A Hypermedia Inference Language as an Alternative to Rule-based Expert Systems

Gerry Stahl

Technical Report CU-CS-557-91
November 1991

revised August 1992

An abriged version of this paper is forthcoming in:
G. Stahl, R. McCall, G. Peper (1992) Extending Hypermedia with an Inference Language: An Alternative to Rule-based Expert Systems, *Proceedings of the IBM Internal Technical Liaison Conference: Expert Systems (October 19-21, 1992)*.

Department of Computer Science
University of Colorado at Boulder
Campus Box 430
Boulder, Colorado 80309-0430 USA

phone: (303) 444-2792
e-mail: gerry@cs.colorado.edu

Abstract

This paper reports on the development of a hypermedia inference language designed to strengthen the ability of hypermedia systems to be used effectively in applications that might otherwise require cumbersome rule-based expert systems. The inference language grew out of a primitive query language which provided the mechanism for navigation in a hypertext system. As the language gained logical and computational capabilities it became increasingly embedded in the nodes and links. A new paradigm of intelligent hypermedia emerged, incorporating "smart" nodes and links that were dynamically computed by means of the inference language. The language itself provided an end-user programming facility that was English-like enough in appearance to be readily comprehensible to non-programmers. An application in the domain of academic advising was developed to compare the inferencing language approach to expert system alternatives.

A Hypermedia Inference Language as an Alternative to Rule-based Expert Systems

Overview of Report

This paper reports on the development of a hypermedia inference language designed to strengthen the ability of hypermedia systems to be used effectively in applications that might otherwise require cumbersome rule-based expert systems. The inference language grew out of a primitive query language which provided the mechanism for navigation in a hypertext system. As the language gained logical and computational capabilities it became increasingly embedded in the nodes and links. A new paradigm of intelligent hypermedia emerged, incorporating "smart" nodes and links that were dynamically computed by means of the inference language. The language itself provided an end-user programming facility that was English-like enough in appearance to be readily comprehensible to non-programmers. An application in the domain of academic advising was developed to compare the inferencing language approach to expert system alternatives.

This report will start by reviewing the actual progression of the research. It will begin with the original assumptions and goals and show how they were explored. A series of discoveries during the year's work led to further ideas and techniques. In the end, certain technical difficulties that had not been envisioned were overcome and a conception of intelligent hypermedia was fashioned. The creation of test applications embodying the new system revealed how its power might best be exploited.

The history of the research will provide an introduction to the system of intelligent hypermedia which emerged and a context for understanding its significance. This will be followed by illustrations of the use of the language in sample applications. They should give a good feel for the system's usability as well as its utility. The inference language will be described next. The content and structure of this language embodies the real power of the system. Much of the research time was spent in the design and development of the language. An important emphasis of the research was trying to keep the appearance of the inferencing language as English-like as possible, and to keep its use intuitive. Finally, implications of the research will be discussed and conclusions drawn.

Comparing Rule-Based Expert Systems with Hypermedia

Expert systems are most useful in well-defined domains in which the rules can be made explicit. However a study by researchers at IBM (Peper, et al, 1990) identified a number of problems with traditional rule-based expert systems. The design of systems of rules is difficult and problem-laden. Even more of a concern is the issue of maintaining rule-bases. Maintenance is always a primary concern in the software lifecycle, as both the rules of the domain and the needs of the users evolve. Expert system shells, designed to obviate the need for specialists with computer programming expertise, have not eliminated these difficulties.

There are a number of reasons for the problems with rule-based systems. The first step, encoding and ordering the rules, is a major challenge. This is because the syntax of the rules is non-intuitive, and hence hard for users to understand and modify as well as awkward to encode. Furthermore, because of the nature of the inferencing process in the expert system engines, the ordering of the firing of rules is critical. The firing of rules can also have unwanted side-effects. In particular, conflicts between which rules to fire can arise, creating what is perhaps the most significant problem for maintenance of expert systems: conflict resolution. In addition, even once the rules have been adequately debugged, special procedures often still need to be programmed in source code (e.g., Lisp). Finally, expert systems tend to be inflexible. They pursue a fixed line of inquiry entirely under the computer's control. Thus, it is not possible for the user to introduce new information unless explicitly prompted for it, or to explore the information in the system in an unrestricted manner.

The IBM study showed that hypermedia navigation could often provide an effective alternative to rule-based inference systems. Such an approach gives users greater control and allows them to explore the knowledge base. The study concluded that a hypermedia system could be just as effective and easier to create and maintain. Also, the hypermedia system can run faster and require less computer resources than rule-based expert systems.

The original idea led to HyperWin, an IBM product. Applications are, indeed, easy to construct, understand, and maintain with it. They can be used in an exploratory way with no training. Several applications have been developed in HyperWin, including an academic advising system for Auburn University.

Hypermedia represents an appealing alternative for situations in which all the knowledge can be laid out as a network of textual nodes and links for traversal by the user. However, there are many applications in which inference by the system would also be desirable or necessary. Hypermedia represents an appealing alternative for situations in which all the knowledge can be laid out for the user as a network of textual nodes and links. However, there are many applications in which inference by the system would also be needed.

Extending Hypermedia with Inferencing

It was decided to add the power of inference to the elegance of hypermedia navigation. Some expert system applications can be defined as a network of nodes for navigating without inference. Others are wholly reliant upon inference computations. But many applications fall between these two extremes and might best be served by combining the two approaches.

The project began by building on an existing hypermedia system called Mikroplis (McCall, 1989), with an English-like query language that has been successfully used for several years. The Mikroplis query language is a comparatively simple language for navigating across links in a hypermedia document. It has a total of 12 syntactic options and a limited potential complexity, compared to over a hundred options in the inference language developed in this research. It allows the system to select links from a node and to check the content of nodes for the inclusion of a substring of characters.

To support a wide range of inferencing, the language had to be extensively expanded to include true/false conditionals, numerical calculations, comparison operations, and nesting of phrases. (See the appendix for a listing of the abstract syntax of the new language, with the options from Mikroplis underlined.) A typical request in the new language -- taken from the test domain of academic advising -- might look like the following:

```
Display all courses of Sandra which have studio_types and
which also have less than 3 prerequisites, with their
prerequisites.
```

To evaluate this statement, the system would navigate from the student node, `Sandra`, across all its `courses` links; check which nodes arrived at had at least one `studio_types` link and also had less than three `prerequisites` links; and output a list of the course nodes that satisfied these conditions, along with a sublisting of their prerequisites. The output might look like this:

```
***COURSES:
1. ENVD 2110  Architectural Studio
     *** PREREQUISITES:
     1. ENVD 1000  Environmental Design Studio
     2. ENVD 1014  Intro to Environmental Design
2. ENVD 2120  Planning Studio
     *** PREREQUISITES:
     1. ENVD 2110  Architectural Studio
```

The structure of statements in the inference language and their method of evaluation are based on the structure of hypermedia. The queries investigate the node and link structure, rather than the content of a database, and their evaluation proceeds by navigation across the links from initial nodes. In this sense, the research represents an effort within the hypermedia paradigm. The thrust of the effort is to exploit hypermedia mechanisms to achieve certain functionality of artificial intelligence and information retrieval technologies. Thus, the goal was to expand hypermedia to include:

* Some of the inferencing capability of Prolog, but without the comprehension difficulties of predicate calculus and explicit variables;
* Some of the querying ability of SQL, but without the inefficiency of relational joins;

* Some of the advantages of semantic databases, but allowing semantic relationships to be defined between instances as well as types; and
* Some of the utility of semantic networks, but without restriction to a pre-defined set of types.

A Navigation Language for Nodes

The original approach relied heavily on the idea of *smart nodes*, in which the inferencing power is embedded in the nodes of the hypermedia. This was conceived primarily in terms of *virtual structures*, an extension of the fixed structures of textual or graphical nodes in traditional hypermedia systems, suggested by Frank Halasz (1988). The navigational (or structural) approach to query evaluation was used, as found in the Mikroplis language, and embedded the language in the hypermedia nodes. This was done to avoid simply gluing together two different paradigms (e.g., hypermedia and Prolog, or hypermedia and SQL, or HyperCard and HyperTalk) and to develop the querying or inferencing capability out of the hypermedia paradigm itself.

The content of a smart node is not limited to the text or graphic originally entered into it. Instead the content is determined by the results of a query or conditional phrase associated with the node. The query traverses the hypermedia network, so its result depends upon the current state of the network: the existence of other nodes, their links and their current content. When smart nodes are displayed, the appearance of the hyperdocument itself changes dynamically.

Two forms of smart nodes were explored: *conditional nodes* and *virtual structures*. A conditional node contains a conditional phrase in the inference language and normal text or graphics. If the condition evaluates to true, the text is displayed. If the condition is false, nothing is displayed. For instance, in the academic advising application a node with the text, "`Are you interested in a studio course?`" might have the condition, `If there are courses which have studio_types`. Then the text would be displayed only if there actually were studio courses for the student to choose from.

A virtual structure differs from a conditional node in that it contains only a query. Instead of fixed text, the system displays the result of the query. So, in the previous example, if there were studio courses and the user responded to the question with a "yes," then the yes response might be implemented as a link to a virtual structure node with the query, `Display all courses which have studio_types`. The user would not see the statement of the query, just the results.

Conditional nodes and virtual structures add significant flexibility to hypermedia. They allow specific nodes to be responsive to changing conditions in other nodes of the hyperdocument. For instance, decision trees can be implemented using smart nodes by basing new decisions on nodes that contain the results of previous decisions.

The major surprise of this research was an important limitation of smart nodes. Suppose you had defined an inference computation for a specific node, embedded it in that node, and found that it

worked fine. But now you wanted to apply the same computation to other nodes without explicitly entering the condition or query in each of the other nodes. More generally, suppose you wanted to apply the computation as an operation on an arbitrary list of nodes. This turned out to be a critical concern because it was important to be able to do this within the inferencing language itself.

Adding Smart Links

Smart *links* or *predicates* solved the limitation of smart *nodes*. Smart links are different from primitive links or defined link types. When a hypermedia system is designed, a set of link types is defined. For instance, in the academic advising application there might be links of type `proposed_courses` from a student's node to his or her chosen course nodes, and other links of type `prerequisites` from course nodes to other course nodes. A smart link would then be a virtual link that was computed based on the definition of a predicate. For instance, a predicate might be defined as:

```
required_prerequisites = proposed_courses which have
prerequisites, with their prerequisites.
```

`Required_prerequisites` would not be a primitive defined link type, but a computation or an inference.

This is an example of a query using normal primitive links:

```
Display the proposed_courses for Sandra.
```

It would be evaluated by following the `proposed_courses` links from the student node `Sandra` and displaying the nodes reached:

```
*** PROPOSED_COURSES:
1. ENVD 2110  Architectural Studio
. . . .
```

This is an example of a query using smart links:

```
 Display the required_prerequisites for Sandra.
```

It would be evaluated by substituting the definition for the computed link type into the query and displaying the result:

```
***PROPOSED_COURSES:
1. ENVD 2110  Architectural Studio
     *** PREREQUISITES:
     1. ENVD 1000  Environmental Design Studio
     2. ENVD 1014  Intro to Environmental Design
     . . . .
```

The idea of substituting a definition for a term in a query is known as *macro expansion*. The definition of smart links as macros turns out to be an extremely powerful mechanism for the inferencing language. Because of the way the substitution is implemented, recursive definitions of smart links are possible. This allows simply stated queries to evaluate tree structures and easily display transitive closures, in both breadth-first and depth-first order -- an accomplishment not matched by relational query languages like SQL.

During the research, the process was refined to distinguish between macros and predicates. A predicate is like a macro; however, when its results are displayed, they are labeled to appear as though the predicate were a primitive link type. This is critical for the user. Now when the user says,

```
Display the required_prerequisites for Sandra.
```

the user does not need to know that `required_prerequisites` is anything but an ordinary link type. The result is displayed like this:

```
*** REQUIRED_PREREQUISITES:
1. ENVD 2110  Architecture Studio
2. ENVD 1000  Environmental Design Studio
3. ENVD 1014  Intro to Environmental Design
. . . .
```

So now there are three kinds of links:

* Primitive links, which are the traditional link types of hypermedia.
* Macros, which add significant inferencing power.
* Predicates, which use the power of macros but hide the complexity from the user.

Predicates like `required_prerequisites` had to be defined and the differences between types, macros, and predicates had to be considered during system development, but the eventual user can use the computational power without knowing that no links exist between student nodes and their required prerequisites. The predicates look like simple links to the user. Therefore, they are called smart, computed or inferred links.

Smart links overcome the limitation of smart nodes. Because macros and predicates are syntactically equivalent to primitive link types, they can be bound to arbitrary nodes or lists of nodes as if they were actual links coming out of those nodes. Smart links turned out to be so powerful and flexible that the academic advising application was primarily developed with them. Smart nodes were incorporated for only a few special situations. (The application will be described later in this report.)

The Future: Intelligent Hypermedia

The implementation of smart nodes and smart links in effect defines a new paradigm of intelligent hypermedia, in which the elements of the system -- the nodes and the links -- are not necessarily fixed text (or graphics), but can in general include any query results. The inferencing language is thereby conceptualized as integral to the system elements. The new paradigm of intelligent hypermedia represents a leap of abstraction with major practical implications. These implications will need to be explored in the future.

The major consequence of the new paradigm is that all nodes can be conceptualized as query results. This means that the inferencing language and the hypermedia built on it must incorporate all media on an equal footing, so that query results can display text, numbers, truth values, drawings, bit maps, animation, sound, etc. Inferencing and computation mechanisms must be fully polymorphic, so they can be applied to content from any medium. Furthermore, if nodes can

contain query results, then they are typically lists of elements rather than single elements of text, graphics, etc. This means all node processing must be list-oriented.

An example of a query in the future system might be,

```
Display the kitchen with appliances which are less than 6
feet high but more than 3 feet high and are seen from the
doorway.
```

This might result in the display of several graphical objects representing appliances meeting the stated conditions and viewed from the specified perspective. That is, this query -- which might be the content of a node which the user can navigate to -- evaluates to a computed list of graphical objects (each of which might itself be computed).

The new paradigm has particularly broad consequences for graphics. Composite drawings, rather than being conceived as fixed arrangements of polylines, could be thoroughly reconceptualized to integrate inference language queries. Correspondingly, the syntax of the inference language could be extended to encapsulate computations of scale, spatial transformations in 3-D, hidden plane removal, etc. Detail-on-demand could similarly be incorporated, so that only details above a specified or calculated extent would be displayed.

A follow-up grant to the research reported here is supporting research exploring mechanisms for virtual copying of parts of the hypermedia network, implementing a kind of local versioning. This work introduces the notion of inheritance. Inheritance has proven to be a fundamental concept for other AI work. It could conceivably have broad use in intelligent hypermedia. In addition to inheritance of specific networks of nodes and links, the node and link types could participate in inheritance hierarchies. Thus, for example, the `studio_courses` type could be a kind of the `courses` type. Then inference about `studio_courses` could take advantage of this relationship as well as those defined by primitive and smart links in the network.

The envisioned form of intelligent hypermedia has sufficient power that all displays in an application could be defined as queries in the inference language. Then there would be no system-defined screens, like general browsers in which the user can become "lost in hyperspace." All application screens would be under the complete control of the user through the formulation of appropriate queries. (This report will discuss features of the current version of the language which make it plausible that a non-expert user could reasonably be expected to handle such a powerful inferencing language.)

An Example of Inference

Having reviewed the goals, progress and implications of the research project, let us take a closer look at the new paradigm of hypermedia and its inference language.

A textbook example from logic programming like Prolog provides a good illustration of how predicates can be used in the inference language to break down and solve a typical inference problem. Take the problem: given a network of `people` nodes linked by `son` and `daughter` links, infer `cousin` relationships. Inference is defined as the combining of facts to deduce new

facts. Here, facts about sons and daughters are combined to produce facts about who is a cousin of whom. This is a non-trivial task for humans, generally requiring people to consciously articulate part of the computation (e.g., "Let's see, her mother is my father's sister. . . .")

In the new inference language, the problem could be solved by the definition of the following predicates:

```
children = sons and daughters.
parents = converse sons and converse daughters
siblings = children of parents which are not self which
          contain no duplicates.
cousins = children of siblings of parents.
```

Of course, these definitions require some explanations about the language -- although far less explanation than corresponding definitions in a traditional programming language like Lisp or Prolog. The `children` predicate includes nodes linked by both `sons` and `daughters` primitive links. Converse links are primitive links traced backwards, like from the son back to the person whose son he is. The definition of `siblings` is inherently tricky. Most likely, the definer of this predicate would discover an adequate definition through a series of successive refinements. If one defines `siblings` as just `children of parents`, one discovers upon first use of the predicate that the original people are always included among their own siblings, because they are sons or daughters of their parents. Therefore, a condition must be added to exclude the original person from the result list. Similarly, there will usually be duplicate names on the list of siblings because they are children of both the mother and the father of the original child. The simplest way of solving this problem while maintaining the ability to handle children of multiple marriages is to simply eliminate duplicates from the final list of results.

The complexity of the definition of siblings is telling. Although the task of determining cousins is difficult for people, the problem does not lie in the definition of `siblings` but rather in the sequence of steps that must be put together. People naturally exclude the extra results that pop up surprisingly when the `siblings` predicate is incompletely defined. This is symptomatic of the fact that programming in any language in any domain is going to require some steps of logical analysis and some efforts at debugging. No language, however English-like can entirely avoid that. The primary advantage of the inference language described here is that once predicates are successfully defined, it is clear what they mean, even for someone with little training in the language. The definition of siblings is about as obscure as any statement in the language need be.

Given the above definitions, the following computations can now be evaluated:

```
Display the cousins of Sandra.
Display all people which have cousins and which also have
     less than 3 cousins, with their cousins.
```

Furthermore, these definitions have begun the creation of a domain language for family relationships. It is an easy matter to add predicates for brothers, aunts, grandparents, etc.

A particularly interesting definition is that of descendants:

```
descendants = children with their descendants.
```

A programmer would recognize this to be a *recursive* definition. That is, it not only lists the descendants of the starting node, but the descendants of those descendants, the descendants of descendants of descendants, etc. until there are no more generations. A non-programmer might be able to see that this definition would produce such a result, without having studied recursive function theory in the abstract. Again, the non-programmer might not be able to generate recursive definitions easily from scratch, yet might understand them when seen.

Styles of Computation Using the Inference Language

Recursive programming is a potentially powerful technique. The language Lisp (the traditional language for artificial intelligence programming) relies almost exclusively on recursive processing. This technique is particularly useful for processing trees of data, like family trees. In the academic advising application, tree structures appear in the list of course prerequisites. The full set of tree elements is called the *transitive closure*.

The two primary approaches to enumerating a transitive closure by navigation through a tree structure are *depth-first* and *breadth-first*. Both of these approaches can be programmed in the inference language. The following definition and query produce a nested, depth-first listing of the transitive closure of course prerequisites:

```
prerequisite_trees = prerequisites with their
          prerequisite_trees.
Display ENVD_4550 and ENVD_4560 with their
          prerequisite_trees.
```

The following definition and query produce a flat, breadth-first listing of course prerequisites:

```
prerequisite_lists = prerequisites and prerequisites of
     prerequisite_lists.
Display the prerequisite_lists for ENVD_4550 which contain
     no duplicates.
```

The computation through trees has important applications in practical problems. For instance, in a hypermedia system of issues, subissues of the issues, subissues of the subissues, etc., it is useful to define the `issue_trees`, a depth-first listing of the whole tree of issues. If the issues can each have answers and arguments for the answers (as in the popular hypertext IBIS systems), then one wants to list `deliberations` -- the tree of arguments on the issue tree. This is straight-forward to do in the language. It is trickier to produce a list of the *terminal* issues, that is subissues at the leaves of the issue tree which have no subissues themselves. This can be done with a predicate for `terminal_issues`:

```
terminal_issues = terminal_issues of issues if there are
issues of issues, else issues.
```

The inference language is also designed to take advantage of *defeasible reasoning* in an intuitive way. Defeasible reasoning allows a system to be designed with certain default behavior which results unless explicit action is taken to change it. Suppose in a hypertext network of issues and answers one wants to allow a user to accept, reject or ignore answers by attaching `status` links

to nodes containing words like "accept", "reject", "ignore", "don't care", or no links. One might also want to allow multiple `status` links from any given answer node. So there may be contradictory information attached to an answer, or no information at all. Suppose further that one wants to display an_important_issue unless all its answers have been explicitly rejected with status links to "reject". This would require defeasible reasoning, a very robust approach. The following query could be used:

```
Display an_important_issue if there are not answers of
an_important_issue which have no statuses which equal
reject.
```

Another important programming technique -- particularly for expert system applications -- is *decision trees*. A typical example of using a decision tree is categorization of fauna and flora. One proceeds through a sequence of questions posing alternative choices. Based on ones answers, the choices lead down a path through the tree of decisions to the answer, e.g., the name of the animal or plant corresponding to the choices. Here is an example from the domain of academic advising, implemented with virtual structure smart nodes. For purposes of the example, most of the nodes have been given names like `node_4` to make it clear that they are (smart) nodes; in a realistic setting they would have more meaningful names. Suppose we have the query,

```
Display the suggestion.
```

And suppose the node named "`suggestion`" contains the following query:

```
Display node_2 if envd_semester of student is less than 3,
else node_3.
```

Assuming that the proposition (if envd_semester of student is less than 3) turns out true, `node_2` is evaluated. It contains the query,

```
Display node_4 if completed_courses of student do not
contain ENVD_1000, else node_5.
```

Suppose we take the branch of the tree to the simple `node_5`, which contains the text, "`Take ENVD 1000.`" Then this text is displayed in response to the original query. The virtual structure nodes have implemented a decision tree in a way which is relatively easy to understand and to modify if necessary. The links through the hypermedia defined by the embedded queries reflect in a very straight-forward way the structure of the abstract tree of decisions. Here again, the system requires some analysis to set up, but once defined in the inference language it is rather self-documenting.

While the above implementation of a decision tree is appealing, it demonstrates the limitation of smart nodes as well as their power. Note that in the last two queries the node `student` was referred to by name. If one next wants to evaluate the decision tree for another student, the new student information must be substituted in the hypermedia network which contains the smart nodes. The decision tree cannot be simply applied somehow to other existing nodes, let alone to arbitrary lists of nodes (the way predicates can). This is a form of the general binding problem, a consequence of avoiding the use of variables in order to keep the language easy to understand. In the inference language one cannot say "`If envd_semester of X is less than 3`," except by defining a predicate to encapsulate that computation and applying the predicate to an

arbitrary subject. That is why predicates are used so extensively in applications using the inference language.

But predicates have their own *binding problem*. When it is used in the evaluation of a query, a predicate is implicitly (automatically) bound to whatever subject it is applied to. Therefore, any unbound relationship in the predicate definition is implicitly bound to that subject as well. However, predicates can have whole queries embedded in them and so a question arises concerning the subjects of these embedded queries. If there is an explicit subject node named in the embedded query, then there is no problem. However, predicates draw much of their power from binding to implicit subjects, as explained in the previous paragraph. Therefore, the inference language permits leaving the subject unnamed in an embedded query. In such a case, the implicit subject of the embedded query is bound to the last explicit subject of a query (i.e., to the subject of the query in which the embedded query is embedded, or if that query has no explicit subject then the subject to which its subject is bound). This procedure is based on the usual assumptions of the English language, so that inference statements behave the way English-speaking users would expect them to, without the user having to think in programming terms.

For an example of the two binding mechanisms presented in the previous paragraph, consider the problem of determining what problems a student has with missing prerequisite courses. The query for this can be based on a predicate named "`prerequisite_problems`" which contains a predicate named "`prerequisites_not_taken`":

```
prerequisites_not_taken = prerequisites which are not
     contained in Display the courses_taken.
prerequisite_problems = proposed_courses which have
     prerequisites_not_taken, with their
     prerequisites_not_taken.
Display the prerequisite_problems for Sandra.
```

In this query, "`prerequisite_problems`" is bound to the explicit subject of the query, `Sandra`. The other predicate used in its definition, `prerequisites_not_taken`, is applied to `proposed_courses` through composition. So `prerequisites` in its definition is bound to `proposed_courses` (i.e., we are concerned with the prerequisites of the proposed courses). The issue arises with `courses_taken`. These are not courses taken by the proposed courses, but by Sandra. According to the syntax of the query, `courses_taken` is part of an embedded query: `Display the courses_taken by X`. The subject is left implicit, which to English speakers means it refers to the previous main subject, Sandra. This is in fact the rule used for binding implicit subjects of embedded queries in the inference language as well.

The inference language solves the binding problem through the two mechanisms illustrated above. This allows predicates to exercise their power of leaving their subjects implicit, to be bound at runtime. The solution maintains the language's English-like quality by corresponding to the intuitions of non-programmers. While it cannot handle arcane examples requiring binding to multiple or obscure subjects, it handles reasonable, humanly comprehensible examples -- including arbitrarily deep embedding of queries. The example of `prerequisite_problems` is a realistic one, occurring in the main test application described below.

The Academic Advising Application

The IBM HyperWin system provided not only the starting point for this research, but also the sample application for testing the results of this research: an academic advising system. The HyperWin version allowed a user to navigate through a hypermedia database of information about courses at Auburn University. This system asked the user about interests, courses already taken, etc., and responded to answers chosen by the user with appropriate further information.

To demonstrate the inference language, an application was created using information about the curriculum of the College of Environmental Design at the University of Colorado. This information included not only lists of offered courses, but other facts and rules used by the College's official student advisor. Courses were linked to their prerequisites and to their categories, such as which curriculum they belonged to and which elective breadth requirements they satisfied. Other, less formal factors were also included, like which courses were particularly labor intensive.

The centerpiece of this application was the definition of a predicate named `advice`. This predicate was built on a combination of several specific kinds of advice, which in turn used predicates to compute inferences across the hypermedia. The idea was that a student, Sandra, could enter her name, curriculum option, semester number, completed courses, current courses and proposed courses into the hypermedia system. By clicking on the Advice button, Sandra would initiate the query,

```
Display the advice to Sandra.
```

The query critiques Sandra's proposed list of courses. This is a typical result:

```
Here is some advice on your choice of courses:

The following courses each require a lot of work. It would
be wise not to take them in the same semester:
          ENVD 3220  Planning Studio 2
          MATH 1300  Calculus

The following courses are not designed for your curriculum
option:
          ENVD 3220  Planning Studio 2

You have not taken the listed prerequisites for the proposed
courses:
          ENVD 3220  Planning Studio 2

With your proposed courses you will not satisfy the
following elective breadth requirements:
          science
It would be wise to take a course in one of these areas
rather than the following proposed courses in elective areas
for which you have already satisfied the breadth
requirements.
          FINE 1012  Art History
```

The above is the actual system output for a sample student. Relatively intricate computations have been performed to check, count and list courses meeting or not meeting certain conditions. In particular, for instance, the advice about breadth requirements is only displayed if the proposed courses include an elective in an area that has already been satisfied and do not include one in an unsatisfied area. This kind of inferencing facilitates the offering of important information tailored to a particular user in a way that is impossible in a purely navigational hypermedia system. It begins to look like a rule-based expert system, but without many of the problems of such systems.

Developing an application of this level of complexity requires some system designing expertise. One needs to know how to represent the knowledge in hypermedia and how to build up a sequence of modular definitions. This is probably inevitable in any system. Once designed, however, the system is significantly easier to understand, modify, and extend than alternative implementations would be. While the result of the advice predicate looks like the output from a traditional expert system, the flexibility is still there to explore the underlying knowledge base by navigation. Alternatively, one can reformulate the major query or execute a series of simpler queries using components of the advice predicate.

The Inferencing Language

An abstract syntax defining the structure of the inference language is included in the Appendix. The language consists of a number of options for various clauses of a query: Subjects, Relationships, Filter conditions, Boolean propositions and Queries. The interface for constructing queries allows the user to choose from appropriate options at each step. Although the language is in fact tightly constrained, statements of queries always look very English-like. The options are displayed in ways which make programming of the language as intuitive as possible.

The general form for a query is defined as:

```
Q::= display R of S which F if B with their R', else Q.
```

Each of the capitalized symbols in this definition stands for a clause, which can be chosen from a number of options. An example of this form is the following, in which slashes have been inserted to distinguish the major clauses:

```
Display those proposed_courses / for Sandra / which have
prerequisites_not_taken / if there are more than 3
proposed_courses / with their prerequisites_not_taken, /
else display proposed_courses of Sandra with their
prerequisites.
```

The ordering and phrasing of clauses is designed to give an English-like expression to the query. The actual evaluation of the query is implemented in a very different order:

```
if B then R' ( F ( R ( S ) ) ), else Q
```

That is, first the Boolean propositional clause is evaluated to see whether it is even necessary to evaluate the following clauses. Then, the system starts with the Subject clause to determine where to begin in the hypermedia. From there the Relationship clause is applied to the list of

subject nodes to follow specified primitive or smart links. The Filter clause is subsequently applied to the nodes reached to see which of them contain the specified contents. Finally `with their R'` is applied to the current results, typically to produce recursive sublistings of nodes traversed to by relationship `R'`. Thus, although it is not necessarily apparent to the user, the language is implemented through mechanisms based on navigation of the hypermedia structure.

The language itself is applicative and declarative, rather than procedural. This allows query clauses to be successively applied without restriction -- imposing finer and finer information filters. Applicative programming also simplifies things by requiring no variables or state changes. This eases the cognitive load on the user, who need not be concerned about variables -- or even understand what a variable is. The absence of state change substantially reduces the possibility of side-effects of rule-firing, which are so problematic in rule-based systems. In effect, the language lets the user specify *what* should be done by successively applying conditions, without worrying *how* the task should be accomplished by the computer -- the way a programmer does who writes a sequence of procedural commands in a programming language. This, again, makes the inference language more like an English language communication than like a traditional computer programming language. The user can concentrate on what is desired in domain terms, rather than worrying about details of computer hardware and software. It is true that the user needs some familiarity with how the hypermedia is structured, but that structure should correspond closely to a representation of the domain, so being aware of the hypermedia structure should not distract too much from the user's focus on domain concerns.

The development of the software architecture to support this very flexible and powerful language required a complete re-write of the Mikroplis hypermedia and query language system, which had explored some of these approaches. The new implementation is object-oriented, fully modularized and consistently applicative. Each clause has its own methods for being evaluated and displayed. Polymorphic techniques allow the methods for the selected clause option to be executed. This allowed the rapid expansion of the language through the successive implementation of new features and options. It also allows arbitrarily deep nesting of clauses within clauses, making the language infinitely generative and as complex as one wishes. The expressive power of the language is further enhanced by the recursive potential of macro and predicate smart links.

Most of this complexity can be hidden from the user. Queries, macros, predicates and other syntactic clauses can be defined once and stored with a descriptive name (like "`cousin`" or "`advice`"). From then on, the user can treat the name as a primitive term -- or reuse and modify its definition. Typically, names will be taken from the user's application domain. In this way, statements of queries will read like English sentences in the domain:

```
Display the cousins of David.
Display advice for Sandra.
```

Viewed the other way, the relationships of a domain can be programmed into a vocabulary of user-defined node types, link types, macros, predicates and queries. This may be assisted by a system designer programming a *seed* vocabulary for the domain as a basis for the user to start with. At any rate, the vocabulary is always open-ended so that users can add their own concepts

as needed. These terms embody the semantics of the domain in a form that can be used naturally by anyone familiar with the domain whether or not they have any programming experience.

Using the Language

While the inferencing language is meant to give the appearance of natural English, it is in fact tightly constrained in its syntax. The vocabulary is less constrained than the syntax, in that it is primarily based on node and link types, which are named by the user when they are defined. The syntax of statements in the language is constrained operationally through the user interface. While a point-and-click interface is now available in Windows, the origianl interface consists of textual menus. The older menu interface will be used here to illustrate the process of defining a predicate and a query statement.

Suppose we first want to define the following predicate:

```
cousins = children of siblings of parents.
```

Assume that `children`, `parents` and `siblings` have already been defined as predicates. Then from the main menu we first select the option, `Define a predicate`. The following menu is displayed. (Because predicates are just a form of Relationships, this menu is used for Relationships and predicates.) The options in this menu correspond to the options for Relationships in the language syntax (cf. Appendix):

```
0 = EXIT (nil)                          1 = everything
2 = a relationship type                 3 = converse T
4 = R which are not self                5 = the Nth R
6 = R which F if B with their R, else R    7 = R of R
8 = R and R                             9 = a named predicate
10 = a named macro
number: __
```

We are trying to define `children of siblings of parents`, so we select option `7 = R of R`, a composition of two Relationships. This choice results in the same menu being displayed to select the first of the two composed Relationships. Again we choose `7 = R of R`, thereby defining the syntax of our predicate as: `(R of R) of R`. Next we will be given the menu for Relationships three more times, to define each of these three Relationships. In these cases we will select `9 = a named predicate`, and type in "`children`," "`siblings`" and "`parents`" respectively. (In the Windows interface they are selected from a mousable pick list.) The system now displays the defined predicate and asks for the user to name it.

Having defined `cousins`, we are ready to define a query using this predicate. Suppose we want to define the following query:

```
Display all cousins of Sandra which have children.
```

From the main menu we select `Define a query`. The menu for queries is:

```
0 = EXIT (nil)
1 = display Art R Prp S which F if B, with their R', else Q
```

```
2 = the Nth result of Q
3 = Q and Q
4 = a named query
number: __
```

We choose option 1, the general form for a query. We are then successively prompted with menus for the components of this syntactic form: Article (a, all, an, that, the, those), Relationship (menu options as above), Prepositions (about, by, for, from, in, of, on, over, to, under), Subject, Filter, Boolean, Relationship, Query. We select the following: `all`, `cousins`, `of`, `Sandra`, `have Qop R which F`, nil, nil, nil. Note that the article and preposition chosen by the user, as well as the word "`Display`" which is prepended to the query statement are purely cosmetic and have no significance for the query evaluation. The same is true for the connective English terms in the syntax, like "`which`" or "`with their`".

The Filter clause (`have Qop R which F`) shows how clauses can be nested to arbitrary depths. The Relationship or the Filter could be defined through various sequences of menus for choosing their constituents. For our example query, we keep things simple and choose for Qop (the quantity operator): "`at least one`"; for R (Relationship): "`children`"; for F (Filter clause): nil. Thus, our defined query is: "`all cousins of Sandra which (have at least one children which nil) if nil, with their nil, else nil`". The nil clauses are not displayed or evaluated. By convention, the "`at least one`" is kept implicitly, but is not displayed. So the defined query is displayed back to the user as: "`Display all cousins of Sandra which have children.`"

An end-user programming language statement thereby appears in an English-like form. The menu system has stepped the user through the process of formulating a query without either pretending that the user can say anything he or she wants to in English or allowing the user to enter anything which is ungrammatical in the inferencing language. The point is not to pretend that English is being used by the computer, but to make the statements in the language easy for an English-speaking user to comprehend.

Because the language has an object-oriented implementation, the query can be constructed while it is being defined by the user. Therefore no parsing routine is necessary, and the syntax is not required to be unambiguous. The binding and nesting of clauses is done based on the order of the defining steps, so an intuitive approach by the user generally results in the intended meaning of the query.

The query is an object, whose data consists of Article, Relationship, Preposition, Subject, Filter, Boolean and Query objects. They are each defined as the user makes the corresponding selections for them or for their components. Each type object has methods to display and to evaluate themselves. For instance, a Query object displays itself by first displaying the word "Display", then having its Article object display itself, then having its first Relationship object display itself, and so on. Similarly, a Query object evaluates itself by first having its Boolean object evaluate itself and then (assuming that evaluates to True) having its Subject evaluate itself,

etc. This approach allows a flexible, infinitely generative language to be built up from a limited number of carefully crafted elements.

Research Directions

This research is situated in three distinct traditions: expert systems, hypermedia and design environments. The focus of the work reported here was on developing intelligent hypermedia as an alternative to rule-based expert systems, following the lead of HyperWin and extending that with an inferencing capability. That approach seems to have considerable promise for the future. One possibility for developing that alternative is in the direction of *design environments*. Another possible direction is defined by the hypermedia community itself. First, this section will consider issues for the future of design environments, and then for the future of hypermedia.

The paradigm of intelligent hypermedia built around an inferencing language seems like a useful approach for implementing design environments. A design environment like Janus (Fischer, et al, 1989) is an alternative to an expert system for designers. It is a software system which supports the work of designers by providing a construction kit of parts for building a design in a specific domain. It also supplies design rationale information when it is appropriate in the design process, and critiques the design as it evolves. Hypermedia can provide the text and graphics for design rationale and for design drawings. The language can supply a domain-oriented and fully extensible vocabulary for describing, evaluating and critiquing the design. A design environment represents a more fundamental alternative to expert systems than merely extending a HyperWin system with an inferencing language.

However, design environments suffer from a lack of integration. The most advanced design environment architecture described to date, Hydra (Fischer, et al, 1991), consists of half a dozen different components, all representing domain and design information. Special additional components are required to coordinate the primary components: for instance, a Catalog Explorer links the Specification and the Construction components with the Catalog. Intelligent hypermedia built around the inference language provides a form of knowledge representation that can be used by all of the primary components of a design environment. Then the role of linking is done by the formulation of statements in the inference language. Thus, a statement can request a display of catalog elements which meet conditions defined by textual nodes in the Specification sub-network and by graphical nodes in the Construction sub-network.

Design environments have also suffered from the lack of a user-defined domain language. Janus, for instance, has been used in the domain of kitchen design. However, it has no language for the user to discuss features of kitchens. Both critics in the system and specification forms use terms like "`is_safe`" or "`is_flammable`". But these terms (as well as the critic rules) must be coded in Lisp. Similarly, attempts in Hydra to integrate the content of hypermedia design rationale with filters for selecting from the catalog are problematic because of the lack of a user-definable domain language which could be used in the design rationale and also in query statements.

Generally, design environments have suffered from the lack of an end-user programming language. In an attempt to make these programs usable by non-programmers, developers of design environments have adopted the model of direct manipulation applications. However, such systems are ultimately limited, as argued by Eisenberg (1991). The developers of Janus have recognized this and added an end-user-modification capability (Fischer, et al, 1990). Unfortunately, this capability is limited to certain components of the system and requires knowledge of Lisp. By contrast, the inference language described here pervades the intelligent hypermedia system, because it is embedded in its nodes and links, as well as formulating queries which control displays and navigation throughout the system. It also has the advantage of being English-like in appearance and intuitive in how it works.

The research presented here builds directly on the Phidias system, which has always been intended as a design environment. (McCall, 1990) However, the discoveries in this research have wide-reaching implications for re-designing a system like Phidias as a well-integrated system based on intelligent hypermedia. This involves above all a re-thinking of the graphics system. Currently, graphics in Phidias are conventional vector graphics. They should be re-programmed as composite objects which can include query results. This would be part of a thorough-going conversion to hypermedia, in which text, vector graphics, bit maps, numerical data and truth values (also sound, animation, etc.) would all be handled uniformly. It would complete the re-write of the system as object-oriented and the integration of all information in a single hypermedia network, stored to disk in an object-oriented database.

The fully integrated system suggested by the new paradigm of intelligent hypermedia would integrate all components (critics, palette of parts, catalog of design examples, textual rationale, graphic design, specification). This would permit the display of any one component to be filtered by information from any of the other components. In the kitchen domain, for instance, a palette of appliances would not display dishwashers if there was already a dishwasher in the graphic design, if the specification didn't call for one, or if the design rationale had clearly opted against one. Perhaps more importantly, the user would have full (programmable !) power over all the displays by means of the language. Thus, the user could, at any time, add new concepts to the language and use these in critic rules, specification forms, design rationale or display definitions. This kind of programming power goes beyond the use of HyperTalk scripts in HyperCard because the HyperTalk sequential, procedural programming language is primarily oriented to controlling the appearance of the user interface, rather than providing an intuitive language for talking about a domain and doing computations in the domain.

The issues for the future of design environments correspond closely to the seven issues proposed by Frank Halasz for the next generation of hypermedia systems. (Halasz, 1988) It is worthwhile comparing the results of the research reported in this report with the vision outlined by this leading spokesman for the hypermedia community. He lists the following issues:

1. Search and query in a hypermedia network.
2. Composites -- augmenting the basic node and link model.
3. Virtual structures for dealing with changing information.
4. Computation in (over) hypermedia networks.
5. Versioning.

6. Support for collaborative work.
7. Extensibility and tailorability.

The status of these issues for the paradigm of intelligent hypermedia explored in this research is:

1. The inference language provides a primary access mechanism for search and query.
2. Composites seem particularly important as a way of implementing complex graphics.
3. Virtual structures have been implemented and explored in this research. The discovery of the limitations of smart links and the solution with smart links goes beyond Halasz' view.
4. The inference language performs computation over hypermedia networks.
5. Versioning will be explored in follow-up research on hypermedia network inheritance.
6. Support for collaborative work can be approached through design environments.
7. The inferencing language provides the kind of programmable extensibility that Halasz has in mind; the predicates and queries can be modified and tailored by the user; and the types, etc. are user-defined and always open-ended.

Thus, of the seven issues for hypermedia, four have been directly and successfully addressed in this research, and the remaining three are high on the list of priorities for the future.

A final concern for future work relates to user testing. A central priority of this research has been the attempt to keep the system as easy as possible for non-programmers to use, so it can indeed provide an attractive alternative to rule-based expert systems. As additional power and generality is added to the system it is important to monitor its usability empirically. This means realistic user testing. The inference language is currently in a primitive prototype version. The menu-based interface is adequate for internal debugging, testing and demos, but is not adequate for users. In particular, a construction kit approach to building, testing and modifying queries would be valuable for facilitating use of the language. A much larger, realistic application would also need to be developed in a domain of interest to users, containing information or computations not already available to them.

The software discussed here was developed in an object-oriented extension to Pascal in the MS-DOS operating system. It was subsequently ported to the Microsoft Windows 3.1 environment and enhanced it with a graphical user interface for defining predicates and queries in the language and for navigating through the hypermedia using smart nodes and links.

The language is currently being embedded in a high-functionality design environment. (McCall, 1990) The specific application domain for this is lunar habitat design, a form of space-based architecture. (Stahl, 1993) This work entails three main aspects:

* Building a software environment to support designers, based on the kind of intelligent hypermedia discussed here.
* Extending the inferencing language to be multi-media, incorporating CAD-style vector graphics, bit mapped graphics, boolean conditionals, and numeric expressions, as well as text.
* Developing a hypermedia-based inheritance mechanism for type inheritance, virtual copying, perspectives, and versioning.

Conclusions

The goal of this work has been to show that hypermedia has more computational potential than is generally thought. Hypermedia is often regarded merely as a user-friendly way of browsing "canned" information. The implementation of predicates shows that the link traversal mechanisms inherent in hypermedia also have the potential for knowledge-based computation. The key to exploiting this potential is twofold: 1) the creation of a powerful query language, and 2) the embedding of queries at nodes to create virtual structures. Given a custom implementation of fine-granularity hypermedia which is object-oriented and designed with the language in mind, only minor changes were needed to make the hypermedia perform full-blown inference once a powerful retrieval language was implemented.

Inference is defined as the combining of facts to derive new facts. In the approach to hypermedia reported here, primitive nodes and links are combined by means of embedded predicates and queries written in the inference language to deduce new (virtual) nodes and links. The end effect for a user is the same as that created in expert systems by means of sophisticated inference engines, namely to infer logical or computational results. The *intelligent hypermedia* approach avoids the need for a separate inference module and spares the user the difficulties of formulating rule bases. The computational mechanisms are handled by extending the normal navigational mechanisms of hypermedia.

The power of intelligent hypermedia was demonstrated in a suggestive way through the implementation of a realistic application, academic advising. Experience with this application and other test cases showed that the development of systems runs up against complications inherent in the domain. No matter how natural the language, an application may require the initial assistance of an experienced programmer or system analyst. However, for simpler tasks and, what is more important, for understanding, modifying, and extending existing applications, the inferencing language appears to be quite easy for users.

The inference language allows users to build a vocabulary of terms for their domain (or for their personal way of looking at things). This vocabulary is open-ended and can be modified or extended at any time. The naming of macros, predicates and queries allows the system designer or user to conceal technical complexities. Definitions can be built up step by step in a modular way to divide up complex concepts. In the end, statements can be formulated in a simple, natural way. The underlying details are always available to the interested user for exploring aspects of an overall problem or for tweaking a definition.

The research uncovered an unanticipated limitation of the original idea of smart nodes. It overcame this limitation with the notion of predicates. Instead of using approaches from procedural programming, the problem was solved in a way that is transparent to the user and results in intuitive results without the user having to be concerned with the use of variables. Using smart links implemented as predicates turned out to be a very powerful and useful notion. The power of this mechanism and of the inference language generally is discussed at length in (Stahl, 1993).

Academic advising is an example of a domain that could be modeled in a rule-based expert system or in navigational hypermedia. The use of intelligent hypermedia combines the computational power of the former with the flexibility of the latter. The inference language will continue to evolve in scope, power, and elegance as it is applied in new computational contexts and new application domains. The goal is to develop a language that can be written and read easily by people who are not experienced computer programmers. This will require considerable experience in observing the language in use in a variety of situations.

References

Eisenberg, M. (1991). Programmable Applications: Interpreter Meets Interface.

Fischer, G., McCall, R., Morch, A. (1989). JANUS: Integrating Hypertext with a Knowledge-Based Design Environment. *Proceedings of Hypertext '89*. New York: ACM.

Fischer, G., Girgensohn, A. (1990). End-User Modifiability in Design Environments. *Human Factors in Computing Systems, CHI '90 Conference Proceedings (Seattle, WA)*. New York: ACM.

Fischer, G., Grudin, J., Lemke, A., McCall, R., Ostwald, J., Reeves, B., Shipman, F. (1991). Supporting Indirect, Collaborative Design with Integrated Knowledge-Based Design Environments. Submitted to *Human-Computer Interaction*.

Halasz, F.G. (1988). Reflections on Notecards: Seven Issues for the Next Generation of Hypermedia Systems. *Communications of the ACM*, Vol. 31, No. 7.

McCall, R. (1989). Mikroplis: A Hypertext System for Design. *Design Studies*, 10 (4), 228-238.

McCall, R., Bennett, P., d'Oronzio, P., Ostwald, J., Shipman, F., Wallace, N. (1990). Phidias: A PHI-based Design Environment Integrating CAD Graphics into Dynamic Hypertext. *Proceedings of the European Conference on Hypertext (ECHT '90)*.

Peper, G., MacIntyre, C., Keenan, J. (1990). Hypertext: A New Approach for Implementing an Expert System. *IBM Internal Technical Liaison Conference on Expert Systems , 1989*.

Stahl, G. (1993). Supporting Interpretation in Design. *Journal of Architecture and Planning Research*, Special issue on Computational Representations of Knowledge. Forthcoming.

Acknowledgments

The ideas in this report grow out of several years of research by Ray McCall of the School of Environmental Design. He was the principal investigator for the grant which supported this work, and most of the theoretical insights of the research originated with him. He also contributed many ideas to this report and reviewed an earlier draft of it.

The report benefited from comments at a presentation to the Human-Computer Communication group and at a CASI Research Symposium.

The research reported here was supported in part by a grant from the Colorado Advanced Software Institute (CASI) for 1990-91. CASI is sponsored in part by the Colorado Advanced Technology Institute (CATI), an agency of the State of Colorado. CATI promotes advanced technology education and research at universities in Colorado for the purpose of economic development.

Appendix: Abstract Syntax of the Inference Language

Following is the abstract syntax in BNF notation of the version of the inference language discussed in the paper. The options that existed in the original Mikroplis query language are underlined.

QUERY OPTIONS

Q : Query	Q::= display Art ***R Prp S which F***, if B, ***with their R***, else Q. | the Nth result of Q | Q and Q
S : Subject	S::= all items | ***all K*** | ***Z*** | Q | S and S
R : Relationship	R::= ***everything*** | ***T*** | converse T | R which are not self | ***the Nth R*** | R which F if B with their R, else R | Z | ***R Prp R*** | ***R and R*** | ***R as a macro*** | R as a predicate
F : Filter	F::= ***Eop C*** | Eop Q | ***have Qop R which F*** [, with them] | are Cop N | e Cop N R which F [, with them] | F Lop F | ***F and which also F*** | contain data type Top | contain no duplicates

MEDIA OPTIONS

C : Character	C::= ***String*** | {text of} Z | substring of C from N for N | C append C
N : Number	N::= Real | the count of results of the query: (Q) | N Nop N | Z
B : Boolean	B::= true | false | N is Cop N | C Eop C | Q Eop Q | not B | B Lop B | Pop Q | R of S which F

HYPERMEDIA OPTIONS

K : Kind	K::= node kind
T : Type	T::= ***link type*** | ***macro*** | predicate
Z : Node	Z::= ***C*** | C if B | Z if B | Q
L : Link	L::= ***Z linked to Z***

OPERATOR OPTIONS

Nop : Numerical	Nop::= plus | minus | times | divided by
Cop : Comparison	Cop::= more than | less than | the same as | at least | no more than | not the same as
Lop : Logical	Lop::= and | or | exclusive or | nand
Qop : Quantity	Qop::= no | all | (at least one) | most | several | a few
Pop : Proposition	Pop::= there are | there are not | there are several
Top : Type	Top::= numeric | non-numeric
Eop : Equality	Eop::= equal | are not equal to | do not contain | contain | are contained in | are not contained in
Prp : Preposition	Prp::= of | about | by | for | from | in | on | over | to | under
Art : Article	Art::= | a | all | an | that | the | those