Modular Knowledge Integration Interface

for the START Natural Language Engine

by

Levent M. Talgar

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering and

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

February 4, 1998

© 1998 Levent M. Talgar. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

Author

Department of Electrical Engineering and Computer Science
February 4, 1998

Certified by

Boris Katz
Thesis Supervisor

Accepted by

Frederic R. Morgenthaler
Chairman, Department Committee on Graduate Theses
Modular Knowledge Integration Interface
for the START Natural Language Engine
by
Levent M. Talgar

Submitted to the
Department of Electrical Engineering and Computer Science

February 4, 1998

In Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering and
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

This modular knowledge integration interface assists the natural language engine, START, in answering English language queries regarding a wide variety of knowledge sources external to the base system. It accomplishes this task both by aiding in the preprocessing of queries and by providing a language rich with primitives that retrieve external data from sources such as the World Wide Web, local databases, and other Internet facilities. Preprocessing is implemented by matching queries against a large SQL-based database server. The language uses a LISP-like syntax and is interpreted on a TCP-based server.

Thesis Supervisor: Boris Katz
Title: Principal Research Scientist
Acknowledgements

I am grateful to Boris Katz for providing the opportunity and the resources for working on a project as exciting and groundbreaking as START.

I'd also like to thank:

Deniz Yuret – for introducing me to the project and providing valuable guidance throughout the endeavor.

Adnan Ilik – for invaluable assistance, dedication, and expertise through the many long days and nights of work.

Infolab Group – for their great input and teamwork in making the project successful.

And of course:

Cigdem Penbeci – for her endless love and support.

Steven Leung – for his wisdom and valuable advice.

My parents – for always pointing out the light at the end of the tunnel when times were rough.
Introduction

There is no denying that we've made the leap from the Industrial Age and are now firmly planted within the Age of Information. This last decade of the 20th century alone has introduced the world to the vast strengths and capabilities that arise from a globally distributed network of computers. The Internet has enabled people from all parts of the planet to share and distribute information about anything and everything ranging in media from text to pictures to audio and even video. In addition, advances in data storage technology have led to the ability to store whole volumes, if not libraries, of books and information in a very compact, easily accessible and transportable form.

Having so much knowledge—or power, however one perceives it—from the perspective of those seeking to develop intelligent systems, provides opportunities unimaginable a few decades ago. To develop an omniscient system or “being” could only be perceived as blasphemy a short while ago. Today though, the world is different. By combining years of research into natural language and artificial intelligence with the vast sea of knowledge now before us, such “beings” are not only feasible, but exist in their infant stages.

It seems as though we now have most of the pieces to the puzzle. Or do we? Integrating such systems create great challenges in and of themselves. Then, when we consider that our language and reasoning systems are still in their adolescence, we realize that such a “being” might still be beyond reach.

START, our natural language system, has made great strides in the right direction and is arguably the best system of its kind. Our quest in merging vast knowledge sources to such natural language systems will be an invaluable tool, not just today in research labs, but tomorrow for everybody who seeks to learn and discover.

START

Since 1985, Boris Katz has been developing and enhancing the START natural language engine at the MIT Artificial Intelligence Laboratory. START, which stands for SynTactic Analysis using Reversible Transformations, is a system designed to answer
questions that are posed to it in natural language. It parses incoming questions, matches the queries created from the parse trees against its knowledge base, and presents the appropriate information segments to the user. Since 1993, START has been accessible via the World Wide Web and has answered hundreds of thousands of queries. [3] [6]

**Overview**

The underlying mechanisms that START uses in parsing and interpreting English sentences are quite complex. Instead of delving into them, below is an excerpt from START's Web page that presents a simplified description of its ability to answer questions about different types of information:

The natural language processing component of START consists of two modules that share the same grammar. The understanding module analyzes English text and produces a knowledge base that encodes information found in the text. Given an appropriate segment of the knowledge base, the generating module produces English sentences. Used in conjunction with the technique of natural language annotation, these modules put the power of sentence-level natural language processing to use in the service of multi-media information access.

A key technique called "natural language annotation" helps START connect information seekers to information sources. This technique employs natural language sentences and phrases -- annotations -- as descriptions of content that are associated with information segments at various granularities. An information segment is retrieved when its annotation matches an input question. This method allows START to handle all variety of media, including text, diagrams, images, video and audio clips, data sets, Web pages, and others.

**End-to-End Example**

An effective method for revealing how START works is to walk through an example. When the system receives a query, for example "Who directed Gone With the Wind?", it first passes through the preprocessor. The preprocessor will identify that Gone With the Wind is a symbol of type movie. Then the query plus this additional information is passed on to the natural language engine.

---

1 For a detailed discussion, refer to [4].
The natural engine will then first parse the query without the preprocessor information. In this case, when our query is parsed, START will realize that the two verbs "directed" and "gone" do not make sense next to each other. It will then try a variety of rules to alleviate the problem, such as reassigning parts of speech when possible. Since in our scenario this is not possible, START then reverts to using the information passed from the preprocessor. This tells START that "Gone With the Wind" is a symbol and that it should be analyzed as such. Hence, it essentially sees "Who directed <symbol - type: movie>.

Now that the sentence makes grammatical sense, START will parse it and search for the answer. This is done by matching the query against its knowledge base. Since START does not explicitly know the answer to this particular query, it will recognize that it has the ability to find information about elements that can be "directed". Finally START retrieves the correct information (using the new interface defined by this document), and returns it to the user in the correct format.

Sample Questions / Answers

Here are the responses to a few basic questions as answered by START:

WHERE IS THE MIT ARTIFICIAL INTELLIGENCE LABORATORY?
The MIT Artificial Intelligence Laboratory is located in Cambridge, Massachusetts. The Laboratory's mailing address is -- MIT AI Laboratory, 545 Technology Square, Cambridge, MA 02139.

WHAT IS THE DISTANCE BETWEEN ISTANBUL AND CAMBRIDGE?
The distance between Istanbul Turkey and Cambridge Massachusetts is 4819 miles (7773 kilometers).

WHAT CITIES ARE WITHIN 100 MILES OF FRANKFURT?
In Germany, the following cities are within 100 miles of Frankfurt:
Mainz Germany is 15 miles (24 kilometers) from Frankfurt.
Koblenz Germany is 50 miles (81 kilometers) from Frankfurt.
Heidelberg Germany is 52 miles (84 kilometers) from Frankfurt.
Bonn Germany is 63 miles (102 kilometers) from Frankfurt.
Cologne Germany is 87 miles (140 kilometers) from Frankfurt.
Objective

The goal of this endeavor was to create a modular system to add symbols and information to START’s base knowledge set. Abstracting this functionality out of the core natural language engine was one of the project’s guiding principles. Under the old implementation for dealing with external data sources, non-consistent code segments were distributed within START’s code base. They would provide such functionality as retrieving Web pages, calculating certain data, or retrieving stored information from different hash tables. There existed no uniform interface for adding new functionality.

Besides its own vocabulary of words and symbols, START had no way of recognizing the vast number of proper nouns and symbols that humans can instantly identify, such as movies, people, songs, etc. We felt that such recognition would add tremendous value to our system.

Hence our first objective was to implement a vast database of symbols for START to refer to. We didn’t have many concrete ideas in mind except that we knew this system needed to be extremely fast. Next, to complement this new recognition ability, we wanted to create a language or set of languages that would abstract data retrieval away from START and allow for much easier extensibility.

Some of the concepts utilized in this project are based on ideas and projects that have already been considered and built within the Infolab group. Further reference to these earlier systems will be discussed as we go into further depth.

Design Decisions

Once the basic goals had been established, some initial research was done to discover projects with a focus similar to the enhancements that we proposed. Though there were very few groups with a focus as narrow as ours, we found quite a number of projects dealing with the problems of information retrieval and the compilation of knowledge bases. Towards the end of this document, a brief discussion can be found of some of these projects and how we hope to eventually interface with these new and more exhaustive knowledge bases.

The different modules that make up our design are discussed below:
**Pre-processing**

Pre-processing refers to the subsystem within START that analyzes the contents of a query before that query is presented to the main natural language engine. Under the current implementation, START is just fed the text query directly. Our goal was to develop a way in which we could both identify certain elements within the query that would help START parse the query and also identify certain symbols to START such that it would have an easier time providing a coherent answer to the user.

This idea of preprocessing was first implemented within the Infolab group by Jimmy Lin, Rebecca Schulman, and Deniz Yuret in an effort to aid START in recognizing names, titles, numerals, addresses, and other special phrases using a set of heuristics. This project, named Blitz, is discussed further in the preprocessing chapter. Though Blitz had yet to be integrated with START, they had developed a concise interface through which both the query and a set of pre-defined statements would be sent to the engine. Since code for handling these types of statements was already being developed, we decided to be consistent and return symbols recognized by the database in the same statement format, called “frames.”

Another important consideration in the feasibility of such a pre-processing feature was its performance. It needed to be extremely fast so as not to be a bottleneck in terms of speed. To make sure that such an idea was within reasonable speed tolerance (\(< 0.5\) seconds), we did some initial tests with a preliminary database populated with about 200,000 symbols. Our results indicated that we were within our pre-defined performance tolerance.

**Layering**

Since building a modular system with very well defined levels of abstraction was a primary design goal, we decided that our system would be more effective if layered. Instead of dumping all of the functionality into a big heap, each layer was to focus on different problems that needed to be solved. The big issue though was in determining where these layers naturally lay and how many were really needed.

The initial design called for much of the logic to actually reside within the natural language engine and for it to do the deductive reasoning. In this type of a system, the
only time one would need to make a call outside of the natural language (NL) engine would be to retrieve some type of data. In building the data retrieval mechanism external to the NL engine, we achieved our goal of isolating these functions in an abstracted and modular manner. Much of our initial work was based on this two-layer concept.

While working on the project, another M. Eng. student within our group, Maciej Stachowiak, was concurrently considering ideas to append more functionality to the logic START uses in its manipulation and retrieval of external data. In discussing our different ideas, we realized that it would probably be advantageous to also extract the internal START code that he had been working on and instead develop a multi-layered data retrieval system as illustrated below.
**Inter-layer communication**

Protocols are needed to pass data through the different layers that are displayed in the above diagram. The interaction between the preprocessor and the NL engine has already been mentioned above. A thorough analysis of it will be discussed in the section on preprocessing below. The abstraction between the NL engine and "Language 1" is being developed by Maciej Stachowiak and thus is not defined here.

This brings us to the interface that we developed for "Language 1" to request data using "Language 2". As a result of different implementation decisions regarding computer platforms, the "Language 2" interpreter is located on a different machine from
the rest of the components above it in the diagram. Hence, the interface between these two languages necessarily involves some kind of network communication. We considered several options, such as setting up a Web interface or using some RPC (Remote Procedure Call) mechanism. We finally decided upon writing a TCP (Transfer Control Protocol) port based server. This would allow "Language 1" to directly send and receive data by opening a socket to the port our server was running on. More information is provided in the chapter on implementation decisions.

**Language Design**

We considered for quite a while the format to be used for "Language 2." We needed to make it simple enough to be easily interpreted but we wanted it to have enough power to express any type of query that might be needed. After some initial thought, we narrowed our options down to two different formats.

The first was a very LISP-like language using Polish notation of the form (function name > parameter 1 > parameter 2 > ...). This form had the definite advantage that it was easy to parse and we were already familiar with writing basic interpreters for such functions. We had reservations though as to whether it would be versatile enough to provide all of the functionality we needed.

The second format we considered was creating a language mimicking the standardized database querying language SQL (Structured Query Language). This language would definitely provide all of the power we'd need, but we had serious doubts about the ease with which we could build a robust interpreter for it.

In the end, it was the ability to easily implement recursion and embedding into the LISP-like language that led us to choose it. This would allow us to easily answer such questions as "list all the countries that border the country with the lowest GDP." Though we opted not to use SQL as our native language, we most definitely made great use of it in making calls to our database of symbols, as well as other data sources that we eventually implemented as local databases.
Implementation Decisions

In implementing this project, many of the choices we made were atypical for our research group. Though sticking to convention would have worked, there were several opportunities for us to benefit from the strengths of more commercial applications, especially in the arena of databases.

Platform

We opted to implement our system on the Intel x86 architecture running the Microsoft® Windows NT Server operating system. There were a few basic reasons for our decision and more support of it will become apparent as the rest of the implementation decisions are discussed below. The first reason, as mentioned above, was the wide variety of robust commercial applications and authoring environments that were available to us. Second, a comparable UNIX workstation would cost on the order of 2 to 3 times the price of the PC. Finally, a major incentive for using this architecture was my background, comfort level, and programming experience with PC’s.

Data storage

The biggest advantage to having utilized NT becomes apparent in how we opted to store our large database of symbols. Earlier attempts at creating similar lists of symbols were performed in our group using optimized hash tables in C. Though they were very fast, they had the limitations that they were difficult to modify and also needed to be rebuilt every time a single change was made to them. Since a major goal of ours was to make it very easy to add additional resources to our system, this was a definite problem.

After considering different ideas for modifying the hash table more easily, we decided instead to go with a commercial system. Since we wanted to use a standardized package that was reasonably easy to modify and extremely efficient, we chose to use the relational database program Microsoft® SQL Server. It definitely had a superset of all the functionality that we required but more importantly had the power and speed that were essential to our project. More discussion of SQL Server can be found below in the chapter on database design.
**Authoring environment**

Given that we were working on a Windows platform, we could choose from a wide variety of programming languages. We sought an environment that enabled us to create relatively small programs both quickly and easily. We also sought a language that natively supported TCP sockets.

We finally decided on Microsoft® Visual Basic for most of our development. It worked quite well in that all of the networking functionality was trivial to implement plus it allowed us to directly access our SQL Server databases using ODBC (Open Database Connectivity), Windows' native database connectivity.

**Web interface**

We chose Microsoft® Internet Information Server (IIS) for our Web server. Our initial work on the project had a much larger segment designed for using the Web. With time though, as we realized that there were definite performance degradations in executing things over the Web, we migrated most of that code to Visual Basic.

We still have some basic utilities on our Web server that will allow users to test a query against the symbol database in the same way the symbol database is used by the preprocessor. Users can also use that interface to easily add, delete, or modify individual symbols in the database.

**Database Design**

A vital aspect of this project revolved around databases, and especially our symbol database. A lot of time and analysis went into designing an efficient database architecture that would support exactly what we needed without being overly complex. Below is a discussion of the different phases of design and understanding we went through in regards to our databases.

**SQL**

Part of the power of SQL actually derives from the simplicity of its syntax. The main Transact-SQL (our particular dialect of SQL) functions that we used were INSERT (to add rows to our tables), UPDATE (to modify rows), DELETE (to delete rows),
CREATE TABLE (to create tables within a database), and SELECT (to display those rows that matched our given criteria) [7].

We achieved impressive results even during our initial endeavors in creating our database when we used very simplistic and inefficient approaches to creating and populating it. As the project evolved, we improved by creating more advanced queries and more optimized databases [11]. More detail can be found below.

**Design & Creation**

Our first attempt at creating a database involved using SQL Server's Enterprise Manager graphical user interface. We first had to allocate physical hard disk space to SQL Server so that it could format it for database storage. Creating a database in that space was then quite simple.

In our first implementation of the symbol database, we had written a Web application using Active Server Pages (ASP) to add symbols to the database. A big concern was checking the validity of each symbol added to the database. For example, before we would add it, we'd make sure that there were no other instances of the symbol. Of course, as the database grew to a size of over 100,000 symbols, having to check each of the existing symbols caused a strain on the system and was reflected in slower performance. Another reason for our lagging speed in adding symbols was the fact that we were creating a new database transaction for each symbol added. The overhead processing time for each transaction is huge compared to the simple process of just adding that symbol. In time we learned that we could add multiple INSERT statements within a single transaction and performance recovered.

**Populating the Database**

As mentioned above, we started out by creating a database that was very inefficient, both in storing symbols as well as in retrieving them. It was a two table system. The first table had two columns: the UID (unique ID) and symbol columns. The second table again stored the UID but also stored the source type for the symbol associated with the UID. At first this seemed like a good idea in that using this scheme, we could easily check for identical symbol entries and confirm if they were of the same type. We soon realized the inefficiencies of our model.
Our next approach was to implement a much simpler table design. This allowed us to propagate the problem of dealing with multiple symbol entries to the point when we actually queried the database. This made life both a lot easier as well as a lot faster! We had only one main table, with a direct relationship between the symbol column and the source column.

By this time, we had also rewritten our symbol adder program in Visual Basic. It had a GUI interface and took as parameters an input filename, the source the symbols were retrieved from, the symbol type, and an output filename in which our SQL transaction could be stored. Its interface can be seen in the figure below. The input file was just a simple text file with all of the symbols delimited by carriage returns. For each symbol added, a line would be created in the output file of the form: INSERT INTO Sym(Symbol, Source) VALUES ("<symbol>", "<source>") where <symbol> and <source> are replaced by their respective values.

![Symbol List to SQL Utility](image)

**Figure 2: Symbol List to SQL Utility**

When we were ready to append these SQL files to our database, we would execute them in SQL Server just as we would any other query. The advantage of storing these SQL statements in files as opposed to directly executing them through our program was that should any of the symbols be corrupted or should the database need to be recreated, all of the work is already done. All one would need to do is re-execute all of the SQL files.

As we mentioned above, we wanted to make sure that each INSERT statement was not executed as a separate transaction itself. Hence we implemented our program...
such that the GO statement delimited every 100 INSERT statements in our output file. This tells SQL Server to consider all statements between the GO's as a single transaction. By using this idea of lumping multiple INSERTS into the same transaction, our performance in terms of database creation and symbol appendage skyrocketed.

Optimization

Besides the improvements we got in rethinking our database architecture, we were also able to tweak SQL Server to help it optimize based on certain parameters. For each table within a database, one column can be defined as the primary key. This means that when SQL Server stores a certain row, it uses the row's primary key as part of the storage algorithm. Hence, should you later try to search for that row using the key, your performance will be considerably faster. This concept is very similar to hashing in a hash table except that SQL Server optimizes based on an algorithm involving each of the characters in the symbol.

SQL Server is also great about optimizing on the fly. Based on the queries that the database is getting, it will dynamically swap pages of the database into and out of RAM not only by caching, but also by using different techniques of anticipatory retrievals. It caches as much of the database as it can but how much is dependent on the saturation level of each page. We found that if we increased the saturation on each page, the time it took for us to add symbols to the database went up, but our query times to retrieve symbols definitely went down when doing large searches against the database.

Extensibility

As stated in our objectives, we strove to build a system that would allow its administrator to easily add new functionality, be it in the form of new symbols or new primitive functions to our language. As described above, with both our Web-based and NT-based utilities, we have met our objective with regards to the modification and addition of new symbols to our database.

Local Data Sources

Besides its functionality as our symbol database server, we also utilized SQL Server to parse and store local data sources. We initially created these new databases as a
basic way of storing and retrieving simple text data. Once we had parsed a few text files into databases, we were very pleasantly surprised at the speed and complexity with which we could query our data.

Hence, we took on the task of creating a database to store all of the CIA's 1996 World Factbook\(^3\). The Factbook contains information about 261 countries. Specifically, it contains 133 fields for each country ranging from numerical values for population and land area to pure text describing their economies and histories. Using SQL Server, we were not only able to query specific information about any country, but also perform different functions such as taking the maximum or minimum of any value or searching for a substring in any of the text fields.

We eventually wrote a program in Visual Basic for our own debugging and enjoyment purposes that would query the database and return the results in an Excel-like spreadsheet. Below is an illustration of the interface:

![CIA Database Retrieval Application](image)

Figure 3: CIA Database Retrieval Application

\(^3\) [http://www.odci.gov/cia/publications/nsolo/wfb-all.htm](http://www.odci.gov/cia/publications/nsolo/wfb-all.htm)
Preprocessing

This first stage in the process of understanding an English language question, we believe, will be an extremely important step in enabling START to understand a much larger range of questions than it currently does. By identifying special phrases and symbols through a variety of heuristics and simple lookups, we present vital information to START and improve its comprehension ability. Below is a basic discussion of the first-generation pre-processor developed in our group, as well as the additional functionality that our new system will provide.

Blitz

As stated before, Blitz was a project with the aim that along with the query, it would pass to START a set of statements, which we called frames. Within each frame was information regarding different words or phrases that were identified as being either proper nouns or specific information representation forms, such as times and dates.

While the authors were developing this system, they defined a very specific frame definition that can be seen below:

[frame-type "name" :synonym "synonym" :field1 "symbol1" :field2 "symbol2" ...
:span (begin end)]

[frame-type (required) the type of frame
name (required) the central "pivot" of the frame
synonym (optional) synonym for the name. name is always the string found within the text, but the synonym can be any other string.
fields, symbols (optional) field names and their corresponding values.
span (required) begin is the character position of the first character in name, where it was found. end is the last character position.

An example of Blitz in use can be seen below. In response to the query, "What two items were invented on October 15th?" the system returns:

(date "October 15" :month October :span (32 41))
(propernoun "October" :span (32 38))
(propernoun "What" :span (0 3))
(number "15th" :value 15 :notation ordinal :span (40 43))
Blitz was initially implemented as a stand-alone C program that would be called remotely with a query as an argument. Because of the difficulty in sometimes closing these remote processes, a Blitz server was written using Visual Basic that would act as a liaison between the C code and the network. Not only did this alleviate the old networking problems, it also acted as a log of calls to Blitz that helped quite a bit during debugging efforts. A picture of its interface is below (the numbers at the top represent the status of the current connection and the network):

![Blitz Server Interface](image)

**Figure 4: Blitz Server**

**Our system**

Though Blitz did a good job identifying particular elements within queries, we wanted to take advantage of our vast symbol database in recognizing and identifying the type of all symbols within a given query. Hence, we developed a basic system that would look up each word in the query against the database. Though we were pleased with the lookup speeds, we realized that just this functionality was not sufficient.
The biggest problem with this new system was that it only matched one-word symbols, since we were only looking up one word at a time. In response, we came up with the idea to search all possible subsequences of the words in the query. So, if we were to give it the query "This is a test", it would try "This is a test", "This is a", "This is", "This", "is a test", "is a", "is", "a test", "a", and "test". Though this requires order \( n^2 \) lookups, it performs quite well where \( n \), the number of words, is small. We also defined a concept of depth, which refers to the maximum word-count string that we would try to match against the database. For example, for a search with a depth of four, "a b c d e f" would match the subsequences "a b c d", "a b c", "a b", "a", "b c d e", "b c d", "b c", ..., "f." After doing some testing using this algorithm, we found that a depth of four actually worked well in that it provided the best balance between speed and coverage. Since there are obviously symbols of word length greater than four, we also have the system check the database with each of the subsequences of maximum length with a wild card appended to them. Hence, for the example above, we would also query "a b c d *", "b c d e *", and "c d e f *", where * is a wild card, just in case the database contained the symbol "b c d e f".

For testing purposes, we created a simple Visual Basic application that would run the algorithm described above and display the results in a simple list. The figure below illustrates its interface:

![Query Analyzer](image)

Figure 5: Query Analyzer
As shown above, in response to the query, “Who directed Gone with the Wind”, the system listed all of the symbols (with their source and type information appended to them) found in the database.

When we were confident that our algorithm worked, we then modified the program to return the results in the same frame format used in Blitz and described above.

**Integration with START**

Both Blitz and our Query Analyzer work well independently, but in fact it is advantageous to combine their results when integrating them into START, in order to eliminate duplication and reduce external calls and network traffic. START can immediately utilize symbols that match its native types, but its internal architecture must be modified slightly to reap the full benefits of other symbols. Much of the strength of knowing symbol types will become more apparent as Language 1’s interface becomes clearer and START’s methodology for using Language 1 gets better defined.

**“Language 2” (Our Language)**

This section focuses on the low-level language that we implemented as an abstraction for extracting data from a myriad of different sources. As stated before, each of its primitives are functions that return data of a particular type and from a particular source. For example, the IMDB (Internet Movie Database) function would take certain parameters, form and retrieve the query in the syntax that the IMDB uses, and then parse the response for the correct piece of information and send it back to the calling user or process.

**Syntax**

Our syntax is very similar to a LISP style language, with all statements enclosed in parentheses. The reasoning behind our decision to use such a format was discussed above in the section on design decisions. Not only are the commands parenthesized, but all responses from the interpreter are as well. Though this works well in that it defines a consistent model, its value becomes more apparent when we discuss statement embedding and error-handling below.
A few examples of the system at work are found below. Notice that the
commands always take the form: (<function name> <parameter 1> <parameter 2> ...).
In the IMDB example, the parameter directedby tells the function to search for the
director of the second parameter, “Gone with the Wind.” Whenever a parameter is a
string that is being searched against, it must be enclosed in quotes. This does not mean
that they are matched directly based on case though as we implemented most functions
such that they are case insensitive.

(IMDB directedby "Gone with the Wind")
("George Cukor" "Victor Fleming" "William Cameron Menzies" "Sam
Wood")

(MIT email "Boris Katz")
("boris@AI.MIT.EDU")

Interpreter

Our interpreter was implemented in a format somewhat similar to the Scheme
interpreter developed at MIT. We used the Visual Basic authoring environment to
develop it. In its current state, it can be accessed both from a GUI interface and over a
TCP socket.

Once it initializes itself and receives an input, the interpreter first parses the string
to check for consistency in parentheses and syntax. If no errors are detected, it then calls
separate subroutines (or functions) based on the first element within the parentheses. The
section on embedding below discusses how the system handles functions embedded
within functions. When the subroutine finishes executing, it passes the result back to the
main parser, which, in turn, returns the answer in the correct parenthetical format.

Error handling

Should an error occur at any point while the interpreter is running, the error is
propagated back to the parser and is returned in the format “(error “<reason for error>”).”
Errors can occur either in the parsing, such as with parenthesis mismatches, or in the
subroutines themselves. Such examples can be seen below:

(Acronym "asdfasd")
(error "Acronym not found")
Embedding

A particularly powerful aspect of our language is our ability to imbed functions within functions recursively. This enables the user to constrict the domain from which a solution can be picked. For example, one might want to know the country with the largest land area of all countries where French is spoken. This feature was implemented by using a procedure of parsing the query and assigning depths to each function depending on the number of parentheses it was embedded within. Then, one by one, all of the functions, going from maximum depth to depth zero, are evaluated. At each smaller depth, the result of the previous function restricts the domain on the current function. The query mentioned above is illustrated below in the sequence of events that our interpreter would go through in executing it.

```
(CIA MAX land_area (CIA MATCH languages "french"))
$$1 = (CIA languages "french") \rightarrow ("Canada" "France" "Switzerland" ...)
(CIA MAX land_area $$1)
(CIA MAX land_area ("Canada" "France" "Switzerland" ...))
("Canada")
```

The parser first isolates the inner function and assigns its value as $$1. Until $$1 is reduced, the CIA MAX function can not execute. Once it has the list of those French speaking countries, they become the new domain for the CIA MAX function called with the parameter land_area.

Network usage

Our interpreter accesses the network in two different ways. First, it acts as a server waiting for requests over TCP. Should any external client send a request and query to its port, it will begin parsing the query and send back an answer. Our first iteration of this server was single threaded. This meant that only one client program could connect to it at a time. We later modified it to a multi-threaded architecture, in case multiple requests are sent to the system by “Language 1.”
The second reason why the interpreter might access the network is to retrieve data from the Web or other Internet facilities. In this scenario, another socket is opened to the required port on some external machine and a query is sent. This connection stays active until either a response is retrieved or our system times out. Should it time out, an error is then propagated back to the process calling the interpreter.

**External database handling**

With the vast number of data sources out there on the Web today, it would seem that retrieving data would be easy. Unfortunately, most systems today have safeguards against Web crawlers and robots that sometimes make our lives miserable in terms of retrieving data. Many a time throughout the course of this project, we found ourselves writing clever hacks to get access to the precious knowledge we sought so desperately.

In our current implementation, as a proof of concept, we have provided primitives in our language that retrieve external Web data from the Internet Movie Database\(^4\), an acronym server\(^5\), and the US Postal Service's database of Zip Codes/Cities\(^6\). Examples of these primitives can be found in the syntax section above and in the example below:

```
(Acronym "CCCP")
("Union of Soviet Socialist Republics [Cyrilic alphabet equivalent of USSR]" "[Russian for] Soyuz Sovietskikh Sotsialisticheskikh Respublik")
```

**Local Database handling**

When a primitive that accesses local data is called, another socket need not be opened. Depending on what the source is, the subroutine accessing it will either query a database, read text from a certain local file, or retrieve data from some removable media, such as CD-ROM. All of the local data primitives that we currently support are stored within a SQL database. As of now we have the CIA primitive and the States primitive. Examples of CIA are shown below:

```
(CIA MAX population)
("China")

(CIA MAX land_area (CIA languages "French"))
("Canada")
```

---

\(^4\) [http://us.imdb.com/](http://us.imdb.com/)

\(^5\) [http://www.ucc.ie/acronym/](http://www.ucc.ie/acronym/)

\(^6\) [http://www.usps.gov/ncsc/](http://www.usps.gov/ncsc/)
(STATES MAX population)
("California")

**Extensibility**

Again, we felt it important to stand by our design philosophy of creating a system that would be easily expandable. With the way our primitives have been designed in Visual Basic, adding a new one is quite simple. It's just a matter of pattern matching. Using the GUI development environment, one can modify one of our template functions (either for local or external data sources) or with enough familiarity one can write a new function that would be recognized by the parser.

In this sense, we've solved the difficult exercise of modifying START's LISP code to implement data retrieval procedures.

**"Language 1"**

At this time, because the implementation of "Language 1" has not been clearly defined, its interface with the START NL Engine is not available. Regardless, discussions have already taken place to consider certain interface modifications to "Language 2" that will make its interface with "Language 1" more consistent and fluid. These will involve modifying the current interpreter to accept queries of the general form ( <universal function> <database name> <parameter 1> <parameter 2> ...). The universal function will be one of a few primitives including Select (to return a list of data), Invert (to return in the inversion of the query), Max/Min (obvious), Match (to match a substring within the data), or Greater/Than/LessThan (obvious). Though these changes are not reflected in this particular version of this project, work is currently underway by a group member to modify the system to this new interface.

**Related Research**

Though our focus was very specific in our quest to enhance START's comprehension and information retrieval system, other research groups around the globe are working towards the goal of compiling, customizing, standardizing, storing, and
retrieving all types and sources of information. Below is a discussion of a few groups who have made progress toward these goals.

**The Knowledge Sharing Effort**

This project, sponsored by several government organizations such as the Advanced Research Projects Agency (ARPA), the Air Force Office of Scientific Research (AFOSR), the National Science Foundation (NSF), and the Cooperation for National Research Initiative (NRI), is an initiative "to develop the technical infrastructure and conventions to support the sharing of knowledge and information."\(^7\) [8] [9]

This group realized that many different institutions were re-inventing the wheel in terms of knowledge base acquisition, representation, and utilization. They defined four complementary areas in which the development of common, agreed-upon conventions would enhance leverage between individual research efforts. Ramesh Patil, in his paper entitled "Coordination of Evolving Conventions; Enabling Sharing of Knowledge," states that these areas are:

1. Mechanisms for translation between knowledge bases represented in different languages;
2. Common versions of languages and reasoning modules within families of representational paradigm;
3. Protocols for communication between separate knowledge-based modules, as well as between knowledge-based systems and databases; and,
4. Libraries of "ontologies," i.e., pre-fabricated foundations for application-specific knowledge bases in a particular topic area.

Efforts such as those of the Knowledge Sharing Effort will help in creating an organized domain of knowledge from the chaos that exists today. Currently, very few of these data sources are organized. In time, with the different protocols and mechanisms they define, we will be able to connect START to vast arrays of knowledge bases and ontologies ranging from medical systems to databases of political philosophy. To compensate today, the power behind our proposed system is its ability to access non-uniform data from a variety of sources without the organization that the Knowledge Sharing Effort is proposing.

---

\(^7\) Quote and background information retrieved from the URL: http://www.isi.edu/isd/KRSharing/.
This project, just as the Knowledge-Sharing Effort, focuses on the importance of knowledge bases. Instead of organizing already existing systems, EXPECT provides an environment to aid in the knowledge acquisition, maintenance, and documentation of new knowledge based systems. They feel that though such tools exist today, they are often too complex and difficult to use by those experts constructing new knowledge systems.\footnote{For more information about EXPECT, see URL: http://www.isi.edu/expect/expect-homepage.html}

The heart of the EXPECT architecture is a component that they call an Automatic Program Writer (APW). It combines both domain facts and general problem solving principles to produce a knowledge-based system. The APW can be thought of as a knowledge compiler that creates an efficient representation of the knowledge as well as storing a design history based on its analysis. Whenever a concrete problem is then addressed, an answer is retrieved by instantiating the design history.

At this point, START is not connected with nearly as many of the vast data and logic systems that we'd like. Eventually, we hope to integrate advanced knowledge-systems, as those designed by EXPECT, to aid START in answering questions involving more detailed logic or analysis.

\textbf{Intelligent Integration of Information Technology (I3)}

The I3\footnote{For more information about the I3, see http://mole.dc.isx.com/I3/} is focusing very closely on ways in which raw or scattered data can be organized and subsequently easily retrieved by applications and end users alike. By extracting, integrating, and abstracting information from sources ranging from databases, knowledge bases, sensor-based subsystems, and simulations, they transform this data into a uniform interface to yield relevant information for future use.\footnote{For more information about the I3, see http://mole.dc.isx.com/I3/}

Again, information that has been clearly organized and classified as the I3 has done will play a vital role in simplifying the interface between START and new sources of knowledge. With time, hopefully all such data sources will be compliant to some standard, maybe even those defined by the Knowledge Sharing Effort.
HAWK

Up until now, our discussion has revolved around using natural language to retrieve and answer questions from different knowledge sources. HAWK, instead, is a project that uses natural language as an agent to parse information from a variety of expert sources and then inform a single knowledge based system. Its members have identified that in the past, there have been several barriers to making significant progress with such systems. Today though, they believe that the advances in natural language processing have come to the point that their goals are now feasible. [5]

If projects such as HAWK make significant progress, we will be a giant step towards building truly intelligent systems. That’s to say, should START be able to both learn (acquire data itself) as well as answer questions, we’d be on the road to omniscience.

Future Applications

We are very excited about the opportunities presented as we amass an arsenal of different data sources about which we can answer questions. We believe that as the system incrementally grows, its functionality will address more and more of the world’s questions and problems. We believe that because of our modular design, this enhanced system can be used in different environments for a variety of different tasks. Below are a couple of applications in which we feel our system would be ideal.

Help Desk Tool

A large cost for many businesses today is the need to hire a staff of Help Desk professionals whose job it is to answer questions from either internal employees or external customers. Each of these individuals go through a rigorous training program and then undergo addition periodic training. Instead of investing all of this time and money in training individual employees, if all of that data were to be added to a system like START, it seems that massive costs could be cut. Obviously, such a system would not be able to answer all questions, but it would definitely have the capability of answering the majority of the “stock answers” that the current Help Desk professionals answer.
Data Mining Tool

Many large companies invest a huge amount of money maintaining vast inventory, accounting, and order databases. Although our system could not replace any of these resources, it could most definitely make them more accessible. Most companies today hire specific people just to retrieve data from and modify these systems. Should a START-like system be interfaced with such resources, access to any piece of information (given that one has authorization), would be as easy as a simple English question. Imagine being able to simply ask the total number of item X sold on August 10th or the remaining inventory of item Y. Such systems could be implemented using START and the enhancements discussed in this document to provide great increases in simplicity and productivity.

Conclusion

Now that our base components have been appended to START, we hope our system will be the next step on its path of maturing and flourishing. As we provide START with a larger knowledge base, along with a more extensive symbol table, we should see glimpses of an omniscient "being" beginning to emerge. As endeavors in artificial intelligence from the past have illustrated though, building such systems takes both patience and perseverance. With a solid foundation plus more advances in data acquisition and organization, I see few critical limits to our building an ever-evolving and strengthening system.
References


