modeling the world. Compared to structured programming, for example, the higher level of abstraction in OOP makes the handling of complex tasks much easier. The main parallel between OOP and AOP is the object-agent analogy. Like objects, the agents also encapsulate data in the form of their mental states and have certain *behaviors* according to their algorithms that are analogous to object methods. The agents inside a MAS communicate by passing messages, as well. The notions of inheritance and overloading have their equivalents in AOP, although we need much more experience and understanding of MAS's to be able to effectively use these design features. Perhaps the most important similarity lies in the programmer's mind in the processes of designing an object-oriented program and a MAS. In this view, it makes sense to talk about AOP only in the context of MAS's and not individual agents.

The differences are also quite obvious. Objects are packed physically in one program while agents are more separated, often running on different machines and communicating over physical communication links. Agents have a higher level of encapsulation; e.g., it is not possible to access data in an agent directly, it has always to be done by passing messages. Although objects in OOP hide the actual way in which their methods are executed, their interface is precisely defined and a lot of actions are done sequentially. On the other hand, agents have a more general interface and do not significantly depend that much on each other's specific message formats. Messages have a more flexible structure expressed in a high-level language. Agents normally work in parallel, with the exception of situations when one agent is waiting for another. Some of these differences disappear if we have in mind distributed object systems, which are even more close to MAS's.

A MAS can be *open* or *closed.* A MAS is open if the outside agents, not created as a part of it, can join the system. Otherwise, a MAS is closed.

Our approach is based on an open, Internet, information gathering MAS.

It relies on a more general idea about the open Internet MAS's. Namely, if we have a standard method for inter-agent communication on the Internet, then various Internet agents and Internet MAS's could combine and interact autonomously, forming large super-MAS's. The first step in this direction is the definition of a standard inter-agent communication language (or, more generally, the communication form). The inter-agent communication language has to be very rich to express the universe of various tasks, functions, facts, etc. among agents. A lot of work has already been done in the area and, although there is not a standard, the two languages KQML (Knowledge Query and Manipulation Language) [Fin97b] [LF97] and KIF (Knowledge Interchange Format) [Fin97a] make a combination that seems to be closest to a standard. Actually, of these two, only KQML is an inter-agent communication language, but KIF fits in as a necessary complement. KQML is used to express the agent's attitude towards certain information, such as querying or stating. The information itself is embedded in the KQML message and it is expressed in a language called the *content language*. The content language can be KQML, or some other language like KIF, SQL, Prolog, etc. We have chosen KIF, which is a knowledge representation language.

1.3 Use of Natural Language Processing (NLP)

People have been trying to make computers understand natural language since the creation of computers themselves. It has proven to be a hard problem—it has not been solved yet, and it is not likely to be solved soon. The only techniques we can use are the partial ones typically developed in the symbolic programming languages Lisp and Prolog, which are inherently not as efficient as some lower-level languages.

NLP has four levels of processing: *lexical*, *syntactic*, *semantic,* and *discourse* (*pragmatic*).[3] The lexical level is the level of word recognition. Actually, the units of a NL that are recognized are called *lexemes.* Besides recognition, any lexeme processing is done at this level, for example, finding the part-of-speech tags, stemming, prefix and suffix analysis, and all other kinds of word derivations. At the syntactic level, the syntactic structures of sentences are constructed. The semantic level of processing reveals the meanings of isolated sentences. At the discourse level, we build our global world model and decide how it is affected by the NL speech that we analyze.

The idea of using NLP for IR is not new, and the connection between the two areas is quite straightforward. In the domain of IR, we typically have a large collection of documents in a NL. If we take into account that the easiest way for the user to express her/his wishes is NL as well, the connection is there. The relation between these two areas is not superficial nor simply formal, as might be concluded from the above argument. In the core of the most challenging questions in IR is our (in)ability to deal with the higher levels of NLP. When we say that a document is relevant for a query we mean that the meaning of the document satisfies the meaning of the query—that kind of matching would be a part of the perfect IR.

However, we are far from this goal since we do not know how to do complete NLP. Fortunately, we can still do partial processing and use its results. Stemming is a low-level NLP algorithm that is very frequently used in IR systems. So, for example, if the user is looking for documents containing the word "house," then, using a stemming algorithm, a program can decide

[3]There are more levels of NLP but we are concerned here only with these four. More information can be found in [All95].

that documents containing the word "houses" are relevant. Another useful technique is semantic expansion. For example, if documents containing the word "died" are considered relevant, then documents containing the word "killed" should be also taken into consideration.

A much more involved use of NLP in IR is done in the form of *conceptual* IR. It is an attempt to do the ideal IR—to match the meaning of the user's query to the meaning of the retrieved documents. As the term *conceptual* suggests, meaning is represented by concepts. Since this approach relies on higher levels of NLP, it is difficult to implement. Issues include concerns such as deciding what a concept is, how to extract concepts from the NL texts, and how to do concept matching.

The inefficiency of existing NLP systems is a major obstacle in using them in IR. If we want to use an NLP system to analyze the documents in a large document collection—and it is always large in IR—then it has to be very efficient and robust to be useful in practice. As mentioned earlier, existing NLP systems are typically not very efficient.

What can be said about InIR in this context? In InIR we know the document collection in advance—it is the InI-space; and we know it is huge. In a collection of that size, the use of NLP is needed even more, because keyword-based retrieval methods tend to retrieve too many documents. On the other hand, it is more difficult to use NLP, not only because we have more documents in the document collection, but also the collection is very dynamic with a large number of documents being created and deleted. Thus, emphasis must be put on the efficiency of the processing, which is precisely a weak point of existing NLP systems. A solution is to restrict the NLP to its lower levels, which is not what we have in mind. A more positive approach is to implement distributed NLP so that the processing cost is widely distributed in the same way the Internet resources are. The MAS's are appropriate for this task.

Although the combination of NLP and MAS's has already been discussed by some researchers, it is still quite novel in the area of InIR (relative to InIR and MAS's, which are also novel). They are used independently: For example, in [CH95], NLP is used in InIR without a mention of MAS's; while in [Kar96], a MAS for InIR does not use NLP but the system is based on the vector-space model—an advanced, but nonetheless keyword-based IR method.

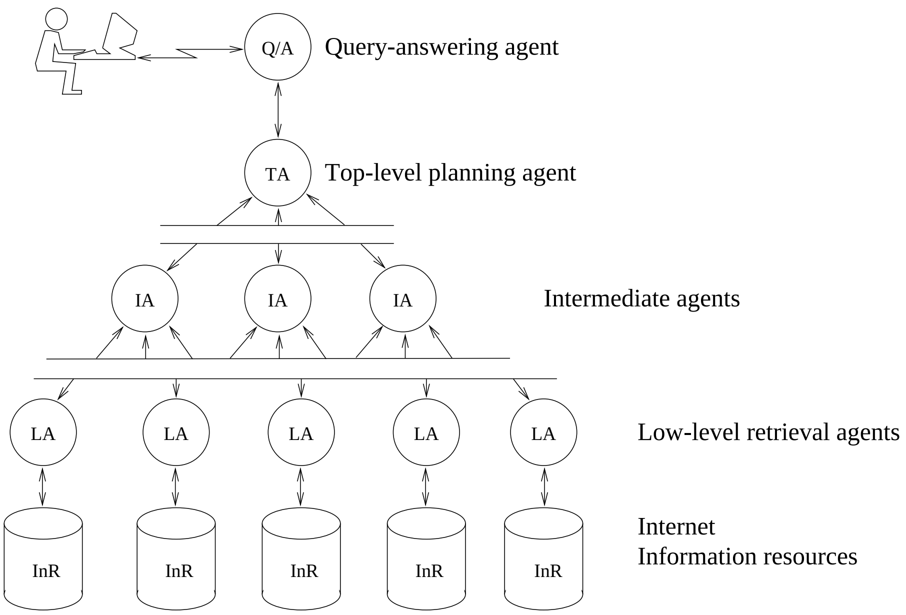

Figure 1.1: General multi-agent architecture

1.4 MIN approach

We can now present an outline of our approach, which we call MIN, for MAS + InIR + NLP, because it combines MAS's and NLP in solving the problem of InIR.

A generalized scheme of a MAS that we use for InIR is shown in Figure 1.1. A user communicates with the *query-answering agent* (Q/A-agent, Q/A, QA). The Q/A-agent transforms the query into an appropriate internal format and passes it to the *top-level planning agent* (T-agent, TA). The T-agent develops a high-level plan and communicates more specific tasks to the *intermediate agents* (I-agents, IA). We need the I-agents to effectively communicate, exchange, and reuse knowledge gathered by various agents working on different tasks and/or for different users. We can have no I-agents at all, or have one or more levels of I-agents in the agent hierarchy.

Finally, several low-level retrieval tasks are sent to the bottom of the hierarchy—to some of the *low-level retrieval agents* (L-agents, LA). Each of the L-agents is capable of making a specific type of connection or perhaps connect to just one specific Internet resource. For example, an L-agent can be capable of connecting to a specific search engine. It can open a connection, form an appropriate query, get the results and pass them up the agent hierarchy. The role of the I-agents and the T-agent in this bottom-up direction is to filter and to combine information. Results finally reach the Q/A-agent, which presents them to the user and the interaction continues.

These are the four general types of retrieval agents. However, the actual agents are more specialized than this. They are designed with more specific capabilities and, since they have persistent knowledge bases, they can specialize in certain domains over time. This is especially true of the I-agents, which are very loosely defined.

The functionality of the I-agents is split into two sets of tasks:

1. transforming and passing the query in a top-down direction, and
2. transforming and passing the results in a bottom-up direction.

A natural question arises at this point: Why do we not assign these tasks to two different types of agents? One important expected feature of the I-agents is their ability to reuse information in the fashion of a cache memory. After filtering and passing up the results, the I-agents can keep the results (selectively or not) in their persistent knowledge base and use them if a similar query comes up. They would not be able to do that if they never saw the results.

There can be specialized I-agents that are not part of this two-direction information flow. They can be recruited by another I-agent to do a specific task that might be only remotely related to information gathering. We will leave this option open but it is not the main concern of our discussion.

There are two areas where NLP is used: in the user-MAS interaction and in the process of resource indexing and matching. These two areas map to the Q/A-agent and to the I-agents. The Q/A-agent translates the user's NL query into an internal form and, later on, it can use NL generation to produce results. The I-agent activities of resource indexing and matching are based on NL processing of the resource content or the information associated with the resources, like an FTP readme file, the front page of a search engine, or a manually created description. These two types of NLP are not merely distinguished by the agent types; the differences are deeper than that. In Q/A-agent processing, we prefer more precise NLP, even though it may take more running time—usually, only one sentence is processed per user query so the time is not a problem. On the other hand, it is preferable that the user is properly understood. The I-agent has the opposite requirements. It processes a large number of documents; hence, it is important that it takes as little time as possible. Parsing correctness is not as vital—if the parser cannot parse the whole sentence, the sub-sentence phrases can still be useful.

Now, that we have completed the introduction of the MIN approach, we can explain why this approach is a good solution to the InIR problem. Let us review this approach in the context of the disadvantages associated with the standard IR approach that we listed in Section 1.1.

1. *Rigidity*: The multi-agent model is flexible. If we want to adopt a new Internet communication form, then we only have to create a new low-level retrieval agent (L-agent). It will speak the new Internet "language" on one side and the inter-agent communication language on the other.

2. *Expensive total indexing*: There is no need for frequent updates. The agents will try to find the answer dynamically, in the allotted search time: they check their private knowledge bases, ask other agents, and then use the Internet resources.

3. *Expensive one-to-all communications*: The network communication overhead is reduced, since the agents memorize and exchange useful information among themselves. They are not trying to collect information in advance but in a lazy fashion.

4. *Hidden information*: Using the semantical level of NLP, the I-agents can match a user query to its generalizations, for example, a description of an archive where the answer to that query can be found.

5. *Keyword barrier*: The use of NLP and conceptual matching is aimed at overcoming the keyword barrier. Using these more sophisticated

matching criteria, the number of retrieved items is reduced to a manageable number. On the other hand, some relevant resources could be detected even though they could be marked irrelevant in the keyword-based models.

1.5 Related Work

The MIN approach to the problem of InIR is the intersection of several areas of computer science, most of which are new. NLP and IR have some history, but the current work on agents and MAS's, the Internet and InIR is quite recent and, in addition, very dynamic, and relatively immature. In such circumstances it is difficult to present related work in an organized manner that clearly extracts the main structure of the most important results. Instead, we will give a list of the publications, projects, and resources that had the most influence on this approach.

Starting from NLP, Allen's book *Natural language understanding* [All95] is a standard textbook. It provides a good overview of the known NLP techniques and we will frequently refer to it for NLP terminology, explanations, and algorithms.

Conceptual information retrieval is analyzed by Mauldin [Mau91a] [Mau91b]. His FERRET system used NLP to retrieve news articles from Usenet. The problem of NLP inefficiency was challenged with an interesting NLP technique called text skimming. The concepts were represented by case frames, which were filled using scripts called *sketchy scripts.* Relevance matching is done using a lexical knowledge base that includes some semantic knowledge about the words. The system was never finished, although very encouraging results were obtained.

A similar idea was used in the NetSerf system by Chakravarthy and Hasse [CH95].It is a less sophisticated system than FERRET, but the basic idea is similar. They also use NLP to construct case frames, which are compared in the matching process. Given a user query, NetSerf tries to find Internet archives that could contain information relevant to that query. It uses semantic matching based on the semantic relations provided by WordNet [Mil95] [Mea] and an on-line Webster's dictionary. NetSerf is also an experimental system, which is not finished, but showed some good results compared to the classical IR system SMART.

These two systems represent two approaches that have several things in common:

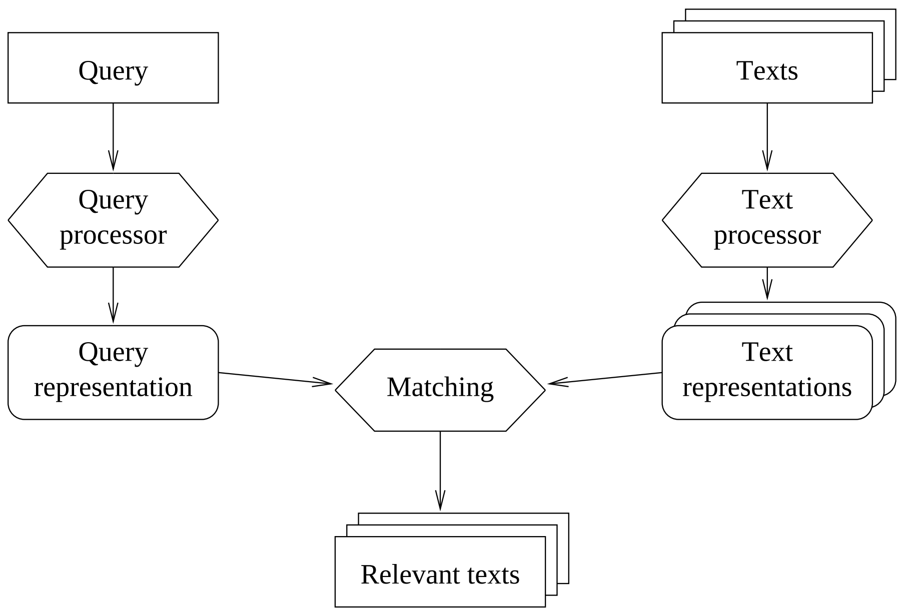

Figure 1.2: Structure of FERRET and NetSerf

- both of them recognize the need for NLP in InIR,
- they use case frames for query and text representations, and for the matching process, and
- they use the global structure of a classical IR system (Figure 1.2).

In some sense, they are complementary since FERRET implemented a parser for incoming texts that were to be searched (Usenet news) and the query processor was never implemented, while NetSerf had a query processor and the text frames were built manually.

Let us mention two other up-to-date systems in the development phase which try to use NLP for InIR. One is the REASON system [AGK+97] by Anikina et al. and the other is the IRENA system [AAT97] by Arampatzis et al. In the REASON system, the document contents are stored into a knowledge base and matching is done using inference rules. In IRENA, matching is based on the "noun phrase co-occurrence hypothesis." The hypothesis states that the nouns embraced by a noun phrase are semantically related. Thus, if the nouns contained in a noun phrase in the user's query are found again in a noun phrase in a document then the document gets a high relevance weight.

The main difference between our approach and the previous systems is that we want to make use of distributed processing in the form of MAS's.

Many general and specific resources about agents and MAS's can be found on the Internet. We have already mentioned some general references—Russell and Norvig's book [RN95], Franklin and Graesser's paper [FG96], Shoham's paper [Sho93], and gave some information about the languages KQML and KIF ([Fin97b], [LF97], and [Fin97a]). In addition to information about KQML and KIF, the WWW site at the University of Maryland Baltimore County contains a rich source of agent-related information. The page about agent-based information retrieval [Sob] contains pointers to many projects relevant to this thesis.

Java parsers for KQML and KIF are available on the Internet. The parser for KQML is contained in the set of Java packages called JATLite [CDR97] developed at Stanford University. JATLite is an agent framework with several layers. The KIF parser JKP [Lua97], developed in Java by Xiaocheng Luan at the University of Maryland Baltimore County, is also used in our demonstration MAS.

Several MAS's for InIR are being developed: At the University of Massachusetts, the researchers in the MAS laboratory group (Lesser, Prasad, Decker, Oates et al. [DLNPW95] [NPLL95] [ONPL94] [Wag]) proposed a MA model for InIR called *cooperative information gathering.* The model is based on the *functionally accurate, cooperative* (FA/C) paradigm for distributed problem solving proposed in 1981. Two systems called Searchbots and MACRON have been built using this paradigm. KQML is used for inter-agent communication with their specific content language TAEMS (Task Analysis, Environment Modeling and Simulation).

Borghoff, Pareschi et al. [BPK+96] [ABRS95] are approaching the Internet as "a kind of gigantic world-wide digital library" and propose agent technology for requirements similar to those of network publication systems. Their model is based on *constraint-based knowledge brokers.*

Birmingham [Bir95] uses a MAS called the UMDL architecture (the University of Michigan Digital Library project) to satisfy the demands of a digital library. A KQML-like language is used for the inter-agent communication.

The SIGMA project implemented by Karakoulas [Kar96] is a *market model* of a MAS for InIR. The inter-agent language is KQML and KAPI (KQML Application Programmer's Interface) [KH95] was used to build the system.

CEMAS (Concept Exchanging Multiple Agent System) by Bleyer [Ble97] is a MAS based on the Java Agent Template (JAT)—i.e., the agents are communicating using KQML. The content language is also KQML.

1.6 Overview of chapters

Chapter 2 presents the model of NLP that we use. Chapter 3 discusses the parser generator system NL PAGE (NL Parser Generator). Chapter 4 is about the MA framework MIN. Chapter 5 gives a description of a test done with the NL PAGE generator and a demonstration MAS based on the MIN framework that uses the NL PAGE system. Chapter 6 concludes with summary remarks, open questions, and topics for future work.

Chapter 2

An NLP Model

A full implementation of the MIN approach[1] would be a large and very ambitious project. The development of this idea will probably last for years in parallel and interdependently with other Internet technologies. Its high-level ideas are so ambitious that, as in most AI areas, it is not clear whether they can be fulfilled in the foreseeable future. But, we believe that even partial results can bring useful improvements to practical applications as well as theoretical foundations if they are developed and analyzed with mathematical precision.

In this sense, a goal of this work is to contribute, in a bottom-up direction, to the low-level basis of the MIN approach by building two tools which are to be used in further research. One tool is the natural language parser generator NL PAGE and the other is the MA framework MIN.

There is also an MA demonstration system MIN built on the MIN framework and acting as a kind of a small glass-box demonstration system. To prevent any possible confusion here let us reiterate that the abbreviation MIN is used to denote three things:

- one is the MIN approach—our general approach to the problem of InIR;
- the second is the MIN framework—the Java [Sun97] framework for

[1] MIN sounds like "minimal," which is not an inappropriate coincidence for our toy MAS, but the whole approach is definitely not minimal.

building MAS's; and,

- the third is the MIN MAS—a MAS built using the MIN framework.

Like our general orientation, the material presented also follows this bottom-up direction. We start with the tool for building NL parsers. This chapter is about the NLP model that is tailored for our approach. The model is precisely described, making the actual implementation design a quite straightforward and easy job.

In Section 2.1 we discuss the requirements for our model; Section 2.2 is about the lexical level of analysis; Section 2.3 is about the syntactic level of analysis; and Section 2.4 is about the higher levels of NLP.

2.1 Requirements

The requirements of our NLP model are determined by the needs of the MIN approach. We need NLP at two points: in the user interaction with the system and in document indexing. These two points dictate two sets of requirements that are different and quite contradictory. In the former case, we can have a better understanding system with possibly slower processing, while in the latter case, the processing has to be fast and parsing correctness can be less demanding. This does not mean that we need only two types of NLP procedures. The I-agents are generally equipped with NLP capabilities. The idea is not to have all of them able to process any kind of text, but to have each agent handling one kind of text. One of the advantages of the MIN approach is that we can have many specialized I-agents that are limited to certain domains. If they have rich domain lexicons, for example, then we can get a high-quality performance and at the same time the processing cost is reduced. In this way, we are actually doing a kind of *parallelized* (or we can call it *distributed*) NLP.

The problem is how to design so many different parsers having different lexicons and, very likely, different grammatical rules. The solution is a parser generator—a system that can generate various NL parsers given various inputs. The input should be easy to manually create and maintain. The process of designing an NL grammar is demanding enough and has to be frequently revised so that we want to remove any procedural burden from it.

More about the process of parser generation will be presented in Chapter 3. Here, we are interested in the model that describes the generated parser. We want the parser to be robust, and efficient in time and space. This implies the following requirements:

- The model has to be simple, efficient, and "low-level."

  The term "low-level" means close to machine language, which in practice means easy translation to efficient C code, for example. Handling all NL phenomena is not an issue of concern here.

- The model will be based on context-free grammar restrictions but we want to avoid any further restrictions such as LL(k), LR(k), and so on. The reason why these restricted grammars are not suited for natural languages will be briefly explained in Section 2.3 when more formalisms are introduced.

- The feature mechanism and movement phenomena have to be handled. They are essential mechanisms in NLP and we will explain them in Section 2.3.[2]

[2]For a more thorough discussion of the feature mechanism and movement phenomena see [All95].

2.2 Lexical analysis

In the parsing of programming languages, as well as formal languages in general, lexical analysis is the first phase during which the input stream is read and transformed into a list of tokens according to the regular expressions associated with the token types. This process is well known and well understood and it may seem trivial to accomplish with a natural language—the tokens are simply words and punctuation marks. However, the situation is not that simple. The problem with NL's is that we almost never have an unambiguous situation. What is a word? Is it `[a–zA–Z]`$^+$,[3] `['a–zA–Z]`$^+$, or `[-'a–zA–Z]`$^+$? Most people will agree that '`semi-colon`' is a word; different opinions might be found regarding '`all-in-one`'; and a construction such as the adjective '`are-you-alive`' in the phrase '`the are-you-alive message`' is definitely not handled by humans as a single word. In practice, the situation gets more complicated because of frequent noise, exceptions, and variations. Humans are good in understanding these—they are even desirable and make the reading more relaxing and easier—but machines have a different taste.

In the MIN context, the lexical analyses done by QA-agents and I-agents are different. For QA-agents, we can define certain conventions that have to be obeyed by the user. The user is expected to make this minimal effort for the sake of much better performance. On the other hand, I-agents are possibly parsing a "who-knows-what" type of gibberish and they are expected to robustly extract as much of the syntactic structure as possible. It is not significant if they fail to find some complex sentence parses, for example. In the I-agent's NLP it is not even an easy task to determine the end point of a sentence and we have to take into account the errors that will propagate from this mistake.

There are three levels of lexical analysis:

1. *preediting,*
2. *tokenization,* and

[3] The notation used for the regular expressions and literal strings is explained in Appendix B.2.

3. *part-of-speech tagging.*

Preediting is the phase in which the raw array of input characters is processed, and its task is to prepare the input for further processing. Preediting is the interface to the outside world—it can expect anything in the *input stream* of characters and its task is to recognize as many elements as possible converting them to a uniform format manageable by the rest of the system. We cannot fix many specifics related to preediting since it greatly depends on the input and, so, mainly relies on heuristics.

For example, the user might type to a QA-agent the following input:

```
␣␣⇥Give␣me␣the␣latest␣␣five␣articles↵
in␣the␣news␣group␣comp.protocols.tcp-ip.↵
```

and a preeditor[4] might change it to

```
Give␣me␣the␣latest␣five␣articles␣in␣the␣news␣group␣
                                         "comp.protocols.tcp-ip".
```

Or, if an I-agent processed the following input:

```
<HTML>↵
<HEAD><TITLE>([.5]<>"<N.9nURRViqz1>"<>[.5])<(-<>29084466657)
                    </TITLE></HEAD><BODY␣bgcolor=#ffffff>↵
<center><h1><a␣href=http://multitext.uwaterloo.ca/>MultiText</a>␣
                          Netnews␣Article</h1></center>↵
<PRE><hr><i>Subject:</i>␣␣␣␣Re:␣Number␣Theory␣Problem↵
```

it might be preedited to

```
<HREF>␣"http://multitext.uwaterloo.ca/"␣MultiText␣Netnews␣Article.␣
             </HREF>.↵Subject:␣"Re:"␣Number␣Theory␣Problem.↵
```

The last output hardly looks like a part of an NL text. Nevertheless, our parser is expected to handle such constructions. They are much simpler than

[4]The *preeditor* is an abstract object performing the preediting phase of analysis. We will feel free to use this kind of "object-oriented" terminology derivations without further notices like this one.

NL constructions and can be handled using the same algorithm. Another thing we want to illustrate is that the preeditor is breaking the input stream into sentences using the string of two characters ‘.↵’. In the case of I-agent processing, the preeditor might not be able to find the end of the sentence. In that case, it will process further until it is pretty sure that the sentence is contained in the captured string, and it is desirable that the stop point has a high probability of being the end of a sentence. The syntactic analysis will then isolatea sentence, if a sentence can be found. In any event, the preeditor outputs a piece of the input stream and we will call it a *sentence.* If there is a possibility of confusion, we will call this sentence the *buffer sentence* and *proper sentence* will denote the “real” NL sentence.

An example algorithm[5] for the preediting by a QA-agent follows:

<u>Algorithm I: **QA Preediting**</u>

Input: *input_stream*

Output: *sentence*

1. Read *input_stream* into *sentence* until the string ‘↵↵’ is read.
2. Remove empty quotations ‘""["]’, and finish a quotation if left open. (*A quotation is delimited by double quotes.*)
3. In the following processing of *sentence* do not touch quotations:
4. Change all invisible characters into spaces.
5. Make sure that the punctuation marks ‘,’ and ‘;’, and the quotations are surrounded by space characters.
6. Remove redundant spaces (at the beginning, at the end, and all strings with more than one space are shrunk to one space).
7. Put in quotes any strings of visible characters which have characters other than letters, ‘-’, ‘’’, ‘,’, ‘;’, or a dot or colon at its end.
8. Translate the letters into their lowercase counterparts.

We have chosen to have a double new-line finishing a sentence since it might reduce the number of errors caused by using a dot inside a sentence. The quotation marks ‘"’ are used to delimit quotations—parts of the sentence which are to be specially treated. The literal quotation mark is entered

[5]The algorithm notation is explained in Appendix B.1.

by doubling it and putting it inside a quotation; e.g., ‘`"quote"""`’ is later translated to ‘`quote"`’.

Tokenization is the process of breaking the sentence into units called *tokens.* Tokens are usually words but they may be numbers, URL's, newsgroup names, e-mail addresses, etc. This is the process we usually call tokenization or lexical analysis in the parsing of formal languages. The preeditor makes sure that the input to this phase obeys certain rules, so it is closer to the input of formal language parsers.

In the context of formal languages, tokens are the final product of lexical analysis, but this is not the case here. In our case, the final product of lexical analysis is the *lexemes.* A lexeme is a sequence of one or more consecutive tokens. The lexemes, in contrast to the tokens, are ambiguous in the sense that they can overlap. If two or more lexemes overlap, then only one of them is correctly found but we do not know which one at this point—the syntactic analysis must make this decision.

Hence, tokenization can be seen as the process of finding appropriate *break points* in the sentence that are considered to be good starting points for later lexeme recognition. These points are numbered starting from zero—the zero point is at the beginning of the sentence and the last point is at the end of the sentence.

For example, if the sentence (the tokenization input) was:

`The␣red␣river␣flows␣into␣the␣sea.↵`

then the tokenization output would be

${}_0$The ${}_1$ red ${}_2$ river ${}_3$ flows ${}_4$ into ${}_5$ the ${}_6$ sea. ${}_7$

As in preediting, the tokenization routine is not generated by NL PAGE but is independently provided. Continuing the example of the QA-agent processing that we have started in Algorithm I, we can give a very simple example of tokenization: Put the break points at the beginning of the sentence, at the end of it, and at all space characters inside the sentence that are not inside a quotation.

Part-of-speech tagging is the process of finding the categories of the *lexemes*—the basic NL units in the linguistic sense—in the sentence. A lexeme is a part of a sentence that extends from one break point found in the tokenization to another, so its position in the sentence is determined by two numbers: its *starting* and its *ending point.* If we imagine lexemes as the blocks covering the sentence from its left to the right end, then at the end of the process the whole sentence has to be covered by a layer of lexemes—more precisely, every part of the sentence has to be covered by at least one layer of lexemes.

Each lexeme is associated with one *lexical category* such as noun (N), verb (V), adjective (ADJ), etc. The set of lexical categories is quite standardized. However, we find it appropriate to change this set arbitrarily to suit particular needs of the domain. For example, the non-standard categories URL and E-MAIL seem the most appropriate for lexemes such as '`http://www.com`' and '`who@where`'. As we will see in Section 4, a simple way to capture semantics is by using special categories. There is also a tradeoff between the usage of features and categories, which can lead to the introduction of some new categories.

Each lexeme is associated with a certain set of *features* and their values. For example, the pronoun 'I' has the feature *person* and its value is *first.* Now, using the fact that the verb 'sings' does not have the same value of the feature *person,* and using the appropriate grammatical rule, we can discard a sentence like 'I sings.'.

One of the specifics of our approach is the use of *binary features.* In this case, a feature value can be *true* (+) or *false* (−). This is a significant restriction, but considering computer efficiency with bit sets, we can expect a great improvement in terms of the algorithm time and space complexity. So, instead of the previous feature *person* we can have three binary features *p1*, *p2*, and *p3*; or, if we want to save on the number of features, we can have only two like *p1or2* and *p1or3.*

There is a global set of features, and for each lexeme and for each feature it is determined what the value of the feature is. It is not clear, for example, what the value of the feature *past tense* is for the word 'I', but this value is not important and can be any of the two values, since it will be ignored in the later processing, anyway.

Following the previous example we continue: If the input to the part of speech tagging were

$_0$ The $_1$ red $_2$ river $_3$ flows $_4$ into $_5$ the $_6$ sea. $_7$

then the output could be

$_0$ The $_1$ red $_2$ river $_3$ flows $_4$ into $_5$ the $_6$ sea. $_7$

THE	ADJ	N	V	PREP	THE	N
	N		N			

We can see one reason why the multiple lexeme layers are allowed: A word can have (and frequently does have) more than one possible lexical category. To see that two overlapping lexemes do not have to have the same starting and ending points (or *positions*), consider the following example:

$_0$ Common $_1$ Lisp $_2$ is $_3$ the $_4$ most $_5$ common $_6$ Lisp. $_7$

<table>
<tr><td>ADJ</td><td>NAME</td><td>BE</td><td>THE</td><td>MOST</td><td>ADJ</td><td>NAME</td></tr>
<tr><td colspan="2">NAME</td><td></td><td></td><td></td><td colspan="2">NAME</td></tr>
</table>

In the above examples the lexemes are illustrated in the form of a table. That table is called the *chart* and we have just started to build it—a process that will be continued and finished during the syntactic analysis. Each lexeme fills one *chart entry.*

We do the part-of-speech tagging by looking at *lexical entries* in the lexicon or by using some procedural means (such as regular expressions). Here is an example of a part-of-speech tagging algorithm:

Algorithm II: **Part-of-speech tagging**

Input: *sentence*
Output: *chart* partially filled

1. **For** each starting position $sp = 0$ **to** (the last position) $- 1$ **do**
2.   **For** each ending position $ep = sp + 1$ **to** the last position **do**
3.   **For** all lexical entries matching the sentence part from sp to ep **do**
4.   Add the lexical entry to the chart.
5.   **If** the string looks like a URL **then**
6.   Add a URL entry to the chart.
7.   **If** the string looks like an e-mail address **then**
8.   Add an EMAIL entry to the chart.
9.   **If** the string from sp to ep is a quotation or

```
        if nothing is found in the lexicon in this iteration then
            Break the inner loop.
If no entries are found starting from sp then
    Add a new entry to the chart starting from sp and ending at ep
    If the substring from sp to ep contains spaces then
        Let the entry have category NAMES.
    Else
        Let the entry have category NAME.
```

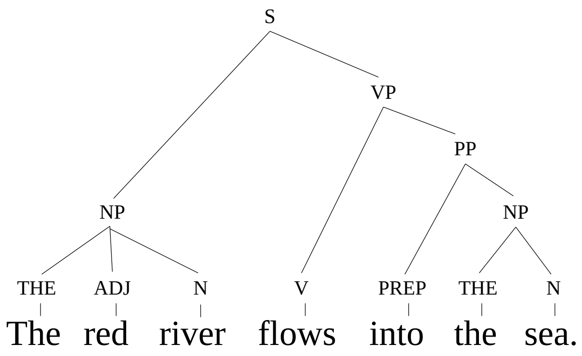

Figure 2.1: Parse tree

2.3 Syntactic analysis

The goal of syntactic analysis is to find the syntactic structure of the sentence. The structure has the form of a tree such as the one given in Figure 2.1.

2.3.1 Context-free grammars

The illustrated tree is a well-known structure in the context of context-free grammars.[6] It is called the *parse tree* of the sentence (also known as the *derivational tree*). We will not only use the term *parse tree* for complete sentence structures; the trees of the subsentence structures (e.g, the sub-tree with the root NP in Figure 2.1) are also called *parse trees.* The leaves of the tree in Figure 2.1 are marked by the word-category pairs. Actually, the leaves of our parse tree are the lexemes obtained from the lexical analysis, and are labelled with their lexical categories. The nodes of the tree that are not leaves are labelled by the *syntactic categories* such as NP, S, VP, etc. It is possible to have categories that are lexical and syntactic at the same time. We can always avoid this by constructing the grammar accordingly but there is no serious reason to do so. Furthermore, it provides more flexibility in the grammar construction. In the text that follows, it may be important to specify that a category is *exclusively syntactic* or *exclusively lexical,* and those are the terms that we will use for emphasis. The syntactic categories

[6] A good reference on the subject is [HU79].

are also called the *nonterminals, variables,* or *nonterminal symbols,* while the lexical categories are called the *terminals,* or *terminal symbols.* The categories of both kinds are called the *symbols.* We will see that in addition to a category label, more information can be associated with a symbol. In order to emphasize this, we will sometimes call a symbol a *node*, even though it might not be part of a parse tree.

The main part of a context-free grammar is the set of its *context-free rules.* From the sample tree in Figure 2.1 we can extract the following rules:

$$\begin{array}{ll} \mathrm{S} \rightarrow \mathrm{NP\ VP} & \mathrm{VP} \rightarrow \mathrm{V\ PP} \\ \mathrm{NP} \rightarrow \mathrm{THE\ ADJ\ N} & \mathrm{PP} \rightarrow \mathrm{PREP\ NP} \\ \mathrm{NP} \rightarrow \mathrm{THE\ N} & \end{array}$$

Each rule has a syntactic category on its left-hand side and a sequence of categories (the *constituents*) on the right-hand side. The rule $\mathrm{X} \rightarrow \mathrm{Y}_1\ \mathrm{Y}_2 \ldots$ is sometimes called the *rule for the nonterminal X.* A special rule, called the *empty rule* and denoted as $\mathrm{A} \rightarrow \varepsilon$ for arbitrary nonterminal A, is used to allow the "disappearance" of a symbol, when looking at the tree in a top-to-bottom direction. We will not allow the empty rules in our context-free grammar. Actually, our grammar for NLP is not precisely a context-free grammar since it has some additional elements, as we will see. We will call it the *NL grammar.*

The exclusion of the empty rule is the only restriction, in the context-free sense, that we put on the NL grammar. Various techniques for the restricted context-free grammars (such as LL(k) and LR(k) grammars) are developed for the purposes of formal language parsing. They require the non-ambiguity of the grammar in the sense that for any sentence there can be at most one correct parse tree. NL's are inherently ambiguous, so we cannot directly use those techniques.

2.3.2 Feature handling

We have introduced features in the previous section. Now, we will explain how they are handled and how they are used. As do lexical categories, the syntactic categories have sets of feature values that are determined as the parse tree is built.

One problem with the previous example of context-free grammar is that it would accept ungrammatical sentences such as:

*The red river *flow* into the sea.

A straightforward solution is, for example, to have a feature *p3sg* (third person, singular), and to propagate it up the tree in a "correct" fashion. Then, close to the root S, we would have the node NP specifying the feature value *p3sg*+, while the node VP would have the value *p3sg*−. If we insist that the categories NP and VP have the same value of the feature *p3sg* when the rule S → NP VP is applied, then the incorrect parse tree will not be obtained.

The *feature constraints* specify how we propagate the feature values up the tree and they add restrictions that must be satisfied if the appropriate node is to be added to the tree. We will build the tree in the bottom-up direction, so when adding a new node, the *child nodes* are known and the *parent node* is added to the tree. This operation is called the *reduction.*

There are two kinds of feature constraints:

1. *Absolute feature constraints*

   If a rule $X \rightarrow Y_1\ Y_2 \ldots Y_n$ is given, then we can prescribe any feature value of any category in the rule. If the category in question is the left-hand side category, then we are assigning the feature value to that category; otherwise, if it is one of the right-hand side categories, we are adding a constraint condition, since the feature values of those categories are already set. If the constraint is not satisfied, the reduction will fail.

   For example,
   1. NP → THE ADJ N $0(p3)+$
   2. NP → A ADJ N $3(n, sg)+$

   The first rule means that if the respective reduction is to be performed then the category NP is going to have the feature value $p3+$. (The number 0 is assigned to the left-hand side symbol and the right-hand symbols are enumerated starting from 1.) The second rule states that the rule can be applied only if the third constituent N has the feature values $n+$ and $sg+$.

2. *Equality feature constraints*

A feature equality states that two categories have to have the same value of a certain feature. If one of the categories is the left-hand side category, then the equality is actually a propagation rule. Otherwise, it is a constraint condition. For example, in
S $\rightarrow$ NP VP $\quad 1 \sim 2(p3)$,
the rule can be applied only if the constituents NP and VP have the same value of the feature *p3*. Or, the rule
VP $\rightarrow$ V PP $\quad 0 \sim 1(past, present)$
states that the values of the features *past* and *present* of the constituent V are propagated to the category VP.

One rule can have any number of the above constraints. All of them have to be satisfied for the reduction to be done. In this sense, the rules related to the right-hand side constituent have to be satisfiable. For example, the propagation is not just the propagation of a feature value—it is necessary to check if it conflicts with some other constraints. Obviously, we can also have contradictory rules, which are unsatisfiable. Those rules are removed by the parser generator. Besides checking for satisfiability, the generator also does some of the feature constraint optimizations. More about that will be considered in the next chapter.

2.3.3 Movement phenomena

Let us consider the following sentence:

What did you put the book in?

Its structure is related to the construct 'you did put the book in what', which is not a grammatical sentence but resembles a known structure of a declarative sentence. This is called a *movement phenomenon* since it can be visualized as a movement of certain components before the parse tree is constructed, see Figure 2.2. The components that are moved are called *fillers* and their final destinations are called *gaps.*

Of course, we cannot know how to move components in advance—it has to be done during the course of the tree construction. As we add new nodes to the tree, we also move constituents around according to certain rules.

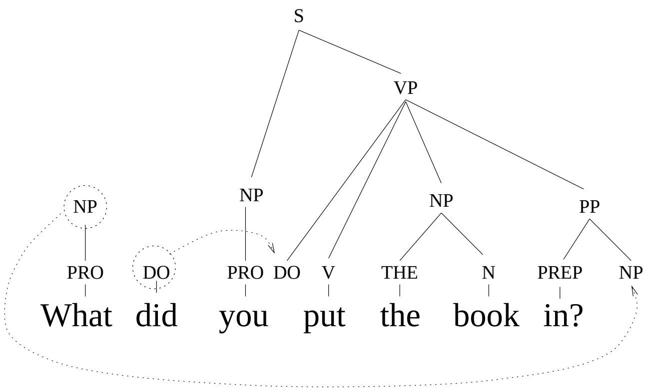

Figure 2.2: Movement in general

The movement can be treated as a special case of feature manipulation: The components are treated as a special kind of feature and feature unification is used to effectively move components. Since we use binary features, this is not a feasible solution, but a special mechanism needs to be added.

We will enrich the grammar rules so that each rule can provide a way of moving a constituent. For this to work efficiently, we have to put some limitations on what we can move and how to do it. However, we believe that the NL's can be successfully modelled within these limits.

There are two limitations: Only right-and-down movements are allowed, and a node can belong to no more than one movement path. To explain what this actually means, we will introduce a couple of terms using the previous example. The example in Figure 2.2 is handled in the fashion shown by the heavy arrows in Figure 2.3. If we redraw the tree in Figure 2.3 as in Figure 2.4 then it becomes clear what the *right-and-down* movement means. We allow movements where the filler is moved to the right to a sibling node and then passed down the tree. The restriction works, since the movements in NL are generally done in this way. It would be very unusual to have a movement to the left or up the tree. The horizontal movement to the right alone, without affecting the deeper levels of the tree, is not handled. It is usual in NL's and it can be easily handled by a special rule. This kind of movement belongs to a class called the *bounded movements,* since they are affecting a bounded number of surrounding nodes. The word "affecting" is not very precise here—we actually want to say that a movement is bounded

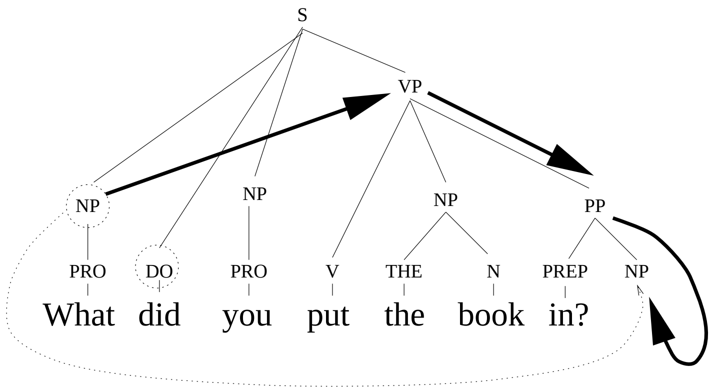

Figure 2.3: Movement in our model

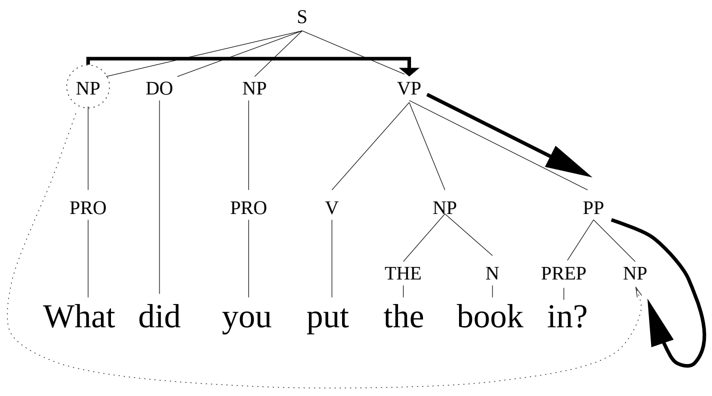

Figure 2.4: Movement to the *right* and *down* the tree

if the length of its movement path (soon to be defined) is bounded. The more important class is the *unbounded movements,* where a filler can pass through many nodes down the tree until it reaches the gap, and we want to make sure that they are handled properly. The movement of the node DO in the example is a bounded movement and it is not a problem to deal with it without explicitly moving the component.

Actually, the movement of the node DO is restricted by the previous movement and this is the second limitation in our approach. The *movement path* is the sequence of nodes that are passed by the filler until it reaches the gap (denoted by bold arrows in the figures). More precisely, the movement path consists of the filler node, the node which is both a sibling of the filler and an ancestor of the gap, and the nodes which are both descendants of this sibling-ancestor node and the ancestors of the gap. The gap is actually the same node as the filler, although their positions are different, and it is included in the movement path.

In our example, the movement path is NP-VP-PP-NP. All nodes in a path, except the filler, cannot participate in another movement path. Thus, we can have more movements in a parse tree, but their paths have to be disjoint (the fillers don't count). This is purely an implementation limitation—if really needed, the intersecting paths could be handled, but it is not the case.

The rules which handle movement have the following form:

$$\begin{array}{lll} \text{S} & \rightarrow \text{NP DO NP VP}\langle 1\rangle & (\textit{horizontal movement}) \\ \text{VP} & \rightarrow \text{V NP *PP} & (\textit{vertical movement}) \\ \text{PP} & \rightarrow \text{PREP @NP} & (\textit{final gap}) \end{array}$$

As we can see, the symbol '$\langle n\rangle$' denotes a horizontal movement where the filler is the nth component, the symbol '*' denotes a right-hand-side component that is involved in a vertical movement, and the symbol '@' is used to denote a final gap. We will frequently refer to the last two kinds of rule (involving the vertical movement and final gap) and we will call them the *gap-rules.*

It is usually desirable that these rules work without movement, too. For this reason the generator, normally, translates each of these rules into more

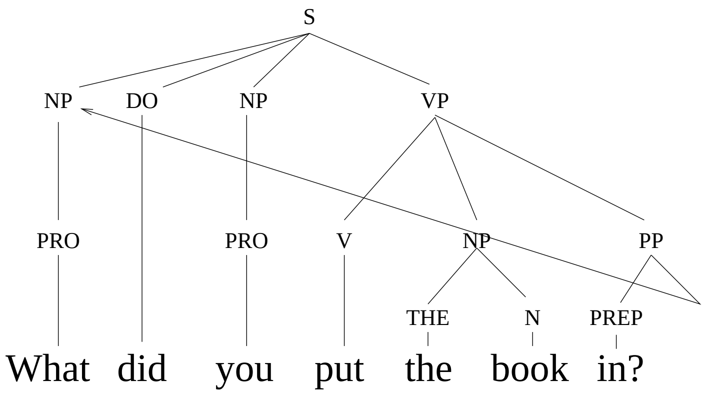

Figure 2.5: Actual parse "tree"

than one rule for code generation. However, the grammar maintainer has control over that process. We consider this issue in the next chapter.

In the actual implementation it is not whole structures that are moved, only the pointers referring to them. And, fillers are not moved at all. So, the actual parse tree will have a non-tree structure shown in Figure 2.5. If we do not track down where the pointers are really referring to, it may appear to us as a real tree, as shown in Figure 2.6.

2.3.4 Chart parsing

Up to now, we have described the NLP model in implicit, descriptive terms, giving from time to time hints about how the parsing is actually done. In the following two subsections the emphasis is on the parsing procedure.

The parsing algorithm is a variant of a well-known algorithm called *top-down chart parsing.*[7]

Let us recall what the result of the lexical analysis is: a partially filled chart. If we use the same example, then the final output of the syntactic analysis should be the chart filled as shown in Figure 2.7. The figure also shows the final parse tree which is implicitly contained in the chart.

Let us see how chart parsing works. The parsing algorithm is a top-down

[7]More about top-down chart parsing can be found in [All95].

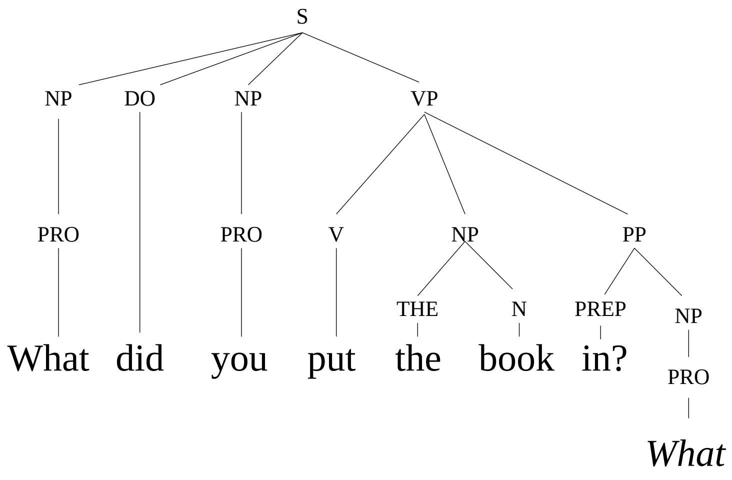

Figure 2.6: . . . and a tree again

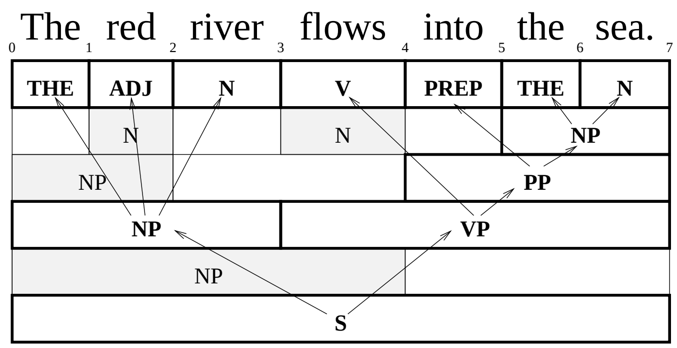

Figure 2.7: Chart and a parse tree in it

algorithm based on procedures for each syntactic category. If we did not have the chart, this would lead to the standard inefficient backtracking algorithm. Using the chart, all intermediate components that are found are stored and possibly reused in the later processing. In the figure, we can see some chart entries (the shaded ones) that are filled with categories but not used in the final tree.

The reuse of previously parsed nodes is the key idea in chart parsing. Moreover, the chart is a very appropriate structure for keeping the results of NL parsing. It can contain more than one parse tree. If a complete parse tree is not found, then it provides some partial parsing results, which are available also if complete parse trees are found. The parsing algorithm does not have to be set up to search only for the complete parse tree—we can use it to find as many partial components as possible.

Why would we need partial results? One obvious reason is that if a complete parse tree is not found then anything is probably better than nothing. But a better argument can be stated in the context of InIR. We know that the I-agents read documents in order to conclude something about their content; e.g., to do conceptual indexing, semantical matching, etc. For these purposes the partial parsing results are very useful. For example, if the parser has found a structure of a large noun phrase, then we know which word is the head of the phrase; knowing that, an agent can put more emphasis on that word and less on the other words in the phrase, and formulate a reasonable guess about semantic content. Or, as in [AAT97], we can make use of the noun phrase co-occurence hypothesis.

2.3.5 Algorithms

Although the algorithms are presented in detail, the tedious details having to do with efficiency considerations, such as bit-field manipulations, are excluded. If interested, the reader may find out more about them from the direct generator output.

Data structures The output of the lexical analysis is the sentence broken into tokens at the break points. The break points are numbered starting from

0 to last_position.[8] BreakPointEntry is an array whose elements are associated with the sentence break points. In addition to the information about the position of the break point, each element of this array contains the following fields:

- Found symbols (*found_symbols*)
  The information about the symbols found so far starting at this position. (The initial values are set in the lexical analysis.)

- Maximal nonterminal position (mp_X)
  For each nonterminal X there is a field mp_X that indicates that we have already tried to parse X from the current position to the maximal ending position mp_X. (The initial values are 0.)

Chart is a two-dimensional table consisting of last_position + 1 columns and a certain (large enough) number of rows. Each column corresponds to a break point. A chart entry is a structure that contains the information about a node. It contains the following fields:

- Category (*cat*)
  The category of the node.

- Ending position (*ep*)
  The ending position (break point) of the node. (The starting position is the column number within Chart.)

- Children (*ch*)
  An array of pointers to the children of the node. The lexical nodes do not have any children.

- Features (f)
  The set of feature values for the node.

- Movement slot (*gap*)
  The slot used to keep the pointer to the filler in the movement handling.

[8]The notation used for data structures and algorithms is described in Appendix B.

The hash table gap_parse_table is used to keep track of the movements in parsing. Using it, we can check for a given starting position *sp*, category *cat*, and a filler *fill,* whether we have already tried to parse the category *cat* from the position *sp* with the filler *fill*, up to which ending position, and whether it was successful.

It can be noted that we frequently refer to the maximal ending position. In particular, when we try to parse a nonterminal, it is done up to a certain maximal position, which is not always the end of the sentence. Since our grammar is an unconstrained context-free grammar, we need this mechanism to prevent infinite recursion (e.g., consider the rule NP $\rightarrow$ NP N). It also makes the algorithm more efficient. For preventing the infinite recursions caused by rules such as X $\rightarrow$ X we also use a sequence of global variables sp_X and mep_X defined for each nonterminal X. These keep track of the current starting and maximal ending position for parsing the nonterminal X. Their initial values are 0 and last_position + 1.

Parsing is done using the algorithms **parseN_X** and **parseG_X.** The algorithm **parseN_X** is defined for any nonterminal X for which there is at least one rule for X that is not a gap-rule (vertical movement or final gap). The algorithm **parseG_X** is defined for any nonterminal X for which there is at least one gap-rule.

Before presenting the algorithms, we have to keep in mind that the rules are "cooked" by the generator, so we do not have to worry about issues such as the feature constraint satisfaction problems in the feature propagation—we simply propagate the feature values.

Algorithm III: **parseN_X**

Input:	*sp*	the starting position
	mep	the maximal ending position
Output:	*ind*	flag indicating whether at least one parse has succeeded;

(*The global variables are always part of the input and output.*)

1. *ind* $\leftarrow$ No parses; *cpos* $\leftarrow$ *sp* (*cpos* — current position)
2. Set the feature values *cf* to the default values for the category X.
   (*cf* — the current feature values for the left-hand side category X)
3. **If** sp_X = *sp* **and** mep_X = *mep* **then**
4. | **Return** *ind* $\leftarrow$ No parses, since an infinite recursion loop is detected.

5. Save locally the values of sp_X and mep_X.
6. sp_X $\leftarrow$ sp; mep_X $\leftarrow$ mep
7. Parse according to the first rule X $\rightarrow \dots$
8. Parse according to the second rule X $\rightarrow \dots$
9. $\dots$
10. Restore the values from step 5 of sp_X and mep_X.
11. BreakPointEntry[sp].mp_X $\leftarrow$ mep
11. **If** ind = successful **then**
12. Indicate in BreakPointEntry[sp].$symbols\_found$ that at least one X is found.
13. **Return** ind

The rules in steps 7–9 are all non-gap-rules (i.e., not involving vertical movement nor final gap) of the nonterminal X. The following algorithm handles the gap-rules for X and it is quite similar to the previous one at the high level.

Algorithm IV: **parseG_X**

Input:	sp	the starting position
	mep	the maximal ending position
	$fill$	the filler node
Output:	ind	flag indicating whether at least one parse has succeeded;

(*The global variables are always part of the input and output.*)

1.–9. The same steps as in parseN_X.
10. Indicate the results of the parsing in gap_parse_table.
11. **Return** ind

The last two algorithms look very similar; however, more differences are hidden deeper in the "Parse according to the rule" steps. Parsing according to a certain rule is done in the following way:

Algorithm V: **Parse according to the rule $\mathbf{X} \rightarrow \mathbf{Y}_1 \dots \mathbf{Y}_n$**

(*The input and output are the global variables and the variables in context.*)

1. Save locally the value of cf.
2. Update cf according to the absolute feature constraints for the symbol X.

3. Process the component Y_1 and for appropriate nodes ch_1 do
4. | Process the component Y_2 and for appropriate nodes ch_2 do
5. | | $\ddots$
6. | | ... Try to add a new node at the starting position *sp*, having the ending position $ep = cpos$, $cat = X$, children ch_1, ch_2, $\ldots$, $f = cf$, and $gap = fill$ if appropriate.
7. | | $\cdot^{\cdot^{\cdot}}$
8. | | End of the loop for Y_2
9. | End of the loop for Y_1
10. Restore the value from step 1 of cf.

The algorithm for the loop interations in lines 3–9 and 4–8 (and so on) in the above algorithm varies depending on whether the component Y_i is a nonterminal or exclusive terminal, whether the parsing includes movement or not, and on the kind of movement involved. Hence, we derive the following four algorithms:

Algorithm VI: **Loop for an exclusive terminal Y_i without movement**

(*The input and output are the global variables and the variables in context.*)

1. **If** there are categories Y_i in the column *cpos* of Chart (the field *symbols_found* is used) **then**
2. | **For** all entries ch_i in the column *cpos* of Chart such that $ch_i.cat = Y_i$ **and** $ch_i.ep \leq mep - rcomp$ **and** the feature constraints are satisfied **do**
3. | | Save locally the values of cf and *cpos*.
4. | | Propagate features to cf according to the rule.
5. | | $cpos \leftarrow ch_i.ep$
6. | | ... (*body of the loop*)
7. | | Restore the values from step 3 of cf and *cpos*.

Lines 1–5 constitute the beginning of the loop, corresponding to lines 3 and 4 of Algorithm V, and line 7 is the end, corresponding to lines 8 and 9 of Algorithm V.

The condition $ch_i.ep \leq mep - rcomp$ includes a new value *rcomp*. The value is actually a constant—it denotes the number of components in the

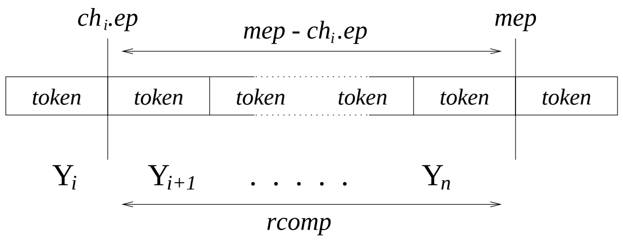

Figure 2.8: $mep - ch_i.ep \geq rcomp$

rule on the right side of Y_i that are not involved in any kind of movement (horizontal, vertical, or final gap). To understand this, we can recall that our grammar does not have any empty rules. If we look at Figure 2.8, we can see that the condition has to hold since we need a minimum amount of "space" for the components from the list $Y_{i+1} \ldots Y_n$ that are not involved in movement.

Algorithm VII: **Loop for a category $\mathbf{Y}_i$ that is a gap**

(*The input and output are the global variables and the variables in context.*)

1. $ch_i \leftarrow$ *fill*
2. **If** $ch_i.cat = Y_i$ **and** the feature constraints are satisfied **then**
3.   Save locally the value of cf.
4.   Propagate features to cf according to the rule.
5.   ... (*body of the "loop"*)
6.   Restore the value from step 3 of cf.

We see that in the last case our "loop" has at most one iteration.

Algorithm VIII: **Loop for a nonterminal $\mathbf{Y}_i$ without movement**

(*The input and output are the global variables and the variables in context.*)

1. **If** BreakPointEntry$[cpos].mp_Y_i < mep - rcomp$ **then**
2.   **parseN_$\mathbf{Y}_i$**$(cpos, mep - rcomp)$
3. **If** there are categories Y_i in the column $cpos$ of Chart **then**
4.   **For** all entries ch_i in the column $cpos$ of Chart such that
        $ch_i.cat = Y_i$ **and** $ch_i.ep \leq mep - rcomp$ **and**
        the feature constraints are satisfied **do**
5.   Save locally the values of cf and $cpos$.

6. Propagate features to *cf* according to the rule.
7. $cpos \leftarrow ch_i.ep$
8. ... (*body of the loop*)
9. Restore the values from step 5 of *cf* and *cpos*.

The following algorithm explains the processing of a nonterminal component involved in horizontal or vertical movement. The case when a nonterminal is a gap is included in Algorithm VII.

Algorithm IX: **Loop for a nonterminal $\mathbf{Y}_i$ involved in horizontal or vertical movement**

(*The input and output are the global variables and the variables in context.*)
1. Using gap_parse_table, find the maximal ending position *mep_gpt* up to which we have already parsed the category Y_i from the position *cpos* using the filler *fill*. If we have not done that kind of parsing then $mep\_gpt \leftarrow 0$.
2. **If** $mep\_gpt < mep - rcomp$ **then**
3. **parseG_Y**$_i(cpos, mep - rcomp, fill)$
4. **If** there are categories Y_i in the column *cpos* of Chart with the filler *fill* **then**
5. **For** all entries ch_i in the column *cpos* of Chart such that $ch_i.cat = \mathrm{Y}_i$ **and** $ch_i.ep \leq mep - rcomp$ **and** $ch_i.gap = fill$ **and** the feature constraints are satisfied **do**
6. Save locally the values of *cf* and *cpos*.
7. Propagate features to *cf* according to the rule.
8. $cpos \leftarrow ch_i.ep$
9. ... (*body of the loop*)
10. Restore the values from step 6 of *cf* and *cpos*.

Finally, here is the algorithm for step 6 in Algorithm V—adding a new node to the table. Considering Algorithm V, we notice that the line reads: "Try to add..." Even though we passed the whole way in constructing a node, at the end we might not add it to Chart because of a condition that indicates that the node is already in Chart. This may sound like a symptom of inefficiency, but if we analyze it in more detail, we easily see that it is not the case and the other strategy yields a less efficient algorithm, in practice. At the core of this issue lies the use of the maximal ending position.

Algorithm X: **Try to add a node not involving movement (*gap* is empty)**

(*The input and output are the global variables and the variables in context.*)

1. **If** $cpos >$ BreakPointEntry$[sp].mp_X$ **then**
2. Add new entry to Chart in the column sp having the field values $ep \leftarrow cpos$, $cat \leftarrow$ X, $ch \leftarrow (ch_1, ch_2, \ldots)$, and $f \leftarrow cf$.
3. $ind \leftarrow$ successful

Algorithm XI: **Try to add a node involving movement (*gap* non-empty)**

(*The input and output are the global variables and the variables in context.*)

1. Using gap_parse_table, find out the maximal ending position mep_gpt up to which we have already parsed the category X from the position sp using the filler *fill*. If we haven't done that kind of parsing then $mep\_gpt \leftarrow 0$.
2. **If** $cpos > mep\_gpt$ **then**
3. Add new entry to Chart in the column sp having the field values $ep \leftarrow cpos$, $cat \leftarrow$ X, $ch \leftarrow (ch_1, ch_2, \ldots)$, $f \leftarrow cf$, and $gap \leftarrow fill$.
4. $ind \leftarrow$ successful

2.4 Higher levels of NLP, or how to use the parser

Semantic and discourse analysis are not handled by the parser and they are not included in our model. We will present some hints of how they can be performed using the described underlying model.

The main product of the syntactic analysis is the chart—more precisely, it is the set of filled entries in the chart. The most usual way of using the parser is to perform the procedure **parseN_S**(0,last_position) and after the analysis is finished to look for the root of the parse tree in the first column of the chart that has the ending position equal to last_position and category S. In the best situation, there will be exactly one such parse tree and we can then use various techniques to use it in the higher levels of NLP. If we get more parse trees, we can try to choose one. One way to do this is to use stochastic methods that can be applied after the syntactic analysis is finished choosing the tree with the highest probability. If no parse trees are found, then we can try to use the partial parsing results. Another option is to change sentence boundaries and to rerun the parser. A good thing about this approach is that if we have not changed the global variables, the parser will be able to reuse the information contained in the chart and get new parses faster. Rerunning the parser can be used in searching for the syntactic components in an NL text—we can go backward and forward starting various **parseN_X** procedures without a danger of getting repeated structures or overflowing the memory, i.e. the chart. This can be a base for some robust NLP techniques.

This clear separation between syntax analysis and the higher levels of NLP has some disadvantages. The semantic and even discourse knowledge can help us a lot in handling syntactic disambiguation. However, we can still accomplish this within our approach by using categories and features that are based on semantics, although syntactically used. For example, if we are building a QA-agent for InIR and we recognize that the user will frequently use queries such as "Find me…," "Retrieve…," "I am interested in…," "Give me…," etc. then we can add the lexemes '`find␣me`', '`retrieve`', '`i␣am␣interested␣in`', etc. to the lexicon under a new category RETRIEVE. Now, we can have a rule such as S $\rightarrow$ RETRIEVE NP. It is obvious that the category RETRIEVE has a semantic connotation, but the parser does not make any distinction. The lexeme '`i␣am␣interested␣in`' seems especially non-standard.

Having this example, we can illustrate another interesting situation: Let us say that we have developed a grammar without the rule S $\rightarrow$ RETRIEVE NP that can handle a sentence like 'I am interested in boats.'. Then, we decide to introduce the rule with the lexeme RETRIEVE. The parser will obviously generate at least two complete parse trees for the sentence. So, we add a new rule to make handling a specific type of sentence easier, and end up having always at least two parse trees for exactly those sentences. A way out is to give a priority to the parse trees that are based on, or which simply contain, categories such as RETRIEVE—we may call them *keyword categories.*

We have mentioned before that a problem with the I-agents is that they can encounter very noisy texts where they cannot determine the sentence boundaries. A solution in such cases could be the following robust algorithm that largely relies on the concept of partial parsing:

Algorithm XII: **An example of the I-agent's NLP**

Input: *input_stream*
Output: useful knowledge

1. **While** *input_stream* not at the end **do**
2. Read further and do preediting. If the end of a sentence cannot be found then stop at an appropriate point (e.g., a space) at a sufficient distance from the starting point and let it be the end of the sentence.
3. Do the tokenization and part-of-speech tagging where not done already.
4. Call sequentially the procedures parseN_X_1, parseN_X_2,…, parseN_X_n for a list of nonterminals X_1, X_2,…, X_n determined in advance.
5. Search the first column of the chart for the categories X_1, X_2,…, X_n and pick the one with the maximal ending position. If more than one of them has the same ending position which is maximal, choose according to a pre-specified criterion.
6. Depending on the category in question, extract some useful knowledge from the chosen structure.
7. Move in the *input_stream* to the ending position of the chosen structure.

Chapter 3

NL PAGE: Parser Generator System

In the previous chapter we have discussed our NLP model and its parsing strategy. It is simplified in order to be efficient and directly translatable to low-level machine operations, but it still provides enough expressiveness to capture the main types of NL constructions.

The approach leads to faster running time and smaller code size in accordance with our goals. But from the user's side it is very unsuitable. It is not hard to code the algorithm, but having to do that for many nonterminals and many rules would be very tedious and prone to errors. It should not be considered for any serious application.

This argument should convince us that we really need a parser generator. Moreover, it is not only that we want to be more convincing in this case but also because we wish to argue that NL parsing consists of two problems that should be treated separately, at least in our InIR context. We have two problems: how should we design an efficient parsing technique applicable to NL's, and how should we create and maintain an NL grammar in a previously defined model?

Once again, let us compare the processing of NL's and formal languages. Although we need parser generators, such as lex and yacc, for formal languages, the problem of creating a parser for a formal language can be managed manually. We would have to plan and write down the entire grammar and lexical rules, and do the coding, and, eventually, we will get a parser. However, this is not a feasible approach for natural languages. It is better even not to try to write a grammar that will capture a whole NL. We can capture only one part of an NL and then we need a lot of experimenting—write a grammar, try the parser, rewrite the grammar, retry the parser, and so on. The creation of an NL grammar is more like a free-style artistic activity.

As stated earlier, our MIN approach may need many different NL parsers and it is good to have a tool to produce them easily. We need at least two parsers—one for a QA-agent and the other for an I-agent. If we want to have more I-agents specialized in certain domains, then we accordingly need more parsers. We can have specialized QA-agents, too.

If we want to deal with various natural languages in the InI-space then we need more I-agents that are specialized in certain languages. Our NLP model is NL-independent, and so is the parser generator. The needed condition is that the NL in question has to be specified by a context-free grammar enriched with features and movement handling. Most natural languages, or at the least European languages, are like that.

Another desirable feature of the generator is support of more target programming languages. The current implementation supports two languages—C and Java. The C language is suitable since it is very portable and appropriate for our efficiency considerations. Static agents running on a machine as server processes could have parsers developed in C. We can use C in Java applications by adding the NLP capability in the form of native code methods, but we will lose the Java portability. Java is an especially desirable target language in our context of Internet agents. It is very portable, although less efficient than C. A typical use of a Java parser is within an applet QA-agent. In particular, we can have a MAS for InIR with the user interface embedded in an HTML page as an applet QA-agent. If the applets themselves are capable of NLP, then it would greatly reduce the server load. When a user types an NL query, the applet can process it, the CPU (Central Processing Unit) time of the user machine would be spent, and the agents on the server machines would receive a message in an easily manageable format.

The parser generator could be used by the agents themselves in an interesting way to improve their NLP capabilities. They could rebuild their own

NLP modules after learning some new NL rules and lexemes.

Section 3.1 of this chapter presents the structure of the parser generator. Section 3.2 is about its input—the parse forest. Section 3.3 explains the parse forest translation to the intermediate representation. Section 3.4 describes the intermediate representation and the code generation.

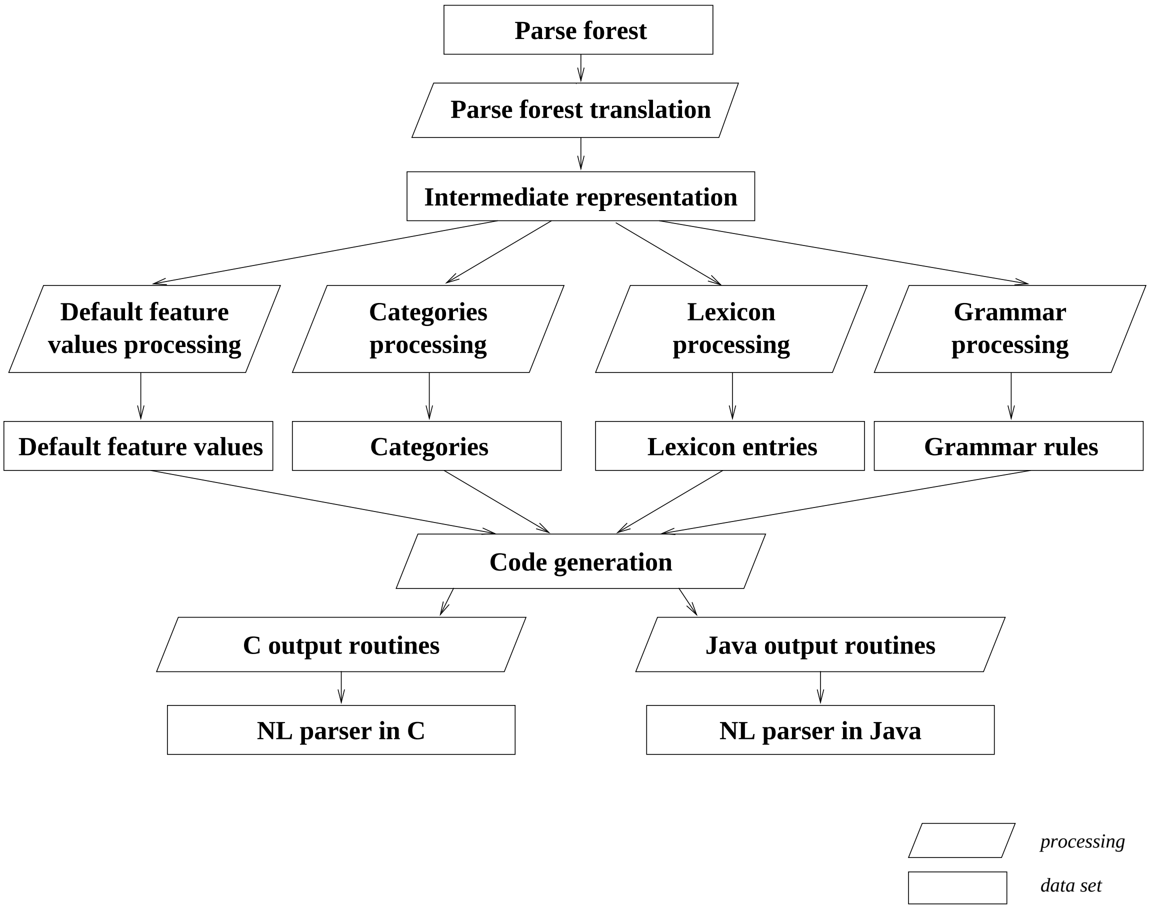

Figure 3.1: Parser generator flowchart

3.1 Parser generator structure

The structure of the parser generator is shown as the flowchart in Figure 3.1. The input to the generator has the form of an NL parse forest. The user[1] inputs the rules, lexical entries, and some other information in the form of a list of parse trees; i.e., a *parse forest.* A special parse forest notation, discussed in the next section, is designed for this purpose. The parse forest is translated

[1]Within this chapter the user is the designer of the grammar; i.e., the term denotes the user of the parser generator.

into an intermediate representation. The translator is developed in C using the standard tools lex and yacc. It is a complex process of extracting atomic items from the parse forest and, after doing certain transformations, outputting them in a new, intermediate format. These transformations include simplification of the syntax rules, checking the constraint satisfiability, the rule optimization, and the removal of duplicate and other redundant rules. These "other redundant" rules will be defined later. The result of the parse forest translation is a set of the three kinds of *atomic items*: the default feature value information (*dfv-items* for short), the lexical entries, and the syntax rules. Each of them is represented in the form of a line in the intermediate representation format. The lines are sorted and prepared for further processing.

In the next phase, the atomic items are processed in parallel by four Perl [Mol97] scripts: the first one extracts the dfv-items, the second extracts the symbols, the third extracts the lexical entries, and the fourth extracts the syntax rules. The results of this phase are fed into the code generator.

Finally, the generator builds the parser using a package of output routines for a target language. There are two packages currently implemented: one is for C and the other one is for Java. The generator is written in Perl.

3.2 The parse forest

A set of parse trees, called the *parse forest,* is the form of input the creator of an NL grammar has to provide to the generator. It has a flexible format that provides various ways to state the syntax rules, to specify the lexemes, and to define the default feature values for categories. The user (creator of the NL grammar) is not required to list the terminals, nonterminals, or features. Many other constants such as the maximal number of children a node can have, which are needed for efficient parsing but would be just a burden for the user, are automatically extracted by the generator. The behavior of the generator is precisely defined and knowing it we can easily adjust the parse forest, gradually converging to a parser that will suit our needs.

3.2.1 Examples

Before giving the exact syntax rules for the forest (they are actually called the *meta-syntax rules* to avoid confusion with the *NL rules*), let us show some example excerpts from the parse forest.

The parse forest can contain comments which are ignored by the parser but help us to document the forest. There are three types of comments:

1. One-line comments start with the character '`#`' at the beginning of the line such as:

   ```
   # Comment line
   ```

2. Multiple-line comments (or exclusion of a part of the parse forest) use the keywords '`\beginComment`' and '`\endComment`' at the beginning of a line:

   ```
   \beginComment
   Comment lines
   ...
   \endComment
   ```

3. The third type of comment is the use of the keyword '`\end`' at the beginning of a line. It is a way to exclude the rest of the forest:

```
\end
```
...(*The rest of the forest is ignored.*)

The parse trees are entered using a standard notation with parentheses. For example, the parse tree

```
(S (NP(PRO He)) (VP (MOD can) (V read)))
```

is going to be translated into three lexical entries: ('he', PRO) ("he" having the category PRO), ('can', MOD), and ('read', V); and the three NL rules S $\rightarrow$ NP VP, NP $\rightarrow$ PRO, and VP $\rightarrow$ MOD V. We don't have to write the complete parse trees of a sentence. For example, the following parse trees

```
(S (NP) (VP))    (PRO he)
```

would define the rule S $\rightarrow$ NP VP and the lexical entry ('he', PRO), just as well.

If a lexical entry contains characters other then letters, then it has to be put in quotes.[2] For example, the lexical entry ('World Wide Web', NAME) is entered as follows:

```
(NAME "World Wide Web")
```

In this way we can enter any string of characters and call it a lexeme. The literal double-quote character itself is entered by doubling it (as suggested in the discussion about lexical analysis). Since we don't make a clear distinction between lexical and syntactic categories, we can use nonterminals such as NP as a lexical category. For example,

```
(S (VP "Enter Keyword(s)") (TO to) (VP Search))
```

will define two lexical entries 'Enter Keyword(s)' and 'search' with the category VP. This will not confuse the parser if it is already using VP as a nonterminal category. Sometimes, we do not want to analyze more deeply the parse tree of a sentence, but we also do not want these long unparsed lexemes to be added to the lexicon; and still, it is good for documentation purposes to have the complete sentence in the tree. In this case we can use the following kind of comment:

[2] Actually, it can contain the underline character and digits at any position except the first, but it is never a mistake to use the quotes.

```
(S (VP % Enter keywords) (TO to) (VP % search))
```

Now, only the rule S $\rightarrow$ VP TO VP and the lexical entry ('to', TO) are defined.

Features Let us show the notation for the feature constraints by an example that involves all four types of constraint (we consider absolute and equality constraints, and for each of these both the propagation rule and the "proper" constraint case). Let us say that in the sentence 'The small computer works.', which has the following structure:

```
(S (NP (THE the) (ADJ small) (N computer)) (VP (V works)))
```

we want to add some feature constraints so that the following rules are defined:

$$
\begin{array}{rlll}
\mathrm{S} & \rightarrow \mathrm{NP\ VP} & 1(nom)+ & 1 \sim 2(p3) \\
\mathrm{NP} & \rightarrow \mathrm{THE\ ADJ\ N} & 0(p1,p2)- & 0(p3)+ \\
\mathrm{VP} & \rightarrow \mathrm{V} & 0 \sim 1(p3) &
\end{array}
$$

Then, the parse tree for the sentence would be written in the following way:

```
(S >nom(NP.p1-.p2-.p3+ (THE the) (ADJ small) (N computer))
   >=1p3(VP >^p3(V works)))
```

or

```
(S >nom(NP.p1-.p2-.p3+ (THE the) (ADJ small) (N computer))
   >=1p3(VP >=0p3(V works)))
```

We also need to assign the feature values for the lexemes. It is done in the same fashion, for example:

```
(N.p3.sg.p1- computer)
```

If the number of features is large, then it can be tedious to list all the feature values for lexemes or symbols in a parse tree. For this reason we can define the sets of default feature values for any symbol, so if a feature of a symbol is not mentioned in a tree, then the default value is assumed.

The default feature values are defined using the keyword '`\default:`' at the beginning of a line; for example

```
\default:  (PRO.p3sg-.p1-.p2-.sg.nom.gen-)
```

defines the default feature values for the symbol PRO. We note that the sign + is assumed if no sign is given.

Writing the feature constraints can be laborious and we often want to experiment with "clean" trees (no features) without bothering to consider the features until it is necessary to do so. The problem is that if we have a rule that involves the feature constraints and we repeat it somewhere else without the feature constraints then the new rule may accept some ungrammatical constructions, while the first rule will be simply ignored. A solution is found in the following way: If we have two rules or two lexical entries such that they are the same except that one has certain feature constraints and the other one does not have any, then the rule or lexical entry without feature constraints is called *inferior* and it is discarded by the generator. Let us emphasize once more that a rule or lexical entry can be inferior only if there is a corresponding rule containing feature constraints.

Movement phenomena The movement phenomena are handled using two reserved feature names *gap* and *fgap*. We cannot have user-defined features with these names. These two words are used syntacticaly in the same way as features are used, but they are *not* features. Instead, the generator will transform them into the movement rules. Let us show how they are utilized with the example from subsection 2.3.3 with the sentence 'What did you put the book in?'. The sentence was parsed as follows

```
(S (NP (PRO What)) (DO did) (NP (PRO you))
   (VP (V put) (NP (THE the) (N book)) (PP (PREP in) (NP))))
```

assuming that the following movement rules were applied:

$$\begin{array}{rll} \mathrm{S} & \rightarrow \mathrm{NP\ DO\ NP\ VP}\langle 1\rangle & (\textit{horizontal movement}) \\ \mathrm{VP} & \rightarrow \mathrm{V\ NP\ {*}PP} & (\textit{vertical movement}) \\ \mathrm{PP} & \rightarrow \mathrm{PREP\ @NP} & (\textit{final gap}) \end{array}$$

In our notation, the movement rules are added to the parse tree in the following way:

```
(S (NP (PRO What)) (DO did) (NP (PRO you))
   >=1gap.gap(VP (V put) (NP (THE the) (N book))
         >^gap(PP (PREP in) >fgap(NP))))
```

Any rule that includes a movement will normally hold when no movement occurs. From the previous example, the rule PP $\rightarrow$ PREP @NP should also work as the rule PP $\rightarrow$ PREP NP when the field *gap* of the node PP is empty. The generator will produce more rules for the final code generation. Sometimes, however, we do not want that and this is facilitated by the feature-like usage of symbol *gap*. By setting the value of the "feature" *gap,* we decide whether the corresponding component has to be involved[3] in movement (*gap*+) or not (*gap*−). If we take a look at the previous example, we can see that in the rule S $\rightarrow$ NP DO NP VP$\langle 1 \rangle$ we force horizontal movement, i.e. the rule cannot be applied unless the first component NP is moved to the last component VP (and down the tree). The string '`=1gap`' denotes the horizontal movement, and the string '`.gap`' denotes that the movement is required.

3.2.2 Formal specification

In this subsection we will define precisely the format of the parse forest. The description is divided into two parts—the first part is about the lexical structure of the parse forest and the second is about its syntactic structure. The use of these meta-lexical and meta-syntax rules should not be confused with the lexical entries and rules concerning the natural language. In this case we are only dealing with a formal language. To prevent confusion, we will sometimes emphasize that a term in question is a *meta*-term or an *NL* term depending on whether it is a part of the formal specification of the parse forest, or a part of the NL grammar that is defined by the parse forest.

[3]Precisely, a node is involved in movement if it is a part of a movement path and it is not the filler nor the gap in it.

Meta-lexical rules The meta-lexical rules are defined using regular expressions. At times, the rules might seem to be ambiguous, but this is not the case, since we adopt the convention that the longest matching meta-lexeme (token) is chosen, and if there are others then the one matching the first specification in the following list is chosen. The rules are used as the input to the program lex, which produces the lexical analyser.

1. The invisible characters ␣ (space), ⇥ (tab), and ↵ (new-line) are accepted and no action is taken, i.e. they are ignored. Their only possible function is to separate other tokens.

2. The characters `(`, `)`, `^`, `.`, `>`, `=`, `+`, and `-` are recognized as separate lexemes.

3. The regular expression $[\mathtt{a{-}zA{-}Z\_}][\mathtt{a{-}zA{-}Z0{-}9\_}]^*$ specifies the lexeme WORD.

4. NUMBER is represented by $[\mathtt{0{-}9}]^+$.

5. STRING is represented by $\left(\mathtt{"}\overline{[\mathtt{"}]}^*\mathtt{"}\right)^+$. As we can see, a string is delimited by double-quote characters. Double-quote characters can occur inside a string and they have to be doubled in order to avoid ambiguity.

   E.g. `"string␣""abc"""` is a valid string. It is transformed internally into `string␣"abc"` after being recognized.

   Tabs and new-lines found in a STRING are transformed into spaces.

6. DEFAULT is a keyword token recognized using the regular pattern $\vdash$`\default:`.[4]

   Certain parts of the parse forest can be excluded from processing (commented out). These parts are ignored during the lexical processing. We have seen in the previous examples that we have four ways to do this:

7. By the string `\end` at the beginning of a line. A comment of this type is recognized using the regular pattern $\vdash$`\end`$\overline{[\,]}^*$.

[4]The notation used for regular expressions and patterns is explained in Appendix B.

8. By enclosing the comment between the strings `\beginComment` and `\endComment`. More precisely, this type of comment is recognized by the pattern $\vdash\texttt{\textbackslash beginComment}\overline{[\hookleftarrow]}^{*}\hookleftarrow\left(\overline{[\backslash]}\,\overline{[\hookleftarrow]}^{*}\hookleftarrow\right)^{*}\texttt{\textbackslash endComment}$.

9. By starting a line with `#`. This is a typical line comment, which is matched by $\vdash\texttt{\#}\overline{[\hookleftarrow]}^{*}\hookleftarrow$.

10. Using the character `%`. The comment finishes at the end of the line or at the first right paranthesis); i.e., it is recognized by the regular expression $\texttt{\%}\overline{[)\hookleftarrow]}^{*}$.

Meta-syntax rules The meta-syntax rules form an LR(1) grammar. The tool yacc is used to build the parser for the parse forest translation. The list of rules with explanations follow:

1. *forest* $\rightarrow \varepsilon$
   forest $\rightarrow$ *forest* NUMBER *tree*
   forest $\rightarrow$ *forest tree*
   forest $\rightarrow$ *forest* DEFAULT (WORD *featurelist*)

   The meta-symbol *forest* is the start symbol. As we can see, the parse forest is a list of parse trees that are matched by the symbol *tree*. The number that can optionally precede a tree can be later used to identify the tree from which an NL rule is derived. The last rule describes the notation for the definition of the dfv-items.

2. *tree* $\rightarrow$ (WORD *featurelist ftree ftrees*)
   tree $\rightarrow$ (WORD *featurelist* WORD)
   tree $\rightarrow$ (WORD *featurelist* STRING)
   tree $\rightarrow$ (WORD)

   These are the rules that describe the meta-syntax of a parse tree. The first rule defines the meta-syntax of an NL syntax rule, the next two rules define the meta-syntax of the lexical entries, and the last rule describes the empty tree such as '`(NP)`'.

3. *ftree* $\rightarrow$ *tree*
   ftree $\rightarrow$ `>` *featurelist1 tree*

The meta-nonterminal *ftree* represents a *tree with features.* This type of tree is always seen as a subtree. A tree with features is actually a tree that can optionally be preceded by a list of feature constraints (*featurelist1*) that are part of the surrounding rule.

There is also the meta-nonterminal *featurelist*, which is also a list of feature constraints but placed on a different point and subject to different rules.

4. *ftrees* $\rightarrow \varepsilon$
   ftrees $\rightarrow$ *ftrees ftree*

   The meta-nonterminal *ftrees* represents a list (possibly empty) of ftrees, i.e. trees with features. This is a right-recursive list definition. It is used to specify the list of rule components in an NL node.

5. *featurelist* $\rightarrow \varepsilon$
   featurelist $\rightarrow$ *featurelist* `.` WORD
   featurelist $\rightarrow$ *featurelist* `.` WORD `+`
   featurelist $\rightarrow$ *featurelist* `.` WORD `-`

   The meta-nonterminal *featurelist* is the list of absolute feature constraints related to the left-hand side nonterminal of an NL rule or to a lexical entry, e.g., in `(NP.p3 (N))` and `(PRO.p1 we)` the parts '`.p3`' and '`.p1`' are examples of the *featurelist*'s.

6. *featurelist1* $\rightarrow \varepsilon$
   featurelist1 $\rightarrow$ WORD `+`
   featurelist1 $\rightarrow$ WORD `-`
   featurelist1 $\rightarrow$ *featurelist1* `.` WORD
   featurelist1 $\rightarrow$ *featurelist1* `.` WORD `+`
   featurelist1 $\rightarrow$ *featurelist1* `.` WORD `-`
   featurelist1 $\rightarrow$ *featurelist1* `^` WORD
   featurelist1 $\rightarrow$ *featurelist1* `=` NUMBER WORD

   This is the second type of list of the feature constraints. These constraints are related to the right-hand side components of the NL rules, like '`nom =2p1`' in `(S >nom =2p1(NP) (VP))`.

3.3 Parse forest translation

Most of the processing in the parser generation is done in this module. It bridges the gap between the user input forest, which can be written in various ways, and the format needed by the code generation routines, which has to be optimized and checked for consistency if we want to obtain a correct and efficient parser.

The translation process is a mapping of the nodes in the parse forest into the atomic items—the syntax rules of the NL grammar, the lexical entries, and the dfv-items. Whenever the translator reads the character ')' (if it is not in a quotation or a comment), marking the end of a node, an atomic item is extracted.

Data structure During the translation, the following data structure is used to keep track of the known information about an item:

- Left-hand side category (*lrule*)
  Used to keep the left-hand side category of a rule, the category of a lexical entry, or the category of a dfv-item. (Example: `X`)
- Right-hand side components (*rrule*)
  This is the list of right-hand side categories of a rule. In the case of a lexical entry or a dfv-item this field is not used. (Example: `Y␣Z`)
- Positive absolute feature constraints (*setT*)
  This is the list of lists of positive (+) absolute feature constraints. In the case of a lexical entry or a dfv-item, there can be at most one list in the field. (Example: `0(a,b)␣1(a,c)...`)
- Negative absolute feature constraints (*setF*)
  This is the list of lists of negative (−) absolute feature constraints. In the case of a lexical entry or a dfv-item there can be at most one list in the field. (Example: `0(d)-␣3(c)-...`)
- Equality feature constraints (*equal*)
  This is the list of lists of equality feature constraints. The field is not used in the case of a lexical entry or a dfv-item. (Example: `1=2(d)␣...`)

- Horizontal movement information (*htgap*)
  This is the list of horizontal movements. The field is not used in the case of a lexical entry or a dfv-item. (Example: `1=>2␣...`)

- Vertical movement information (*vtgap*)
  This is the list of component numbers that vertically receive a filler from the parent node. Usually, there is no more than one number in this list. The field is not used in the case of a lexical entry or a dfv-item. (Example: `2␣3...`)

- Final gap information (*fgap*)
  Similar to the previous field, this is a list of component numbers that are final gaps if the parent node participates in a movement path. The field is not used in lexical entry or dfv-item processing. Usually, we do not have more than one final gap in a rule. (Example: `3␣...`)

Algorithms When the information about an item is collected in the above structure, before outputting the item, the translator performs satisfiability checking and optimization. The following algorithm is performed when dealing with a lexical entry or a dfv-item:

Algorithm XIII: **Check lexical entry or dfv-item**

Input: *item data structure*
Output: error *or* OK Is the item satisfiable?
(*the structure can be changed*)

1. Remove duplicate features in *setT* and *setF*
2. **For** each feature *f* in *setT* **do**
3. | **If** *f* contained in *setF* **then**
4. | | **Return** error
5. **For** each feature *f* in *setF* **do**
6. | **If** *f* contained in *setT* **then**
7. | | **Return** error
8. **Return** OK

If the consistency check is passed, an item entry is generated at the output.

When a rule is generated, the satisfiability conditions get more complicated. As we can see from the following algorithm, when a rule is prepared the translator can output up to eight rules for the code generation.

Algorithm XIV: **Check and output the rule**

Input:	r	the rule data structure
Output:	error *or* OK	Is the rule satisfiable?
	(*up to eight output rules*)	

1. Check the general rule satisfiability of the rule r.
2. **If** test failed **then Return** error
3. $r1 \leftarrow r$; Remove *r1.htgap*, *r1.vtgap*, and *r1.fgap*
4. Check the specific rule satisfiability of *r1*
5. **If** test passed **then** Output *r1*
6. **If** *r.htgap* is non-empty **then**
7. | $r1 \leftarrow r$; Remove *r1.vtgap* and *r1.fgap*
8. | Check the specific rule satisfiability of *r1*
9. | **If** test passed **then** Output *r1*
10. **If** *r.vtgap* is non-empty **then**
11. | $r1 \leftarrow r$; Remove *r1.htgap* and *r1.fgap*
12. | Check the specific rule satisfiability of *r1*
13. | **If** test passed **then** Output *r1*
14. **If** *r.vtgap* is non-empty **and** *r.htgap* is non-empty **then**
15. | $r1 \leftarrow r$; Remove *r1.fgap*
16. | Check the specific rule satisfiability of *r1*
17. | **If** test passed **then** Output *r1*
18. **If** *r.fgap* is non-empty **then**
19. | $r1 \leftarrow r$; Remove *r1.htgap* and *r1.vtgap*
20. | Check the specific rule satisfiability of *r1*
21. | **If** test passed **then** Output *r1*
22. **If** *r.htgap* is non-empty **and** *r.fgap* is non-empty **then**
23. | $r1 \leftarrow r$; Remove *r1.vtgap*
24. | Check the specific rule satisfiability of *r1*
25. | **If** test passed **then** Output *r1*
26. **If** *r.vtgap* is non-empty **and** *r.fgap* is non-empty **then**
27. | $r1 \leftarrow r$; Remove *r1.htgap*
28. | Check the specific rule satisfiability of *r1*
29. | **If** test passed **then** Output *r1*

30. **If** $r.htgap$, $r.vtgap$, **and** $r.fgap$ are non-empty **then**
31. | Check the specific rule satisfiability of r
33. | **If** test passed **then** Output r
34. **If** no rules are output **then Return** error
35. **Else Return** OK

Algorithm XV: **Check the general satisfiability of a rule**

Input:	r	the rule
Output:	error *or* OK	Is the rule satisfiable?
	(*the rule can be changed*)	

1. Check that the component numbers are in the range $0 \ldots$ *number_of_components* and do not refer to the component they are associated with.
2. **If** check failed **then Return** error
3. Check for contradictory *htgap*'s (e.g., `1=>4 2=>4`)
4. **If** a contradiction is found **then Return** error
5. Simplify the rule by applying the equalities (e.g. $1 \sim 2(a)$ and $1(a)-$ implies $2(a)-$).
6. Check for conflicting absolute feature constraints (e.g., $2(a)$ and $2(a)-$ is a contradiction).
7. **If** a contradiction is found **then Return** error
8. Transform the equalities of the form $0 \sim i(f)$ and $0 \sim j(f)$ to $0 \sim i(f)$ and $i \sim j(f)$. (*needed in later optimization*)
9. **Return** OK

Algorithm XVI: **Check the specific satisfiability of a rule**

Input:	r	the rule
Output:	error *or* OK	Is the rule satisfiable?
	(*the rule can be changed*)	

1. Check for conflicts *htgap* vs. *vtgap* (e.g., `1=>4` and *vtgap*:`4`).
2. Check for conflicts *htgap* vs. *fgap*.
3. Check for conflicts *vtgap* vs. *fgap*.
4. Simplify the rule by applying the equalities (e.g. $1 \sim 2(a)$ and $1(a)-$ implies $2(a)-$).
5. Find conflicting absolute feature constraints (e.g., $2(a)$ and $2(a)-$ is

a contradiction).

6. **If** all tests are passed **then Return** OK
7. **Else Return** error

Steps 4 and 5 are repeated in the *specific satisfiability checking,* since the pseudo-features *gap* and *fgap* change values when the specific output rules are formed.

3.4 Intermediate format and code generation

The direct output of the translator is the dfv-items, lexical entries, and syntax rules in the intermediate format, which is a readable textual format. Here we have several examples:

1. dfv-items:

```
[D:ART:1]    ART    0(sg)
[D:N:1]      N      0(sg,p3sg,nom,gen) 0(p1,p2)-
[D:NP:1]     NP     0(sg,p3sg,nom,gen) 0(p1,p2)-
```

2. lexical entries:

```
[L:a :ART:1]         "a"        ART  0(sg)
[L:am :V:1]          "am"       V    0(be,p1,sg) 0(base)-
[L:am!0not :V:1]     "am not"   V    0(be,p1,sg) 0(base,pn)-
```

3. syntax rules:

```
[G:ADJL:1:ADJ ADJL:0]        ADJL -> ADJ ADJL
[G:NP:1:ART N:1]             NP   -> ART N=0:p3sg,sg=1:sg
[G:S:2:PP MOD NP1 VP:1]      S    -> PP+wh MOD NP1 VP<1>+base
[G:S:2:PP S:0]               S    -> PP S<1>
[G:S:3:NP1 *VP:1]            S    -> NP1+p3sg *VP+s-be
```

Each item is written on a line. The first part, which is delimited by brackets, is a key field used in sorting and removing duplicate and inferior items. The key field is removed afterwards, since it includes only redundant information.

The intermediate format itself can be an interesting result. For example, it explicitly presents the grammar induced by the parse forest.

Since this format obeys precise and strict rules, it can be handled using regular expressions, so further processing is implemented in Perl. Relying on the algorithms presented in Chapter 2, it is a quite straightforward process to transform the intermediate format into the final parser. The code generation is structured so that the final generation phase uses the output routines specific to a programming language. The routines are based on code snippets for the language in question. Two routine packages are developed—for C and for Java—to produce parsers in those two programming languages.

Chapter 4

Framework of MIN

In Chapter 1, we introduced the general structure of a MAS for InIR which consists of four types of agents: Q/A-agent, T-agent, I-agent, and L-agent. A Q/A-agent gets the query from the user, translates it into an inter-agent format and sends it to a T-agent. The T-agent can develop a high-level plan for solving the query, it can break it into sub-queries, and, generally, since it knows a lot about other agents, it can decide whom to send the query or sub-queries to. The sub-queries then reach various I-agents. An I-agent knows a lot about a certain domain and it can make the final plan and translate it into the low-level actions. Some of these actions might be done internally and others are realized using the L-agents. The L-agents are the "fingers" of the system. They get in touch with Internet resources, and they are the interfaces that translate the inter-agent language to the resource-specific language and vice versa. Then, information flows in the opposite direction—from L-agents to I-agents, which can do some intermediate processing, information extraction, or caching. The I-agents send the answers to the T-agent, which can do the fusion of several answers if obtained from several agents. The final results reach the Q/A-agent, which decides how to present them to the user.

Some characteristics of these agent types are given in quite vague terms. Although it is relatively clear what is expected from an L-agent and a Q/A-agent, the functionalities of the T-agents and I-agents are definitely not pinned down. A more precise specification of the T-agent and the I-agent activities would bring in many difficult issues such as the problems of planning and agent coordination. In the context of the MIN approach, these problems have to be addressed but we are not going to deal with them now. Instead, we will pack all these high-level problems into two boxes called the *virtual knowledge base* (VKB) and the *virtual control module* (VCM) of an agent and concentrate on the remaining issues. Elementary examples of these modules are implemented in the demonstration system.

The MIN framework at this stage is concerned about the questions of communication issues. It is implemented as a Java package that may be used to build the MAS's.

Before proceeding, let us explain and define some of the terms that will be used. We have already mentioned the agent *VKB* (*Virtual Knowledge Base*) and *VCM* (*Virtual Control Module*). Neither of these two modules is a real individual module. VKB is an abstraction of the agent's knowledge, which can be stored in any internal form—simple data items, database relations, inference rules, and so on. The agent can change its VKB. On the other hand, its VCM cannot be changed and it is an abstraction of the agent's built-in algorithms.

We deal with software agents, and it is usually clear when such an agent begins and ends—these are the program or process boundaries, which are well defined in an operating system or similar environment. We will say that an agent is *born* if it begins with a clear VKB; i.e., its VKB does not and cannot contain any changes made by the agent itself before this startup. Otherwise, we will say that the agent *wakes up.* If an agent ends in such a way that its next beginning will be a birth then we say that the agent *dies,* otherwise we say that it *goes to sleep.*

The last two definitions could be expressed in more simple terms, but they would be less precise; although, if we analyze them in detail, we can still find some controversial situations (e.g., does an agent kill itself or does it simply forget everything?). However, we will not have these problems in our discussion.

When discussing inter-agent communication we will usually analyze it from the viewpoint of one agent, usually denoted by the phrase *this agent.* The other agents will be denoted by the term *other agents.* It also applies to the situation where *this agent* communicates to one *other agent.* If *this agent* knows about an *other agent* (typically, knows about its name, its type, and how to contact it) then we say that the other agent is a *known agent,* otherwise it is an *unknown agent*, or a *foreign agent.* If an agent accepts connections from foreign agents, turning them into known agents, then we say that it is an *open agent,* otherwise it is a *closed agent.*

Section 4.1 introduces questions concerning inter-agent communication. Section 4.2 describes the MIN communication layers and gives specifications of the associated protocols. Section 4.3 presents the structures of the agent communication modules used in inter-agent communication. Section 4.4 describes the general structure of the four InIR agent types.

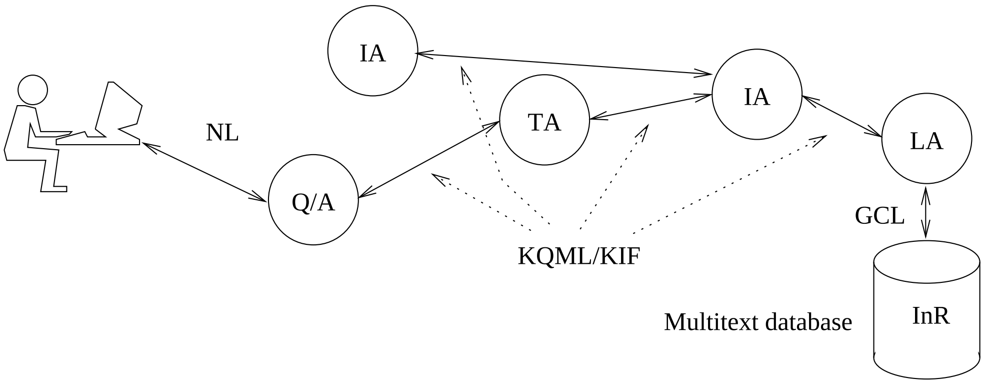

Figure 4.1: Example communication net

4.1 Communication issues

If we want to analyze a MAS, then the first sub-system to be considered is the communication infrastructure. MAS's are sometimes called MA (multi-agent) architectures and when we talk about an architecture here, we have in mind the topology and functionality of the links that interconnect the agents. There are different types of communication in a MAS—communication among agents, communication with humans, and communication with the environment.

Figure 4.1 shows an example situation in our envisioned MA approach. The communication links illustrated in the figure should not be regarded as some sort of static, long-lasting communication; instead, they show what links are activated in resolving an InIR query and what language is used in each of them.

The user interaction with the Q/A-agent is performed in a natural language. It is an important characteristic of our approach, but we do not exclude the use of other forms of user interfaces. They can be integrated with NL communication and, besides, the I-agent's NLP activity is more crucial. Ideally, we want to build open MA systems so the appropriate inter-agent language is the KQML/KIF combination. This seems to be the closest to the actual standard and that is why we have chosen it. The L-agents know resource-specific communication forms—that is their main purpose in a MAS. The Multitext database and its query language GCL [CB] will be our ongoing examples of a resource and its communication language.

NL and resource-specific communications are parts of specific Q/A-agents and L-agents and the discussions about them belong to the descriptions of those agents. A more general issue is concerns the management of inter-agent communication and our focus is here more on that.

KQML and KIF We introduced KQML and KIF in Chapter 1. KQML is an inter-agent language that provides a message format in which an agent can pass some information to other agents presenting its attitude towards that information, such as querying or stating. A KQML message is called the *performative* because it is "intended to perform some action by virtue of being sent" [LF97]. The information contained in a message is embedded in a field that is called the *content field* and it is expressed in another language, called the *content language* (although we can use KQML again). The knowledge representation language KIF is the content language that we will use.

The KQML specification does not include the lower transport layer because it is considered too implementation specific. For the purposes of the MIN approach, one such layer is developed as a part of the MIN framework. Besides being a bridge from the lower TCP layer to the higher KQML/KIF communication level, the new layer is concerned with timing and efficiency considerations important in InIR. A user who searches the Internet usually wants fast results. The protocols we rely on, such as TCP, have already inherent time delays and we do not want to increase them much more.[1] Alternatively, we cannot keep all inter-agent connections open all the time because that will not scale well with the increasing number of agents that we want to inter-communicate.

Both languages KQML and KIF have Lisp-like syntaxes. Since the MIN framework at this stage does not give a clear model higher than the KQML/KIF level, we will not explain their syntax in a systematic manner. Instead, some messages are illustrated in the following examples, which are very readable and self-explanatory.

---

[1] The abbreviation WWW has already got a new popular expansion—World Wide Wait.

Example of a session Before going into the description of the MIN framework, let us give an example of a session of a simple MIN MAS. The example illustrates what features are expected from the framework communication model.

1. The user, using a WWW browser from the Internet host `user.host`, opens a URL that is an interface to the MAS and gets a page with an applet that is actually a Q/A-agent.

2. The Q/A-agent is created and it invokes a CGI (Common Gateway Interface) script at the server machine that wakes up a T-agent TA, an I-agent IA-1, and an L-agent LA-1, if they are not awake already. The script returns the server port number of TA to the Q/A-agent (let us say 3469).

   By the *server port number,* we mean the socket number at which an agent listens for an incoming connection.

3. TA wakes up and listens to the TCP port 3469 at the host `ta.host`.

4. The Q/A-agent contacts TA and gets a unique name QA-1. QA-1 registers with TA using the KQML performative:

   ```
   (register :sender QA-1 :receiver TA :language KIF
       :ontology min-ontology
       :content (agent-info QA-1 QA user.host 0))
   ```

   The relation `agent-info` contains the following four components: the agent name, the agent type, the host name, and the server port number. In the case of QA-1, the server port number is 0 since it, as an applet, cannot open a server port. For security reasons, the applets run in a restricted environment and one of the restrictions is inability to open a server TCP socket.

   TA registers QA-1 in its virtual knowledge base.

5. The I-agent IA-1 wakes up knowing the host name and the server port number of the TA-agent from the start. It registers with TA in a way similar to QA-1 (although it does not have to get a unique name).

6. The L-agent LA-1 wakes up and registers with TA in the way same as IA-1.

7. TA informs IA-1 and LA-1 about each other by sending the following KQML performatives:

```
(register-agent :sender TA :receiver IA-1 :language KIF
    :ontology min-ontology
    :content (agent-info LA-1 LA la.host 3472))
```

   and

```
(register-agent :sender TA :receiver LA-1 :language KIF
    :ontology min-ontology
    :content (agent-info IA-1 IA ia.host 2743))
```

   At this point, the MAS is operational and ready to accept the user's queries.

8. The user communicates the following query to QA:

   > *Find me the number of articles containing the words java, lex, and yacc in the subject line.*

9. The query is transformed by QA into the KQML/KIF format and passed to TA in the form of a performative:

```
(ask-one :sender QA-1 :receiver TA :reply-with TA.1
    :language KIF :ontology min-ontology
    :content (length (keywords-and subject java lex yacc)))
```

10. TA passes the query to IA-1.

11. IA-1 asks LA-1 to send all articles containing the words `java`, `lex`, and `yacc` in the subject line by sending the performative:

```
(stream-all :sender IA-1 :receiver LA-1 :reply-with LA-1.1
    :language KIF :ontology min-ontology
    :content (keywords-and subject java lex yacc))
```

12. LA-1 translates the query into GCL and sends it to the Multitext search engine. The GCL query is:

```
("java" ^ "lex" ^ "yacc") < Subject
```

13. The Multitext engine sends the results (actually, one result article) to LA-1 in the HTML format:

```
<HTML>
<HEAD><TITLE>("java"^"lex"^"yacc")<subject</TITLE>
</HEAD><BODY bgcolor=#ffffff>
⋮
```

14. LA-1 translates the result into this performative:

```
(tell :sender LA-1 :receiver IA-1 :in-reply-to LA-1.1
    :language KIF :ontology min-ontology
    :content (article-outline
        "Re: LEX & YACC for Java"
        "A. Name <AName@a.host.net>"
        "comp.lang.java.programmer"
        29))
```

An article outline consists of its subject line, sender name, newsgroup name, and number of lines. If there were more results, more results would be sent. The end of the stream is notified by sending the performative `eos`:

```
(eos :sender LA-1 :receiver IA-1 :in-reply-to LA-1.1)
```

15. IA-1 simply counts the hits and, upon receiving performative `eos`, sends the answer to TA:

```
(tell :sender IA-1 :receiver TA :in-reply-to IA-1.1
      :language KIF :ontology min-ontology
      :content 1)
```

16. TA passes the answer to QA.

17. QA prints the answer to the user:

1 article.

18. When the user wants to finish the session, (s)he types *Bye.* to QA and QA dies.

19. The other agents TA, IA-1, and LA-1 go to sleep after the timeout time has elapsed if they do not have to serve any other QA agent.

4.2 MIN communication layers

The communication part of the MIN framework consists of several layers, which will be specified in this section. They are based on the TCP layer accessed using the Java socket class. The sub-layers are:

- the agent socket layer (AS),
- the message exchange layer (Comm, Communicator layer), and
- the KQML/KIF layer (KK).

In an inter-agent connection of two agents, the connection is always initiated by one agent and for that agent the connection is *outgoing* while for the other it is *incoming*. However, at the level of messages, each message is incoming or outgoing depending on whether an agent is receiving it or sending it.

In the following subsections, we describe each of the three layers and specify the associated protocols.

4.2.1 Agent socket layer (AS)

The agent socket layer provides a uniform interface to the underlying layer, which is the TCP layer. The Java programming language changes slightly from version to version and this layer is the interface that absorbs the changes in the network package. It also provides an abstraction with several basic operations needed for the upper layers.

The actual data transfer is described by the AS protocol, which is composed of the following operations:

AS-Open This is the initial phase of the incoming and outgoing connections. The sockets are opened and no communication is done. **AS-Open-In** and **AS-Open-Out** denote the incoming and outgoing parts of the protocol.

AS-Send A line is sent. The line is a string of non-new-line characters terminated by a new-line character.

AS-Receive A line is received. On an empty line or a line that starts with the string '`abort:`' the connection is closed and an error is reported. The new-line character at the end of line is stripped.

AS-Signal A line is sent and the output is flushed. This is used by the higher-level protocols to send a signal message that affects the communication.

AS-Abort A line starting with '`abort:`' and containing a message is sent and flushed, and the connection is closed.

AS-Close The output is flushed and the connection is closed.

4.2.2 Message exchange layer (Comm)

The message exchange layer defines how two agents exchange messages. It is based on the agent socket layer. This layer solves the problem of how to effectively send a set of KQML/KIF messages to another agent and how to accept the messages coming in the opposite direction. The agents can work in parallel on several independent tasks, and the messages from one task or another are not distinguished at this layer.

The following are the issues we deal with at this layer:

- mapping the known agent names to the actual physical addresses,
- the actual message exchange, i.e. taking turns in sending and receiving messages,
- taking care that this agent, when it is an applet agent (typically QA), receives messages although it is not allowed to open a server port,
- generating unique reply-with labels,
- taking care that the connection is open for a while if some reply messages are expected,

- checking that important known agents are awake (typically TA),
- performing a rudimentary security verification—when an incoming call from a known agent is accepted, the calling IP number is verified.
- resolving collisions when the other agent is trying to connect to this agent at the same time as this agent is trying to connect to the other agent.

The following four parameters affect the connection at the Comm level:

- *isOpen*
  If the agent is open, then it can accept calls from foreign agents, otherwise it cannot.
- *isApplet*
  If the agent is an applet, then it cannot open a server socket. In order to ensure reception of incoming messages, it must periodically open the connection even if there are no messages to be sent.
- *isVital*
  If the connection is vital, it needs to be checked periodically. If the other agent is not awake, then this agent goes to sleep.
- *isExpecting*
  If this agent is expecting a reply, then the connection is kept open for a longer time even if no messages are exchanged.

The Comm layer is asynchronous with the upper layer. The upper layer sends a message by putting it into the output buffer and does not wait for it to be sent. When receiving a message, it checks the input buffer to get the message.

The communication at this layer is described by the Comm protocol, which consists of the following phases:

Comm-Open-In A call is received from the other agent. It is managed through several sub-phases:

Comm-Open-In-1 A socket is accepted (AS-Open-In).

Comm-Open-In-2 A line is received, which is supposed to be the name of the other agent (AS-Receive).

We have four options depending on the value of the parameter *isOpen* and on whether the received agent name starts with a '`?`'—which means that the other agent expects to be given a unique name. We will call this kind of name *?-name*. Otherwise, we will say that the name is *normal*. The options are:

1. This agent is closed and the name is normal.
   If the other agent is known, then proceed, otherwise abort the connection (AS-Abort).
2. This agent is closed and the name is a ?-name.
   Abort the connection (AS-Abort).
3. This agent is open and the name is normal.
   If the other agent is known, then proceed, otherwise, register a new agent and proceed.
4. This agent is open and the name is a ?-name.
   The question mark at the beginning of the name is removed and, by appending a number to the rest of the name, a new unique name is created. The name is sent back (AS-Signal).
   A line is received (AS-Receive) and if it contains the same name that was sent, then proceed, otherwise abort (AS-Abort).

Comm-Open-In-3 Check whether this agent is trying to connect to the other agent at the same time. If the collision is detected, send the signal '`cancel`' (AS-Signal) and close the connection (AS-Close), otherwise proceed.

Comm-Open-In-4 Signal '`ok`' (AS-Signal) and proceed.

Comm-Open-Out The call is made by this agent.

Open the connection (AS-Open).

Send this agent's name (AS-Signal).

Receive a line (AS-Receive). If '`cancel`' is received then close the connection (AS-Close) because a collision is detected. The connection is retried to open after a pause. The pause time is doubled each time and a random variation is added in order to avoid synchronous collisions.

If this agent's name is a ?-name, then the received line is a new name. Accept it, send it back (AS-Signal), and receive a line (AS-Receive).

If the last received line is '`ok`' then proceed, otherwise abort the connection (AS-Abort).

Comm-Exchange This phase handles the actual exchange of messages. The exchange is synchronized so that the agent that made the call sends messages first. After it has sent all messages, the roles are exchanged, and so on.

If in two consecutive operations of Comm-Exchange-* (Comm-Exchange-Listen or Comm-Exchange-Talk) nothing is exchanged, then both parties wait a small period of time and try again. This is repeated a couple of times and if nothing was exchanged, then the connection is closed. The time spent in keeping the connection open depends on the parameter *isExpecting*.

The sub-phases are done as follows:

Comm-Exchange-Talk Keep sending messages (AS-Send) until the output buffer is empty.

Signal '`switch`' (AS-Signal).

Comm-Exchange-Listen Keep receiving messages (AS-Receive) until the line '`switch`' is received.

The lines containing the string '`ping`' are ignored. These lines are sent when it is necessary to open the connection periodically, such as with the applet agent or in vital connections. Their only purpose is to open the connection, so they can be ignored after they reach the destination.

4.2.3 KQML/KIF layer (KK)

In this layer, an agent constructs the outgoing KQML/KIF messages and the incoming messages are parsed. First, the KQML part is parsed and if the message contains a content field, the KIF parser finishes the process. In case of a parsing error, a KQML error performative is sent (by putting it in the output buffer).

Besides accepting and sending messages, the KK layer is connected with the Comm layer in two additional ways:

- the Comm layer generates unique reply-with labels for the KQML performatives, and
- the parameter *isExpecting* of the Comm layer is directly manipulated by the KK layer.

4.3 KQML/KIF communication module

The KQML/KIF communication module (KKCM) handles the communication of an agent with other agents at the communication layer KK and all lower layers. In one direction, it receives messages from the agent VCM and sends them to the appropriate agents and, in the other direction, it receives messages from the other agents and, after doing KQML and KIF parsing, leaves them to be picked up by VCM.

There are two kinds of communication modules: the applet communication module and the team communication module.

4.3.1 Applet KKCM

The applet communication module (Applet KKCM) is used by the applet agents. It cannot listen to a server port and it does not have contact with more than one other agent. The structure of the applet KKCM is shown in Figure 4.2.

When a message has to be sent, then one of the KQML/KIF generator methods is invoked and the KQML/KIF message is generated and put into the output buffer. The communicator is a daemon thread running in an infinite loop. It checks whether there are any messages in the output buffer and, if there are, starts the connection (Comm-Open-Out).

During the Comm-Exchange phase, some messages are received. If they are not ignored (ping messages) they are parsed, asynchronously by the interpreter threads. If a parse error occurs, the message is discarded and an error message is put into the output buffer. The final interpretations are stored into the input buffer, where the VCM can pick them up.

The AS level of communication is handled by the agent socket. The Comm level is handled by the communication manager. The communicator handles the Comm-Exchange part of the protocol. The KK level is handled by the KQML/KIF interpreter and the KQML/KIF generator methods.

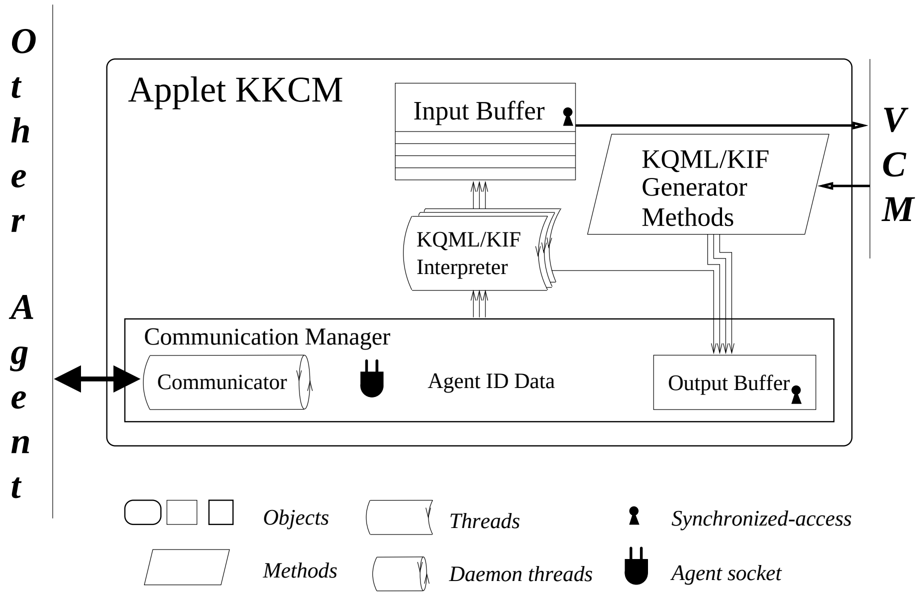

Figure 4.2: Applet KKCM structure

4.3.2 Team KKCM

The agents that are not applets are less restricted, so they can communicate to more than one other agent. The communication is realized over an array of communication managers, called the communication team, hence this kind of KKCM module is called the team KKCM.

Figure 4.3 shows the structure of the team KKCM. The main difference compared to the applet KKCM is the listener. The listener daemon opens a server socket and listens for incoming connections. It handles the phase Comm-Open-In-1 of the Comm protocol. The phase Comm-Open-In-2 is handled by the communication team object, and Comm-Open-In-3 and 4 by the communication manager. As with the applet KKCM, the communicator performs the Comm-Exchange phase.

Figure 4.4 shows the communication layers and the objects and methods in the KKCM team that handle the protocol portions.

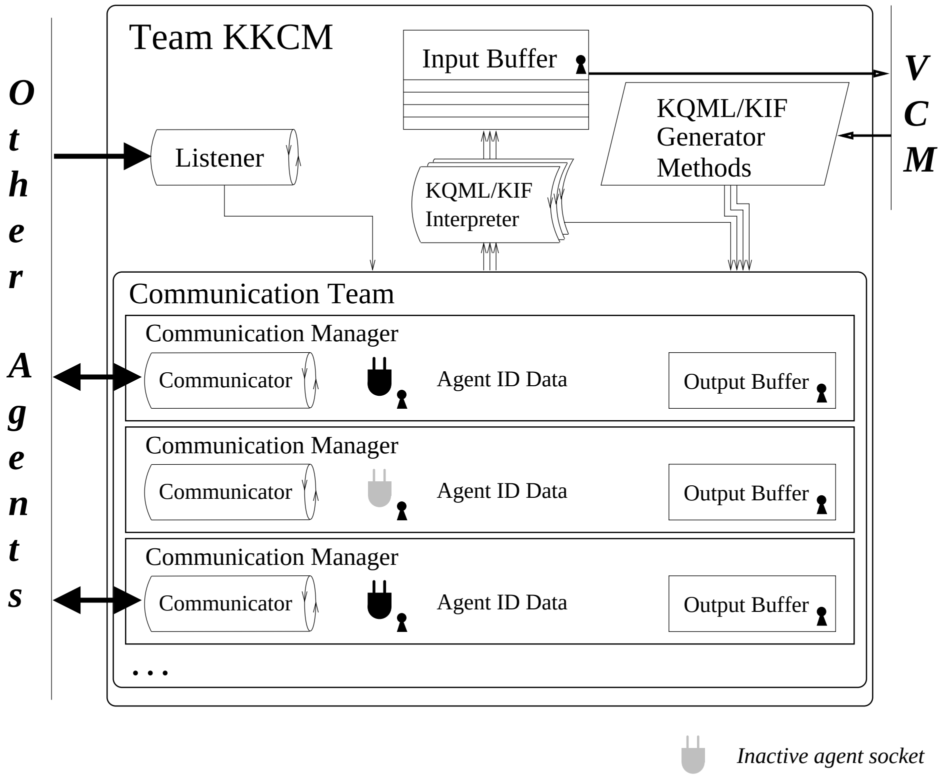

Figure 4.3: Team KKCM structure

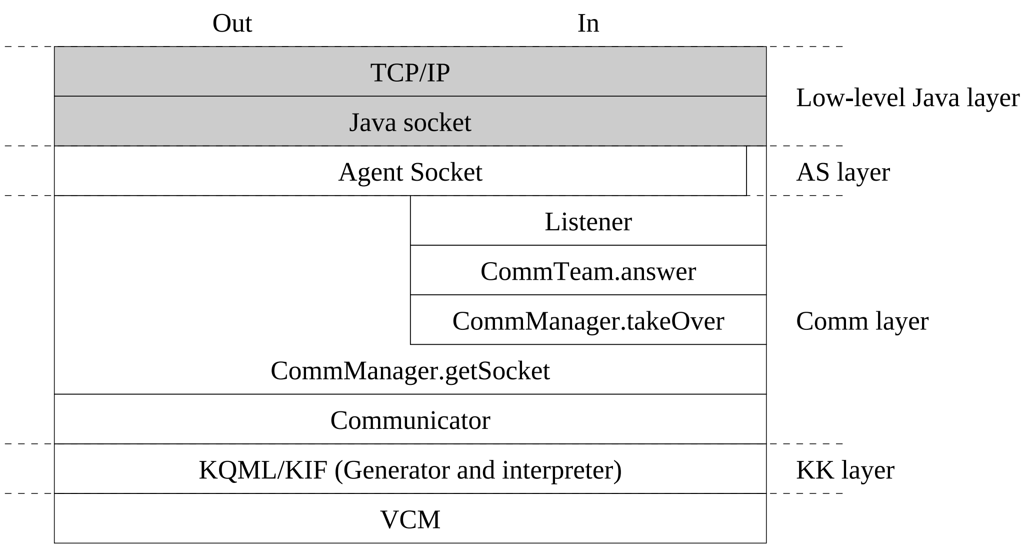

Figure 4.4: Communication layers

4.4 Agent structure

We now give a coarse structure for each agent type. The complete MIN framework will include more specific agent models. However, it is left for future work and, for now, we present only a high-level algorithm and the agent structures that were the motivation for building these lower layers.

The following algorithm is the general VCM algorithm (the main agent algorithm):

Algorithm XVII: **General VCM algorithm**

Input: VKB
Output: VKB (changed)
(*The exchanged messages are also part of the input and output.*)

1. *Sleepy* $\leftarrow$ false
2. Load the persistent part of VKB and initialize
3. **While not** *Sleepy* **do**
4. | Accept an out-of-context message m
5. | Start a new task-handling thread
6. Wait for all threads (except daemons) to finish
7. Save the persistent part of VKB

While a task is handled, a series of messages is exchanged with other agents and the environment. These messages are part of a context. The messages that are not related to any existing task are called *out-of-context messages.*

The conceptual structure of each agent type is presented in Figures 4.5–4.8. The figures speak for themselves. We will not describe them further.

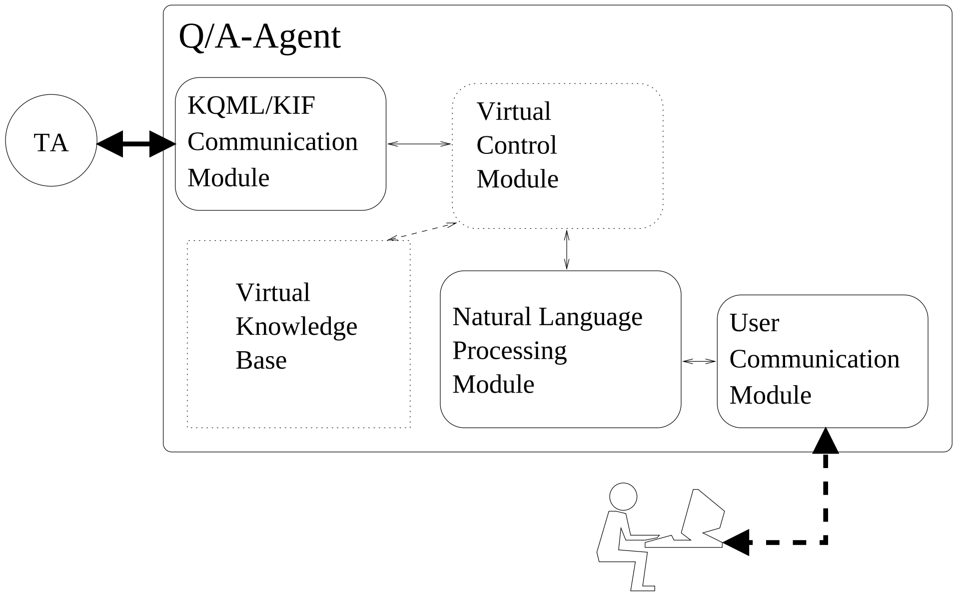

Figure 4.5: Q/A-agent structure

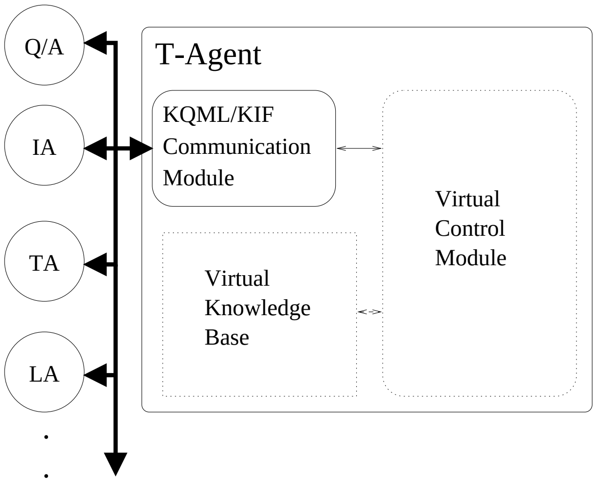

Figure 4.6: T-agent structure

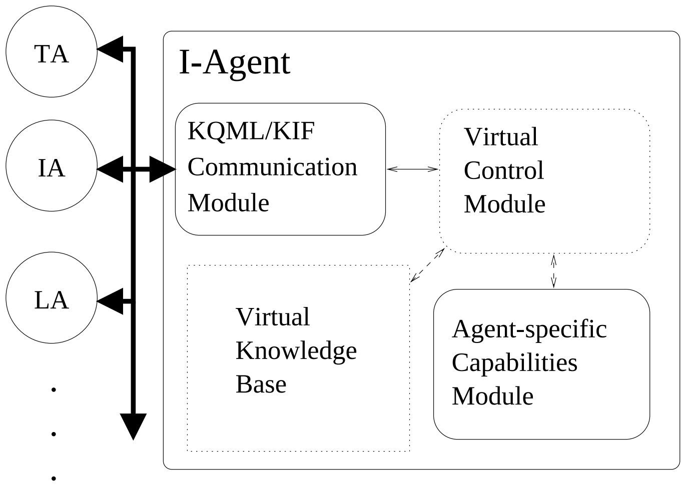

Figure 4.7: I-agent structure

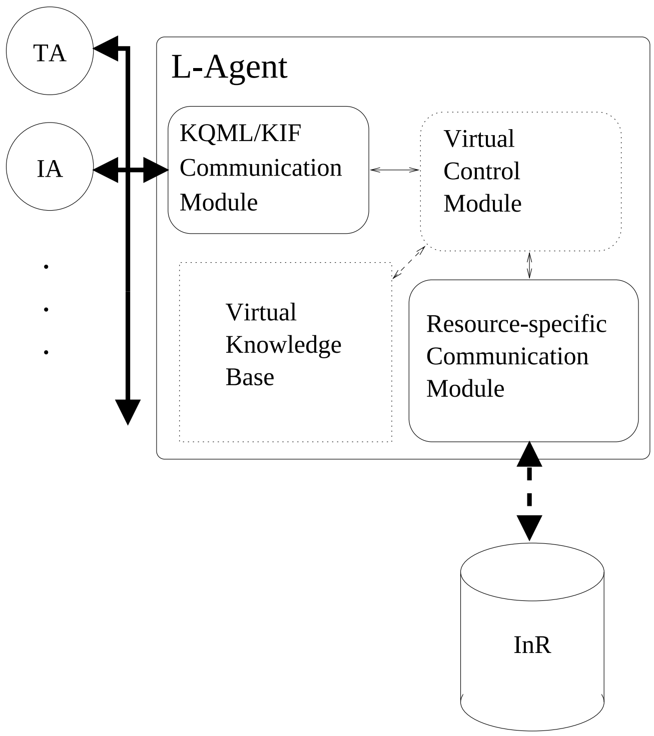

Figure 4.8: L-agent structure

Chapter 5

Experiments and Demonstration

The experiments and testing are not done on a scale that could be considered a proof of stated goals. However, they are encouraging and worth mentioning.

There are two sections in this chapter. Section 5.1 presents the testing results of the system NL PAGE, and Section 5.2 describes a demonstration with a simple MAS.

5.1 NL PAGE

An experiment with the parser generator NL PAGE was done using 31 English sentences.[1] The grammar constructed included all feature constraints that we discussed, as well as handling of movement phenomena.

The input forest generated a lexicon containing 100 lexemes, and a grammar containing 206 rules. Obviously the emphasis was on the grammar and not the lexicon. Managing a large lexicon is an old problem and efficient techniques for dealing with it are well known. In our example, the lexicon is simply stored in the form of a sorted array and the lexemes are found using

[1]The list of sentences and obtained parses is given in Appendix D.

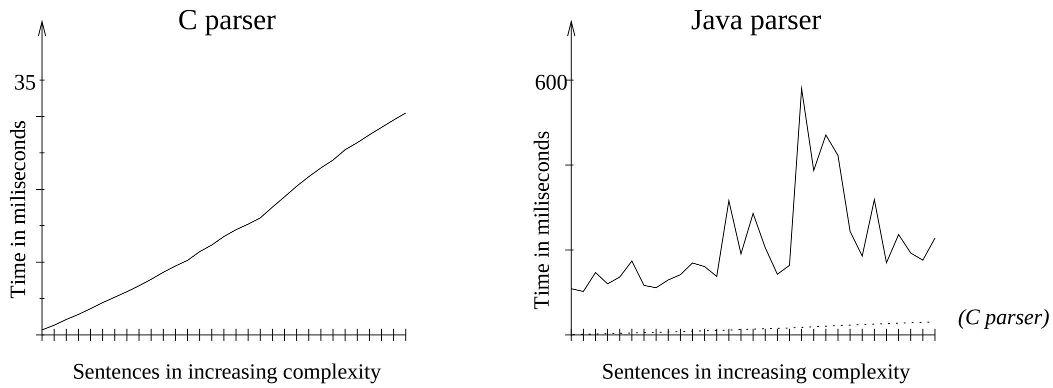

Figure 5.1: Timing results for the generated C and Java parsers

binary search.

The following timing experiments are done on a Sun SPARCserver 670MP machine running the SunOS 4.1.2 operating system. The C compiler gcc 2.7.2 was used to compile the C code and the Sun Java Development Kit (JDK) 1.0.2 was used to compile and execute the Java code.

The generated parsers (C and Java) are run on 31 test sentences in increasing order of complexity and the timing results obtained are shown in Figure 5.1. The actual numbers are given in Table 5.1. The minimum, maximum, and average parsing times and standard deviations are given in Table 5.2.

Because the test sentences are ordered by increasing complexity (more or less), it is not a surprise to see that the time spent by the C parser is monotonically increasing (almost linearly), as shown in the figure. The behavior of the Java parser parsing time is more chaotic.

The parsers were not compared to any known parsers. However, an experimental parser built in Lisp and using a more standard parsing algorithm was tested on the same sentences and its parsing times were in a range approximately one second to a minute per sentence.

Sentence	C parser ($msec$)	Java parser ($msec$)	Sentence	C parser ($msec$)	Java parser ($msec$)
1.	0.68	108.7	16.	13.52	285.8
2.	1.33	102.1	17.	14.45	205.3
3.	2.11	146.8	18.	15.21	142.8
4.	2.82	120.2	19.	16.06	163.7
5.	3.60	136.1	20.	17.55	579.9
6.	4.44	173.9	21.	18.96	387.3
7.	5.19	116.4	22.	20.42	470.8
8.	5.94	111.0	23.	21.74	422.3
9.	6.75	129.3	24.	22.94	243.7
10.	7.63	141.7	25.	24.02	185.7
11.	8.60	169.3	26.	25.43	318.4
12.	9.47	160.8	27.	26.40	169.9
13.	10.24	137.5	28.	27.48	236.2
14.	11.44	315.8	29.	28.50	192.7
15.	12.35	190.8	30.	29.51	176.0
			31.	30.49	227.9

Table 5.1: Parsing times

	C parser ($msec$)	Java parser ($msec$)
Minimum parsing time	0.68	102.1
Maximum parsing time	30.49	579.9
Average parsing time	14.36	215.1
Standard deviation	9.35	115.7

Table 5.2: Parsing time statistics

5.2 A simple MIN system

A simple demonstration MIN system integrates the results presented so far. The NL PAGE parser generator is used to build a parser in Java, which is then incorporated into a Q/A-agent. Three other agents are built: a T-agent, an I-agent, and an L-agent. The inter-agent communication relies on the communication layers presented in the previous chapter.

Figures 5.2–5.5 show four images captured from Netscape windows during a system run. As we can see, the time spent for parsing the query is negligible (0.049 seconds) while answering the query took 14 seconds. The time was spent on the inter-agent message exchange (QA-TA-IA-LA and back), on the connection between the L-agent and the Multitext database, and on the message processing (each agent does at least KQML/KIF parsing). There is space for improvement in terms of efficiency, which is needed if we consider that the NL grammar and lexicon are very small, and that the agents do very light processing in this system.

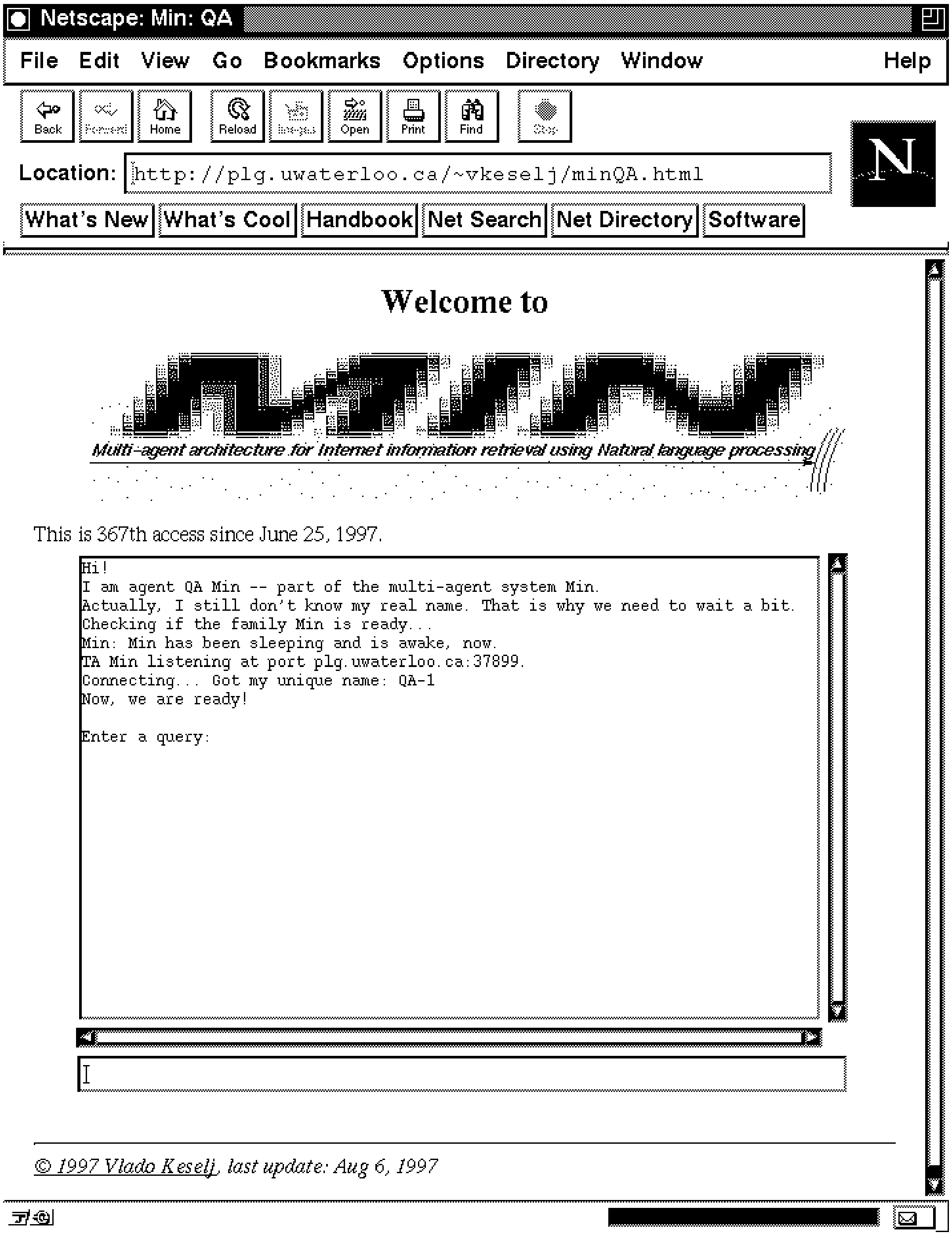

Figure 5.2: Demonstration MIN MAS: Initialization

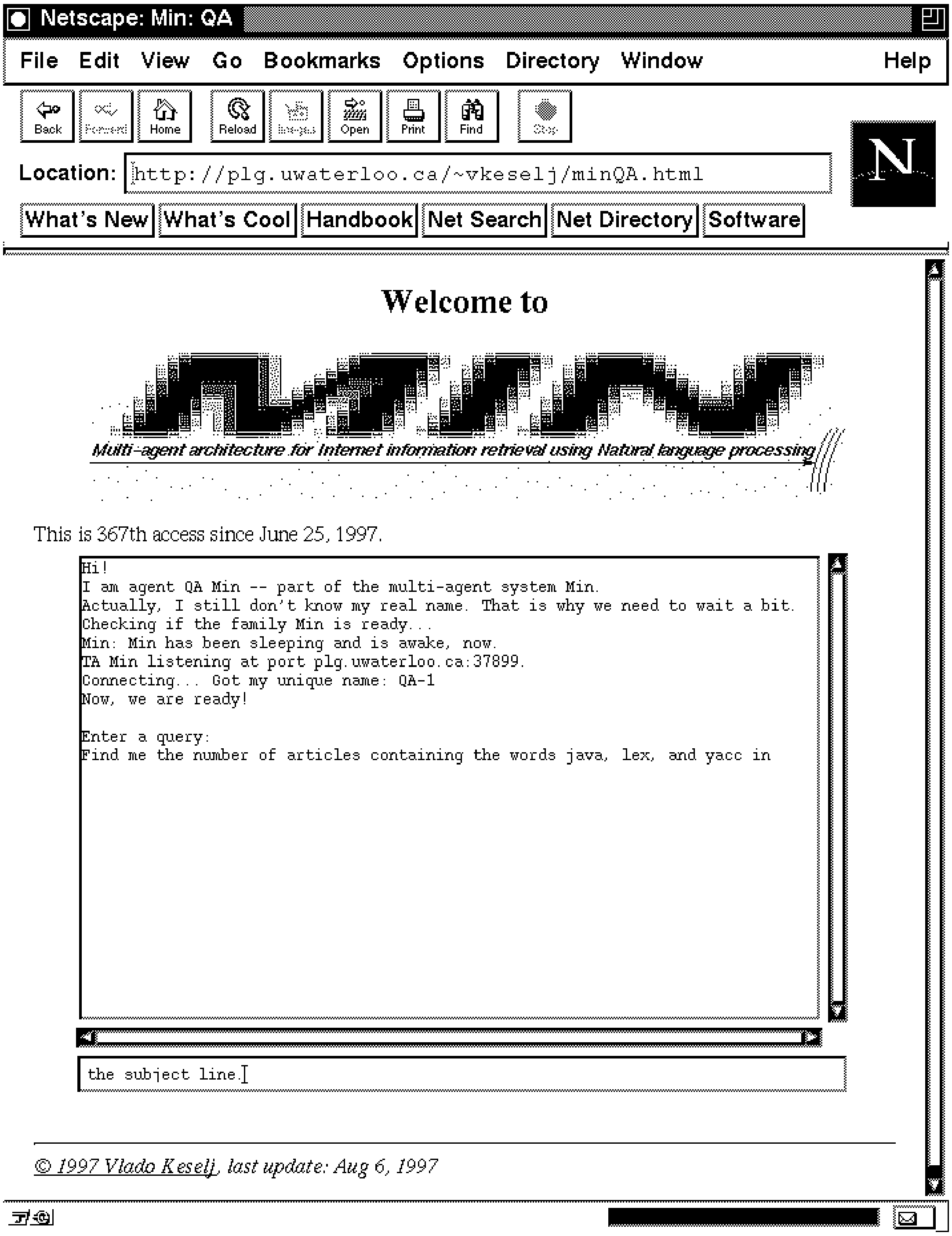

Figure 5.3: Demonstration MIN MAS: Entering a query

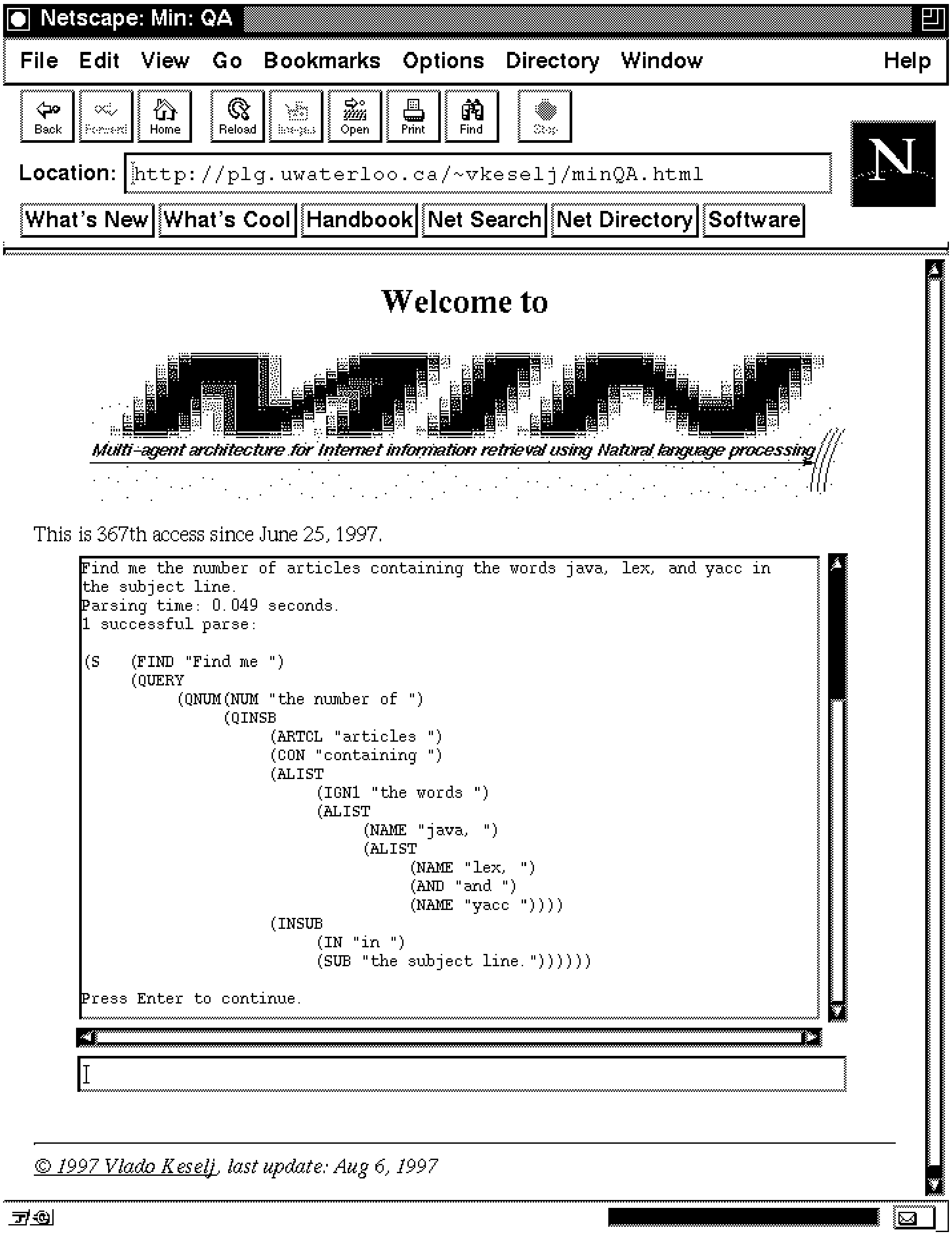

Figure 5.4: Demonstration MIN MAS: NL parsing

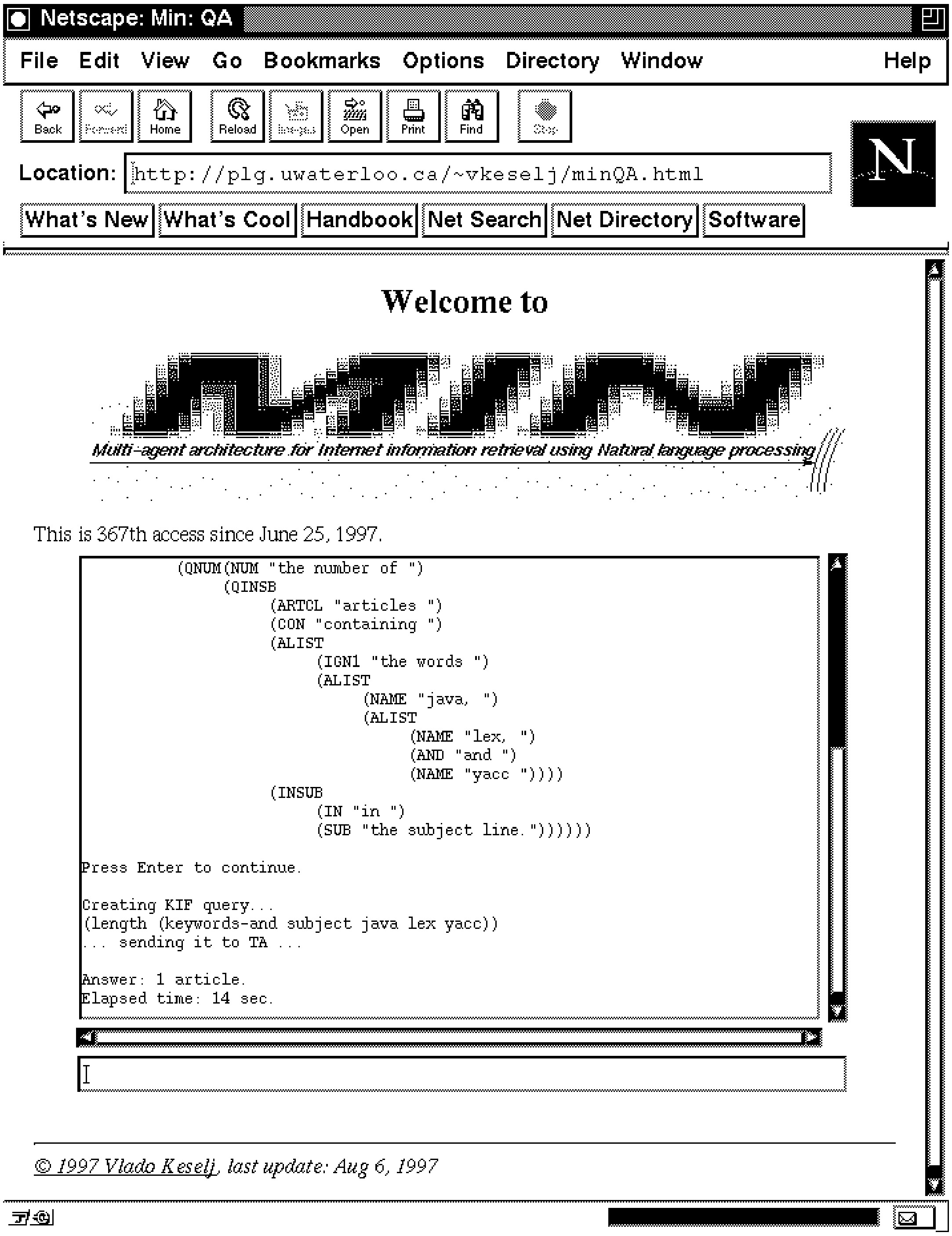

Figure 5.5: Demonstration MIN MAS: Retrieving the answer

Chapter 6

Conclusion

In this work, we have defined a problem—the problem of InIR—and we proposed an approach to this problem that combines natural language processing and multi-agent systems.

The Internet information space is both vast and complex—it embraces the whole planet and has heterogeneous contents related to all kinds of human activity. Therefore, the InIR problem is a hard and complex one requiring that many solution models be extensively tried to obtain a satisfactory retrieval system.

It is recognized that multi-agent systems could provide a scalable and flexible solution. There is a long list of projects that can be classified as agent-based information retrieval systems [Sob]. The MIN approach can be seen as a part of this large research effort.

The use of NLP in IR, including InIR, is not a new idea. However, it is a new approach to combine NLP with MAS's on this task. The research direction set in this work was to examine the combination of MAS's and NLP in solving the problem of InIR, and to make some first specification and implementation steps that will be the basis for further research.

The first step was to specify an NLP model that would satisfy the requirements imposed by our environment. The specification is based on a trade-off between the NLP formalisms proved to describe natural languages well (at least English and the like), and some of the opposing requirements such as efficiency considerations.

The NLP model found is not appropriate to be used directly. In addition to this argument, the need for numerous parser variations, the need for flexible maintenance, and the generally dynamic nature of any NL lead us to the construction of a parser generator—NL PAGE.

The generator can generate parsers in both the C and Java programming languages. It is independent of both the natural language in question and the target parser programming language. NL PAGE is the first step towards MIN systems. It provides a tool for building the NLP agent components.

The second part of this thesis is the MIN framework—a Java-based multi-agent framework. It is relatively independent of NL PAGE. Their only connection is the purpose for which they are built, and that is the design of MIN systems. Special attention is paid to the inter-agent communication module, which includes the KQML/KIF layer and the lower layers. In the KQML specification, the transport layer, is not specified since it is too implementation specific. We implemented the transport layer which satisfies the requirements defined by the needs of a future MIN system.

The MIN framework is presented in an intermediate stage. It is not supposed to consist only of the transport layer implementation, but should also embrace the agent specification above the KQML/KIF level. At this stage, this higher-level part of the framework is described in very general terms.

The number of open issues is much larger than the number of solved ones. We list some of them:

- NL PAGE is tested with a small grammar and lexicon. Before building a working MIN prototype, several carefully prepared and complex parsers should be created for Q/A-agents and I-agents. For this purpose, appropriate parse forests have to be constructed.

  As a part of this effort, NL PAGE should be tested with large grammars and large lexicons.

- What is the inter-agent query format? We know that it is expressed in the combined KQML/KIF language, but the ontologies (a part of the KIF specification) for certain domains have to be specified. This issue must be resolved so that we know how to form the query that is passed by a Q/A-agent to other agents.

  The format has to support the conceptual IR.

- How does an agent advertise its capabilities to other agents? We have to decide how to represent any specific InIR capabilities that an agent can have. The general concept of an agent advertising its capabilities is described in the KQML specification but more should be said in the context of InIR.

  For example, how can an L-agent advertise itself? We could manually construct the representation of its abilities, but if the resources for which it is responsible change, then it should be able to change its advertising information.

- The question of agent coordination. When a new agent joins a MAS and advertises itself, how should we interconnect it with other agents?

- Agent planning. When a T-agent or I-agent accepts a query, it has to decide whether to break the query into sub-queries, how to form new queries, which agents to send it to, and what internal actions to perform.

- Caching and document indexing. How does an I-agent use its NLP capabilities? How should documents be indexed? How are documents matched to queries? How can information be reused, i.e. how can results be cached?

There are many open questions and the answers to many will be difficult to obtain. But the pressing need for better orientation tools in the exploding Internet space forces us to keep looking for solutions.

Appendix A

Abbreviations

		page number
AI	Artificial Intelligence	7
AOP	Agent-oriented Programming	8
AS	Agent Socket communication layer	82
CGI	Common Gateway Interface	78
Comm	Message exchange communication layer Communicator layer	82
CPU	Central Processing Unit	55
dfv-item	default feature value information	58
FTP	File Transfer Protocol	2
HTML	Hypertext Markup Language	4
I-agent	Intermediate Agent	15
IA	Intermediate Agent	15
InI-space	Internet Information space	3
InIR	Internet Information Retrieval	1

IP	Internet Protocol	2
IR	Information Retrieval	3
KAPI	KQML Application Programmer's Interface	22
KIF	Knowledge Interchange Format	11
KK	KQML/KIF communication layer	82
KKCM	KQML/KIF Communication Module	88
KQML	Knowledge Query and Manipulation Language	11
L-agent	Low-level retrieval Agent	15
LA	Low-level retrieval Agent	15
MA	multi-agent	76
MAS	Multi-agent System	7
MIN	Multi-agent system for Internet information retrieval using Natural language processing	2
NL	Natural Language	3
NLP	Natural Language Processing	12
NL PAGE	NL Parser Generator	23
OOP	Object-oriented Programming	8
Q/A	Query-answering Agent	15
Q/A-agent	Query-answering Agent	15
QA	Query-answering Agent	15
T-agent	Top-level planning Agent	15
TA	Top-level planning Agent	15
TCP	Transmission Control Protocol	2
TCP/IP	Transmission Control Protocol/Internet Protocol	2

URL	Uniform/Universal Resource Locator	3
VCM	Virtual Knowledge Module	74
VKB	Virtual Knowledge Base	74
WWW	World Wide Web	2

Appendix B

Notation and Terminology

B.1 Algorithms and data structures

The notation used in the algorithms is quite straightforward and readable, so we will list the less common conventions:

- There are no beginning and ending statements for a block. Instead, blocks are denoted by indentation and a vertical rule.

- Algorithm names are printed in boldface font (e.g., **ParseN_X**). The algorithm input parameters as well as the local variables are printed in italic font (e.g., *sp*), and the global variables are printed in sans-serif font (e.g, Chart).

B.2 Regular expressions

We use *regular expressions* to represent *regular sets.* We are going to define them in an implicit way through the following notation and terminology description.

The *visible literal characters* are represented in typewriter font. They can be concatenated in strings that represent literal strings. For example:

`abc`	represents a string of 3 letters;
`112%()*+`	is a string of 8 characters.

The list of all literal visible characters follows: `!`, `"`, `#`, `$`, `%`, `&`, `'`, `(`, `)`, `*`, `+`, `,`, `-`, `.`, `/`, `0`, `1`, ..., `9`, `:`, `;`, `<`, `=`, `>`, `?`, `@`, `A`, `B`, ..., `Z`, `[`, `\`, `]`, `^`, `_`, `` ` ``, `a`, `b`, ..., `z`, `{`, `|`, `}`, and `~`.

The *invisible literal characters* (space, tab, and new-line) are represented as ␣, ⇥, and ↵, respectively. The visible literal characters and the invisible literal characters are called the *literal characters.* The set of literal characters has a double role in a certain sense. Besides being used in regular expressions, the set is actually the alphabet over which we want to construct and represent our regular languages (as well as other languages). When we want to precisely refer to the latter function we will use the term *target alphabet.*

All characters that occur in regular expressions and are not literal characters are called *meta-characters.* An example of meta-characters is the "true" spaces. Any kind of "true" space (spaces, tabs, or new-lines) occurring in a regular expression should be ignored or understood as separators (i.e. not being part of the regular expression). It should be clear from the context which of these two options is valid. For example, if a regular expression is too long to be typeset in one line of the text, then it will be broken in several lines without any special markup.

As we will see, regular expressions have recursive structure, i.e. some of their parts (connected parts or substrings, more precisely) are also regular expressions themselves. We will call these parts *subexpressions.*

If a regular expression or a subexpression of a regular expression represents a certain set of strings (i.e. language or sublanguage), then we say that that regular expression *matches* those strings. Hence, we can say that any literal character matches itself. Any string of literal characters is called the *literal string* and it also matches itself. The literal strings and the literal characters are called the *literals.*

The two meta-characters [and] (which should be distinguished from the literals `[` and `]`) are used to create a regular expression that matches one-character strings from a set. The expression is a list delimited by meta-characters [and] and consisting of literal characters and meta-character –. The set complement operation can be applied to this expression. For example, an expression can be built:

1. By listing the characters. E.g., [abcd␣⎘] matches any of the characters a, b, c, d, ␣, or ⎘.

2. By using the meta-character – (to be distinguished from the literal -), which is used to represent all characters from an interval. E.g., [_a–z.] matches _, ., or any of the lowercase letters. The ordering from the above list of the visible literal characters is used (ASCII ordering). Only visible literal characters can be used as the limits of the interval.

3. By using the complement. A line above an expression between [and] means the complement set operation. E.g., $\overline{[\texttt{A–Za–z}]}$ matches any character that is not a letter.

To match the empty string, we will use the expression ε. The expression [ ] is a regular expression that does not match any string and it seems to be quite useless. However, after applying the set complement operation we get the expression $\overline{[\,]}$ which matches any one-character string and is very useful and used frequently.

We have already implicitly used concatenation in the definition of literal strings. Generally, the concatenation of two regular expressions creates a new regular expression that represents the concatenation of the two regular languages. E.g., the regular expression [A–Z][0–9] represents all two-character strings such that their first character is an uppercase letter and their second character is a digit.

The Kleene closure is denoted by * (to be distinguished from *). E.g., the regular expression $\texttt{0}^*$ represents all strings of zeros (including the empty string). The Kleene closure (as well as other "superscript" meta-characters) has higher priority than concatenation. When we need to override this rule we use meta-characters (and) (to be distinguished from (and)). If e is a regular expression, then (e) is the same regular expression but we made sure that it won't be "broken" by a high-precedence operator in a larger expression. E.g., $\texttt{01}^*$ represents the strings 0, 01, 011, …; while the expression $(\texttt{01})^*$ represents the empty string, 01, 0101, ….

If e is a regular expression, then $((e)(e)^*)$ can be written as $(e)^+$ (to be distinguished from the literal +). This operation is called the *positive closure.*

The Kleene closure and the positive closure can be defined in the following way: If the regular expression e represents the language L, then the regular expression $(e)^*$ matches any finite concatenated sequence of words from L. The empty sequence, i.e. the empty string, is also included. If we do not want to include the sequence of zero words then we use the positive closure $(e)^+$. If we want to match only the sequences of n words ($n = 1, 2, \ldots$) then we write $(e)^n$. If we want to match only the sequences of m to n words ($m, n = 0, 1, 2, \ldots,\ m \leq n$) then we write $(e)^n_m$. The expression e^1_0 can be also written as $e^?$.

If the regular expression e_1 represents the language L_1 and the regular expression e_2 represents the language L_2, then the language $L_1 \cup L_2$ is represented by the regular expression $(e_1) \mid (e_2)$. The meta-character $\mid$ should be distinguished from the literal `|`. The precedence of the operator $\mid$ is lower than the precedence of concatenation.

We frequently use regular expressions in lexical analysis. In lexical analysis, a long string is processed (typically a file, in a physical sense), so that smaller substrings are extracted (called lexemes) by matching them with the *regular patterns*. Any regular expression is a regular pattern. The regular patterns have somewhat richer structure, since they can include some information about the context in which the matched string appears.

The meta characters called *anchors* ($\vdash$ is called *left anchor* and $\dashv$ is called *right anchor*) can be used in a regular pattern. If e is a regular expression, then the regular pattern $\vdash e$ matches those strings that are matched by the regular expression e and that are positioned at the beginning of a line (at the beginning of the file or preceded by a new-line character). Given two regular expressions e_1 and e_2, the pattern $e_1 \dashv e_2$ matches all strings matched by e_1 that are followed, in a wider context, by a string matched by e_2. For example, the regular pattern $\vdash ␣^+$ matches a non-empty sequence of spaces found at the beginning of a line, while the pattern $␣^+ \dashv ↵$ matches a sequence of spaces found at the end of a line (assuming that each line has a new-line character at the end).

Appendix C

Web links

C.1 Search engines

http://www.altavista.digital.com/	AltaVista
http://www.dejanews.com/	DejaNews
http://www.excite.com/	Excite
http://www.hotbot.com/	HotBot
http://www.infoseek.com/	Infoseek
http://www.lycos.com/	Lycos
http://www.opentext.com/	Open Text
http://www.planetsearch.com/	Planet search
http://www.webcrawler.com/	WebCrawler
http://www.yahoo.com/	Yahoo

C.2 All-in-one pages

http://www.traveller.com/aliweb/	ALIWEB
http://www.albany.net/allinone/	All-In-One
http://pubweb.nexor.co.uk/public/cusi/cusi.html	CUSI
http://www.iTools.com/find-it/find-it.html	Find-It
http://www.search.com/	Search.com

C.3 Meta-search engines

http://lorca.compapp.dcu.ie/fusion/	Fusion
http://www.highway61.com/	Highway 61
http://m5.inference.com/ifind/	Inference find
http://www.mamma.com/	Mamma
http://metacrawler.cs.washington.edu/	Metacrawler
http://www.designlab.ukans.edu/profusion/	ProFusion
http://williams.cs.colostate.edu:1969/	SavvySearch

C.4 Alternative search sites

http://www.humansearch.com/	Human Search
http://www.pacprospector.com/input.html	PACprospector

Appendix D

Test sentences

These sentences are used to test and measure the timings of the NL PAGE system:

1. I give

```
(S   (NP1 (NP  (PRO "I ")))
     (VP  (V "give.")))
```

2. I gives.

```
No successful parses!
```

3. He was given.

```
(S   (NP1 (NP  (PRO "He ")))
     (V "was ")
     (VP  (V "given.")))
```

4. Him was given.

```
No successful parses!
```

5. We was given.

```
No successful parses!
```

6. We were given.

```
(S   (NP1 (NP  (PRO "We ")))
     (V "were ")
     (VP  (V "given.")))
```

7. I can give.

```
(S   (NP1 (NP  (PRO "I ")))
     (MOD "can ")
     (VP  (V "give.")))
```

8. I can given.

```
No successful parses!
```

9. I do give.

```
(S   (NP1 (NP  (PRO "I ")))
     (V "do ")
     (VP  (V "give.")))
```

10. The man has given.

```
(S   (NP1 (NP  (ART "The ")
                (N "man ")))
     (V "has ")
     (VP  (V "given.")))
```

11. The nice men have given.

```
(S   (NP1 (NP  (ART "The ")
                (ADJL(ADJ "nice "))
                (N "men ")))
     (V "have ")
     (VP  (V "given.")))
```

12. You give a stamp.

```
(S   (NP1 (NP  (PRO "You ")))
     (VP  (V "give ")
          (VPtl(NP  (ART "a ")
                    (N "stamp.")))))
```

13. You give I.

```
No successful parses!
```

14. I had been giving men a chance.

```
(S   (NP1 (NP  (PRO "I ")))
     (V "had ")
     (VP  (V "been ")
          (VP  (V "giving ")
               (VPtl(NP  (N "men "))
                    (NP  (ART "a ")
                         (N "chance."))))))
```

15. I gave at the office.

```
(S   (NP1 (NP  (PRO "I ")))
     (VP  (V "gave ")
          (VPtl(PPL (PP  (PREP "at ")
                         (NP  (ART "the ")
                              (N "office.")))))))
```

16. The stamp was bought by the office.

```
(S   (NP1 (NP  (ART "The ")
               (N "stamp ")))
     (V "was ")
     (VP  (V "bought ")
          (VPtl(PPL (PP  (PREP "by ")
                         (NP  (ART "the ")
                              (N "office.")))))))
```

17. This stamp is for you.

```
(S   (NP1 (NP  (DET "This ")
               (N "stamp ")))
     (V "is ")
     (VPtl(PPL (PP  (PREP "for ")
                    (NP  (PRO "you."))))))
```

18. This buys it.

```
(S   (NP1 (NP  (PRO "This ")))
     (VP  (V "buys ")
          (VPtl(NP  (PRO "it.")))))
```

19. Some give to you.

```
(S   (NP1 (NP  (PRO "Some ")))
     (VP  (V "give ")
          (VPtl(PPL (PP  (PREP "to ")
                         (NP  (PRO "you.")))))))
```

20. This stamp was bought at the office by you for me.

```
(S   (NP1 (NP  (DET "This ")
               (N "stamp ")))
     (V "was ")
     (VP  (V "bought ")
          (VPtl(PPL (PP  (PREP "at ")
                         (NP  (ART "the ")
                              (N "office ")))
                    (PPL (PP  (PREP "by ")
                              (NP  (PRO "you ")))
                         (PPL (PP  (PREP "for ")
                                   (NP  (PRO "me.")))))))))
```

21. This stamp was bought at the office by I for me.

```
No successful parses!
```

22. This stamp was bought at the office by you for I.

```
No successful parses!
```

23. This stamp was bought by you at the office.

```
(S   (NP1 (NP  (DET "This ")
               (N "stamp ")))
     (V "was ")
     (VP  (V "bought ")
          (VPtl(PPL (PP  (PREP "by ")
                         (NP  (PRO "you ")))
                    (PPL (PP  (PREP "at ")
                              (NP  (ART "the ")
                                   (N "office.")))))))))
```

24. For her to argue can be a pleasure.

```
(S   (PP  (PREP "For ")
          (NP  (PRO "her ")))
     (S   (NP1 (TO "to ")
```

```
          (VP  (V "argue ")))
     (MOD "can ")
     (VP  (V "be ")
          (VPtl(NP  (ART "a ")
                    (N "pleasure."))
               (PPL {PP  (PREP "For ")
                         (NP  (PRO "her "))})))))
```

25. For her to argue give pleasure.

```
No successful parses!
```

26. For it to have been bought gives pleasure.

```
(S   (PP  (PREP "For ")
          (NP  (PRO "it ")))
     (S   (NP1 (TO "to ")
               (V "have ")
               (VP  (V "been ")
                    (VP  (V "bought "))))
          (VP  (V "gives ")
               (VPtl(NP  (N "pleasure."))
                    (PPL {PP  (PREP "For ")
                              (NP  (PRO "it "))})))))
```

27. When will it give pleasure?

```
(S   (PP "When ")
     (MOD "will ")
     (NP1 (NP  (PRO "it ")))
     (VP  (V "give ")
          (VPtl(NP  (N "pleasure?"))
               (PPL {PP "When "}))))
```

28. Will the man be given a chance?

```
(S   (MOD "Will ")
     (NP1 (NP  (ART "the ")
               (N "man ")))
     (VP  (V "be ")
          (VP  (V "given ")
               (VPtl(NP  (ART "a ")
                         (N "chance?"))))))
```

29. Did the office buy a stamp?

```
(S   (V "Did ")
     (NP1 (NP  (ART "the ")
               (N "office ")))
     (VP  (V "buy ")
          (VPtl(NP  (ART "a ")
                    (N "stamp?")))))
```

30. Did the office buys a stamp?

```
No successful parses!
```

31. How many stamps were bought?

```
(S   (NP1 (NP  (DET "How many ")
               (N "stamps ")))
     (V "were ")
     (VP  (V "bought?")))
```

Appendix E

Agent log files

These are the agent log files created during the demonstration run given in Chapter 5:

The Q/A-agent log file:

```
Applet begins.
?QA-: QA starts!
Applet starts.
?QA-: Reg.new agent:TA, host:plg.uwaterloo.ca(null)
?QA-: changed info for TA: type=TA port=37899
QA-1: Got new name:QA-1
QA-1:>TA(plg.uwaterloo.ca, 129.97.140.10):
(register :sender QA-1
          :content (agent-info QA-1 QA mercator.math 0)
          :ontology min-ontology :receiver TA :language KIF)
QA-1:>TA(plg.uwaterloo.ca, 129.97.140.10):
(ask-one :sender QA-1
         :content (length (keywords-and subject java lex yacc))
         :ontology min-ontology :receiver TA :reply-with TA.1
         :language KIF)
QA-1:<TA(plg.uwaterloo.ca, 129.97.140.10):
(tell :sender TA :reply-to TA.1
      :content (answer 1) :ontology min-ontology
      :receiver QA-1 :language KIF)
```

The T-agent log file:

```
TA starts: Mon Aug 11 01:26:55 EDT 1997
TA:port:37899 host:plg
TA:Reg.new agent:IA-1, host:plg.uwaterloo.ca(129.97.7.16)
TA:Reg.new agent:LA-1, host:plg.uwaterloo.ca(129.97.7.16)
TA:<IA-1(plg.uwaterloo.ca, 129.97.7.16):
(register :sender IA-1
          :content (agent-info IA-1 IA plg 37900)
          :ontology min-ontology :receiver TA :language KIF)
TA:<LA-1(plg.uwaterloo.ca, 129.97.7.16):
(register :sender LA-1
          :content (agent-info LA-1 LA plg 37902)
          :ontology min-ontology :receiver TA :language KIF)
TA:changed info for IA-1: type=IA port=37900
TA:changed info for LA-1: type=LA port=37902
TA:>IA-1(plg.uwaterloo.ca, 129.97.7.16):
(register-agent :sender TA
          :content (agent-info LA-1 LA plg.uwaterloo.ca 37902)
          :ontology min-ontology :receiver IA-1 :language KIF)
TA:>LA-1(plg.uwaterloo.ca, 129.97.7.16):
(register-agent :sender TA
          :content (agent-info IA-1 IA plg.uwaterloo.ca 37900)
          :ontology min-ontology :receiver LA-1 :language KIF)
TA:Reg.new agent:QA-1,
          host:mercator.math.uwaterloo.ca(129.97.140.145)
TA:<QA-1(mercator.math.uwaterloo.ca, 129.97.140.145):
(register :sender QA-1
          :content (agent-info QA-1 QA mercator.math 0)
          :ontology min-ontology :receiver TA :language KIF)
TA:changed info for QA-1: type=QA port=0
TA:<QA-1(mercator.math.uwaterloo.ca, 129.97.140.145):
(ask-one :sender QA-1
         :content (length (keywords-and subject java lex yacc))
         :ontology min-ontology :receiver TA :reply-with TA.1
         :language KIF)
TA:>IA-1(plg.uwaterloo.ca, 129.97.7.16):
(ask-one :sender TA
         :content (length (keywords-and subject java lex yacc))
         :ontology min-ontology :receiver IA-1 :reply-with IA-1.1
         :language KIF)
TA:<IA-1(plg.uwaterloo.ca, 129.97.7.16):
(tell :sender IA-1 :reply-to IA-1.1
         :content (answer 1) :ontology min-ontology
         :receiver TA :language KIF)
TA:>QA-1(mercator.math.uwaterloo.ca, 129.97.140.145):
(tell :sender TA :reply-to TA.1
```

```
:content (answer 1) :ontology min-ontology
:receiver QA-1 :language KIF)
```

The I-agent log file:

```
IA starts:Mon Aug 11 01:26:57 EDT 1997
IA-1:port:37900 host:plg
IA-1:Reg.new agent:TA, host:plg.uwaterloo.ca(null)
IA-1:changed info for TA: type=TA port=37899
IA-1:>TA(plg.uwaterloo.ca, 129.97.7.16):
(register :sender IA-1
          :content (agent-info IA-1 IA plg 37900)
          :ontology min-ontology :receiver TA :language KIF)
IA-1:<TA(plg.uwaterloo.ca, 129.97.7.16):
(register-agent :sender TA
          :content (agent-info LA-1 LA plg.uwaterloo.ca 37902)
          :ontology min-ontology :receiver IA-1 :language KIF)
IA-1:Reg.new agent:LA-1, host:plg.uwaterloo.ca(null)
IA-1:changed info for LA-1: type=LA port=37902
IA-1:<TA(plg.uwaterloo.ca, 129.97.7.16):
(ask-one :sender TA
         :content (length (keywords-and subject java lex yacc))
         :ontology min-ontology :receiver IA-1 :reply-with IA-1.1
         :language KIF)
IA-1:>LA-1(plg.uwaterloo.ca, 129.97.7.16):
(stream-all :sender IA-1
         :content (keywords-and subject java lex yacc)
         :ontology min-ontology :receiver LA-1 :reply-with LA-1.1
         :language KIF)
IA-1:<LA-1(plg.uwaterloo.ca, 129.97.7.16):
(tell :sender LA-1 :reply-to LA-1.1
         :content (article-outline "SOFTWARE ENGINEER, JAVA, C++,
                             COMPILER, LEX, YACC, PCCTS, UNIX, NT,"
                      "jason@futurelink.sfbaynet.com (Jason Cale)"
                      "ba.jobs.offered" 42)
         :ontology min-ontology :receiver IA-1 :language KIF)
IA-1:<LA-1(plg.uwaterloo.ca, 129.97.7.16):
(eos :sender LA-1 :reply-to LA-1.1 :receiver IA-1)
IA-1:>TA(plg.uwaterloo.ca, 129.97.7.16):
(tell :sender IA-1 :reply-to IA-1.1
         :content (answer 1) :ontology min-ontology :receiver TA
         :language KIF)
```

The L-agent log file:

```
LA starts:Mon Aug 11 01:26:58 EDT 1997
LA-1:port:37902 host:plg
```

```
LA-1:Reg.new agent:TA, host:plg.uwaterloo.ca(null)
LA-1:changed info for TA: type=TA port=37899
LA-1:>TA(plg.uwaterloo.ca, 129.97.7.16):
(register :sender LA-1
          :content (agent-info LA-1 LA plg 37902)
          :ontology min-ontology :receiver TA :language KIF)
LA-1:<TA(plg.uwaterloo.ca, 129.97.7.16):
(register-agent :sender TA
          :content (agent-info IA-1 IA plg.uwaterloo.ca 37900)
          :ontology min-ontology :receiver LA-1 :language KIF)
LA-1:Reg.new agent:IA-1, host:plg.uwaterloo.ca(null)
LA-1:changed info for IA-1: type=IA port=37900
LA-1:IA-1 at plg.uwaterloo.ca(129.97.7.16)
LA-1:<IA-1(plg.uwaterloo.ca, 129.97.7.16):
(stream-all :sender IA-1
          :content (keywords-and subject java lex yacc)
          :ontology min-ontology :receiver LA-1 :reply-with LA-1.1
          :language KIF)
LA-1:Constructed GCL query:
                            ("java"^"lex"^"yacc")<subject
LA-1:>IA-1(plg.uwaterloo.ca, 129.97.7.16):
(tell :sender LA-1 :reply-to LA-1.1
          :content (article-outline "SOFTWARE ENGINEER, JAVA, C++,
                                   COMPILER, LEX, YACC, PCCTS, UNIX, NT,"
                             "jason@futurelink.sfbaynet.com (Jason Cale)"
                             "ba.jobs.offered" 42)
          :ontology min-ontology :receiver IA-1 :language KIF)
LA-1:>IA-1(plg.uwaterloo.ca, 129.97.7.16):
(eos :sender LA-1 :reply-to LA-1.1 :receiver IA-1)
```

Bibliography

[AAT97] A. T. Arampatzis, Koster C. H. A., and T. Tsoris. IRENA: Information retrieval engine based on natural language analysis. In *RIAO'97, Conference Proceedings, Computer-assisted information searching on Internet*, pages 159–175, June 1997.

[ABRS95] J. M. Andreoli, Uwe M. Borghoff, Pareschi Remo, and J. H. Schlichter. Constraint agents for the information age. *Journal of Universal Computer Science*, 1(12), December 1995.

[AGK+97] Natalia Anikina, Valery Golender, Svetlana Kozhukhina, Leonid Vainer, and Bernard Zagatsky. REASON: NLP-based search system for the WWW, 1997. `http://www.lingosense.co.il/reason.htm`.

[All95] James Allen. *Natural Language Understanding.* The Benjamin/Cummings Publishing Company, Inc., 1995.

[Bir95] William P. Birmingham. An agent-based architecture for digital libraries. *D-Lib Magazine*, July 1995. `http://www.dlib.org/dlib/July95/07birmingham.html`.

[Ble97] Michael Bleyer. CEMAS: A concept exchanging multiple agent system for information retrieval on the world wide web, June 1997. `http://lia.univ-savoie.fr/ cemas`.

[BPK+96] Uwe M. Borghoff, Remo Pareschi, Harald Karch, Martina Nohmeier, and Johann H. Schlichter. Constraint-based information gathering for a network publication system. In *Proceedings, PAAM'96*, London, U.K., April 1996.

[CB] Gordon Cormack and Forbes Burkowski. Multitext project. `http://multitext.uwaterloo.ca/`.

[CDR97] CDR, Stanford University. Java agent template (JATLite), 1997.
`http://java.stanford.edu/java_agent/html/`.

[CH95] Anil S. Chakravarthy and Kenneth B. Haase. NetSerf: Using semantic knowledge to find Internet information archives. In *Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 4–11, Seattle, Washington, USA, July 1995.

[DLNPW95] Keith Decker, Victor Lesser, M. V. Nagendra Prasad, and Thomas Wagner. MACRON: An architecture for multi-agent cooperative information gathering. CIG home page, November 1995.

[FG96] Stan Franklin and Art Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In *Proceedings of the third International Workshop on Agent Theories, Architectures, and Languages*. Springer-Verlag, 1996.

[Fin97a] Tim Finin. KIF—knowledge interchange format, 1997.
`http://www.cs.umbc.edu/kse/kif/`.

[Fin97b] Tim Finin. KQML—knowledge query and manipulation language, 1997.
`http://www.cs.umbc.edu/kqml/`.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[Kar96] Grigoris J. Karakoulas. The SIGMA project: Market-based agents for intelligent information access. *Canadian Artificial Intelligence/Intelligence Artificielle au Canada*, 40:25–27, Autumn 1996.

[KH95] D. Kuokka and L. Harada. Matchmaking for information agents. In *Proceedings AAAI Spring Symposium on Information Gathering from Heterogenous, Distributed Environments*, pages 672–678, 1995.

[LF97] Yannis Labrou and Tim Finin. TR CS-97-03, a proposal for a new KQML specification. Technical report, Computer Science and Electrical Engineering Department, University of Maryland Baltimore County, February 1997.

[Lua97] Xiaocheng Luan. JKP—Java KIF parser, 1997. `http://www.cs.umbc.edu/kse/kif/jkp/`.

[Mau91a] Michael L. Mauldin. *Conceptual Information Retrieval; A Case Study in Adaptive Partial Parsing.* Kluwer Academic Publishers, 1991.

[Mau91b] Michael L. Mauldin. Retrieval performance in FERRET, 1991. `http://fuzine.mt.cs.cmu.edu/mlm/sigir91.html`.

[Mea] George A. Miller and et al. WordNet home page. `http://www.cogsci.princeton.edu/~wn/`.

[Mil95] George A. Miller. WordNet: A lexical database for English. *Communications of the ACM*, 38(11):39–41, November 1995.

[Mol97] Dustin Mollo. Perl language home page, 1997. `http://www.perl.com/index.html`.

[NPLL95] M. V. Nagendra Prasad, Victor R. Lesser, and Susan Lander. Retrieval and reasoning in distributed case bases. Technical Report UMass Computer Science 95-27, University of Massachusetts, 1995.

[ONPL94] Tim Oates, M. V. Nagendra Prasad, and Victor R. Lesser. Cooperative information gathering: A distributed problem solving approach. Technical Report UMass Computer Science 94-66, University of Massachusetts, 1994.

[RN95] Stuart J. Russell and Peter Norvig. *Artificial Intelligence, A Modern Approach.* Prentice Hall, Englewood Cliffs, 1995.

[Sho93] Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, 1993.

[Sob] Ian Soboroff. Agent-based information retrieval. `http://www.cs.umbc.edu/abir/`.

[Sun97] Sun Microsystems, Inc. Java home page, 1997. `http://java.sun.com/`.

[Wag] Tom Wagner. CIG searchbots. `http://dis.cs.umass.edu/research/searchbots.html`.