Computer Science Department

TECHNICAL REPORT

SYLLOG: A KNOWLEDGE BASED DATA
MANAGEMENT SYSTEM

by

Adrian Walker

JUNE 1981

Report No. 034

NEW YORK UNIVERSITY

Department of Computer Science
Courant Institute of Mathematical Sciences
251 MERCER STREET, NEW YORK, N.Y. 10012

SYLLOG: A KNOWLEDGE BASED DATA MANAGEMENT SYSTEM

by

Adrian Walker

JUNE 1981

Report No. 034

This work was supported by the National Science Foundation under Grant No. NSF-MCS-80-04349.

SYLLOG: A KNOWLEDGE BASED DATA MANAGEMENT SYSTEM

by

Adrian Walker

JUNE 1981

Report No. 034

This work was supported by the National Science Foundation under Grant No. NSF-MCS-80-04349.

ABSTRACT

An experimental data base system, called SYLLOG, is described. The system, which has been prototyped in the language SETL, provides a screen-oriented English-like language for use by non-programmers in setting up and using a data base.

To set up a new data base, some standardized English sentences are typed in, and are combined into syllogisms which indicate how the data will be interpreted. Then, once the data have been loaded, the knowledge in the syllogisms is used for retrievals.

The knowledge is used for retrievals by a backchaining algorithm which operates on the syllogisms alone. A tree resulting from the backchaining controls an iterative algorithm which searches the data base. It is shown that the combined backchain-iteration algorithm is correct for schemas in which no syllogism calls itself, and that under this restriction, the query language is at least as powerful as the relational algebra. An extension is described to handle recursive syllogisms, such as those which yield the transitive closure of a relation.

CONTENTS

1. INTRODUCTION

The relational model for data bases [4] effectively frees a user from the details of physical access to data, and it provides an uncluttered framework in which such topics as normalization and the meaning of updates can be discussed [5,7]. Yet, for people who are not programmers or mathematicians, relational data bases can be difficult to use, even when provided with a high level query language such as Query-by-Example [12]. One view of this difficulty is that, while the user knows many common sense rules about the real world situation which a data base describes, the data base system does not (it only has the raw data), so there is plenty of room for misunderstandings.

In work on computer systems which can represent the knowledge of a human expert, [9] the emphasis is on capturing everyday rules of thumb about a particular subject (e.g. medical diagnosis), and then using the rules to make deductions. Very often, such rules are expressed in a form

If conjunction of premises *then* conclusion

and are called production rules. Such rules are chained together to make deductions.

Clearly, both the relational model of data and the production rule model share some features with the first order predicate calculus [3]. A relational data base can be viewed as a set of explicitly listed predicates (a model), and a set of production rules can be thought of as rules of inference for making deductions. However, logic, as a formalism for everyday computer use, is beset by the problem that its notation is difficult for non-specialists to learn and use. For the computer scientist, automatic deduction in first order logic is undecidable in general, and in decidable subcases can consume excessive amounts of computer time in solving quite small problems. Since data bases can be quite large, there is a difficulty in applying automatic theorem proving directly for retrieval.

Yet, substantial progress is being made in bringing techniques from logic into the realm of practical computation. In the programming language PROLOG [8], a program is a set of ordered sequences of logical clauses. A clause can be a simple ground (variable-free) assertion, which can be regarded as a row in a relation in a data base, or it can be a conjunction of predicates containing variables which implies a conclusion predicate, in which case it can be regarded as a production rule. So, for small data bases, PROLOG contains the means to store data, and to make deductions about the data using production rules.

There are two drawbacks to using PROLOG directly for a practical data base system. First, PROLOG notation, though natural for computer scientists, is probably difficult for most non-specialists to use. For example, an otherwise correct retrieval program, in which the order of two clauses is reversed, can enter an infinite loop. Second, the execution mechanism in current implementations is a depth first backtrack search over internal (or virtual) storage; the problem of efficiently searching external (e.g. disk) storage has yet to be addressed.

In section 2 of this paper, we describe a screen-oriented, English-like language for setting up and using a data base. The language consists of syllogisms. In section 3, we describe a backchaining algorithm which forms the first stage in query processing. Backchaining deals only with intensional syllogisms, not with the extensional data in relations, and it produces a tree as its output. In the second stage of query processing, this tree is used to control an iterative search of the extensional data base. It is shown that the two-stage backchain-iteration algorithm, applied to non-recursive syllogisms, is correct, and has the power of the relational algebra. In section 4, the backchain-iteration algorithm is extended to deal with recursive syllogisms, such as those which yield the transitive closure of a relation. Thus the SYLLOG language becomes strictly more powerful than the relational algebra.

2. THE SYLLOG LANGUAGE

This section describes the SYLLOG language, from the user's point of view, by means of an example of setting up and using a data base.

Suppose we are interested in knowledge and data about cities, about ways of travelling from one city to another, and about ways of getting around inside a city. Then we will want to know about statements such as "Greenwich Village is in New York", "uptown is in New York", and knowledge such as "if two places are in the same city then one can take a taxi from one to the other".

2.1 Data Definition : setting up a schema

In SYLLOG, one says that a new data base will be concerned with such facts by typing in

```
_village is in _New-York
_uptown is in _New-York
----------------------------------------
can take a taxi from _village to _uptown
```

and we call this a syllogism. The underlines in front of words indicate example items. Thus the sentence

```
_village is in _New-York
```

can be read as "the data base will be concerned, amongst other things, with something being in something else, such as the village being in New-York". The whole syllogism can be understood as "if a place is in a given city, and a second place is in the same city, then one can take a taxi from the first place to the second place".

At this stage, the system contains no data, just the statement that it will contain two relations "...is in..." and "can take a taxi from ... to ...", and some knowledge about the second relation given some data in the first.

2.2 Adding Data

One could now type in some data like this

```
_village is in _newyork
--------------------------------
 uptown              New-York
 village             New-York
 white-house         Washington
 patent-office       Washington
```

However, it is not necessary to type the first sentence.The standard SYLLOG prompt to the user is of the form

```
Make a command using these and other sentences:
_village is in _New-York
can take a taxi from _village to _uptown
```

Thus, to make the above command to add some data, one first deletes the sentence "can take a taxi..." from the screen, then types in an underline followed by the data. If the data are in a file, one can give the command

```
_village is in _New-York
----------------------------
   <file name>
```

which adds the contents of the file to the "is in" relation.

2.3 Querying the Data Base

At this point, the system contains some data and some elementary knowledge about how to use the data. Suppose we want a list of places in Washington. The SYLLOG prompt places the prototype sentences

```
_village is in _New-York
can take a taxi from _village to _uptown
```

on the screen. We then delete the second sentence and insert an underline to get

```
_village is in _New-York
------------------------
```

This is a command to print out all places in all cities, so before executing it we change _New-York so that the screen reads

```
_village is in Washington
-------------------------
```

Note that _village is only a place holder here; the query would have the same effect if we used _white-house or _x instead. However, if we changed Washington to New-York, we would have a different query.

We now indicate that the query is to be executed. Data appear on the screen below the command like this.

```
_village is in Washington
-------------------------
patent-office  Washington
white-house    Washington
```

and we have answered the query "which places are in Washington".

Note that we could also have made the query

```
village is in Washington
------------------------
```

i.e. "is the village in Washington ?". The resulting screen is

```
village is in Washington
------------------------
    EMPTY ANSWER
```

while if we asked

```
white-house is in Washington
----------------------------
```

the resulting screen is the confirmation

```
white house is in Washington
---------------------------------
white-house          Washington
```

Now suppose we want a list of places and the cities which they are <u>not</u> in. As before, SYLLOG prompts with the standard sentences

```
_village is in _New-York
can take a taxi from _village to _uptown
```

We replace the second sentence on the screen by an underline, and change the first sentence by inserting "not", yielding the query

```
_village is not in _New-York
----------------------------
```

When the query has been executed, the screen shows

```
_village is not in _New-York
----------------------------
patent-office       New-York
uptown              Washington
village             Washington
white-house         New-York
```

So far, we have just queried the relation "is in" into which we loaded some data. Now suppose we are interested in getting from place to place by taxi. As usual SYLLOG prompts with the sentences

```
_village is in _New-York
can take a taxi from _village to _uptown
```

from which we can construct, on the screen, the query

```
can take a taxi from _village to _uptown
----------------------------------------
```

After the query is executed, the screen shows

```
can take a taxi from _village to _uptown
----------------------------------------
patent-office       patent-office
patent-office       white-house
uptown              uptown
uptown              village
village             uptown
village             village
white-house         patent-office
white-house         white-house
```

The answer is correct, at least in that it reflects the data and the syllogism

```
_village is in _New-York
_uptown is in _New-York
----------------------------------------
can take a taxi from _village to _uptown
```

from which it was computed. However, the answer is lacking in real world knowledge; people don't take taxis from a place to the same place. This fact can be included by changing the syllogism to

```
_village is in _New-York
_uptown is in _New-York
_village not EQUAL _uptown
-------------------------------------
can take a taxi from _village to _uptown
```

where EQUAL is a built-in test in SYLLOG. (We describe how a syllogism can be changed in the next section). With the new syllogism, the query remains the same, and the screen containing the answer is

```
can take a taxi from _village to _uptown
----------------------------------------
patent-office       white-house
uptown              village
village             uptown
white-house         patent-office
```

Note that, for real situations, further refinement of the syllogisms might be needed; for example, a place might have two names.

2.4 Querying, Adding, Deleting and Changing Syllogisms

In the last section, we modified a syllogism about taking a taxi by placing an extra condition in its premise. SYLLOG allows syllogisms to be queried and modified.

To query the knowledge base of syllogisms, one starts, as usual , with the prompt, which consists of the sentences known to the system. In this case we are interested in a rule, or rules, about taking taxis, so we just leave the sentence

```
can take a taxi from _village to _uptown
```

on the screen. This is understood as a command to list all of the syllogisms having this sentence (or one like it but for renaming of _village and _uptown) as a conclusion. Thus the rule

```
_village is in _New-York
_uptown is in _New-York
----------------------------------------
can take a taxi from _village to _uptown
```

appears on the screen. The syllogism is now edited, on the screen, to its new form

```
_village is in _New-York
_uptown is in _New-York
_village not EQUAL _uptown
----------------------------------------
can take a taxi from _village to _uptown
```

and replaces the old syllogism.

An entirely new syllogism can simply be typed in, while a syllogism can be deleted by calling it up on the screeen with a query command, and then erasing it from the screen.

Although it is easy for the user to modify the syllogisms, this should be done with some thought, since some data may be erased in the process. SYLLOG marks each fact in a relation according to whether it has been asserted in an add or change command, or has been deduced via the syllogisms during a query. When a syllogism is added, changed, or deleted, all of the affected deduced data is erased. If a syllogism which contains the last mention of a particular sentence is deleted, then that sentence is dropped from the prompt list, unless there are facts which have been asserted about it.

2.5 Further Querying of the Data Base

Suppose we now add some data and a syllogism about travelling by train. We add the data

```
can go by train from _village to _Newark
----------------------------------------
village               Hoboken
Hoboken               Newark
Newark                Washington
```

and the syllogism

```
can go by train from _village to _Hoboken
can go by train from _Hoboken to _Newark
-----------------------------------------
can go by train from _village to _Newark
```

This syllogism is special, in that the sentence in the conclusion also appears in the premise. We say that the syllogism is <u>recursive</u>.

The SYLLOG prompt is now an invitation to make a command using the sentences

```
_village is in _New-York
can take a taxi from _village to _uptown
can go by train from _village to _Newark
```

To ask which places we can get to by train from Hoboken, we form the query

```
can go by train from Hoboken to _Newark
---------------------------------------
```

When the query has been made, the screen shows

```
can go by train from Hoboken to _Newark
---------------------------------------
Hoboken               Newark
Hoboken               Washington
```

Note that Hoboken-Washington is not in the data we asserted. It has been deduced by using the syllogism to bridge Newark in Hoboken-Newark-Washington.

If we now form the query

```
can go by train from Washington to _village
-------------------------------------------
```

we get EMPTY ANSWER. Strictly, this is correct, since the system only knows about trains in one direction. However, it is not what is really wanted, so we add the syllogism

```
can go by train from _village to _Newark
----------------------------------------
can go by train from _Newark to _village
```

which says that any time we can go from A to B by train, we can also go from B to A. If we now repeat the question about which places we can go to by train from Washington, we get the answer

```
can go by train from Washington to _Hoboken
-------------------------------------------
Washington          Hoboken
Washington          Newark
Washington          village
Washington          Washington
```

which, apart from the last row, is reasonable. The last row could be suppressed, as in the taxi example in section 2.3, by modifying the syllogisms.

2.6 Deleting and Changing Data

Data can be deleted from a data base in SYLLOG by simply bringing it to the screen using a query, and then erasing it from the screen. This works directly for asserted data. However, data which have been deduced using the syllogisms cannot be deleted in this way, and a warning message results.

Similarly, asserted data may be changed by first using a query to bring it to the screen. Attempts to change deduced data yield a warning message.

3. QUERY EVALUATION BY BACKCHAIN-ITERATION

In the last section we described SYLLOG from the point of view of the person who uses the system. This section describes how a query is processed by SYLLOG, in the case that the query syllogisms are not recursive. A proof of correctness of the query algorithm is given, and it is shown that non-recursive collections of syllogisms have at least the power of the relational algebra. Section 4 treats the case in which recursion present.

3.1 An Example

Syllogisms are stored internally in SYLLOG in the form of production rules. For example the syllogism

```
_village is in _New-York
_uptown is in _New-York
----------------------------------------
can take a taxi from _village to _uptown
```

is stored in a form corresponding to

$$C_2(x,y) <- I_1(x,z)I_1(y,z)$$

where C_2 and I_1 are system-generated relation names. We call this form a rule, and we write a rule with the conclusion on the left for convenience in discussing backchaining.

As mentioned in section 2, the system stores a current list of prompting sentences, which contains a representative sentence for each sentence which has been used in a syllogism or a command. If two sentences differ only by renaming of variables, or by instantiation of variables, or by the presence of "not", only one representative is stored in the prompt list. The list is indexed by system-generated relation names, such as C_2 and I_1 above. Thus a sentence is translated into its relation form by a simple pattern match followed by a table lookup, and a relation is translated into a sentence by a table lookup followed by the substitution of the appropriate variables or constants into the sentence. Translations between the rule and syllogism forms are then simply made sentence by sentence or relation by relation. While the English-like properties of the SYLLOG language should be easy for people to use, it is plain that very little computer time or space is needed to translate between syllogisms and rules.

Suppose that only the two syllogisms

```
_village is in _New-York
_uptown is in _New-York
----------------------------------------
can take a taxi from _village to _uptown
```

and

```
can get a taxi from _uptown to _village
can go by train from _village to _Hoboken
-----------------------------------------
can go from _uptown to _Hoboken
```

are present, and that they are represented by the rules

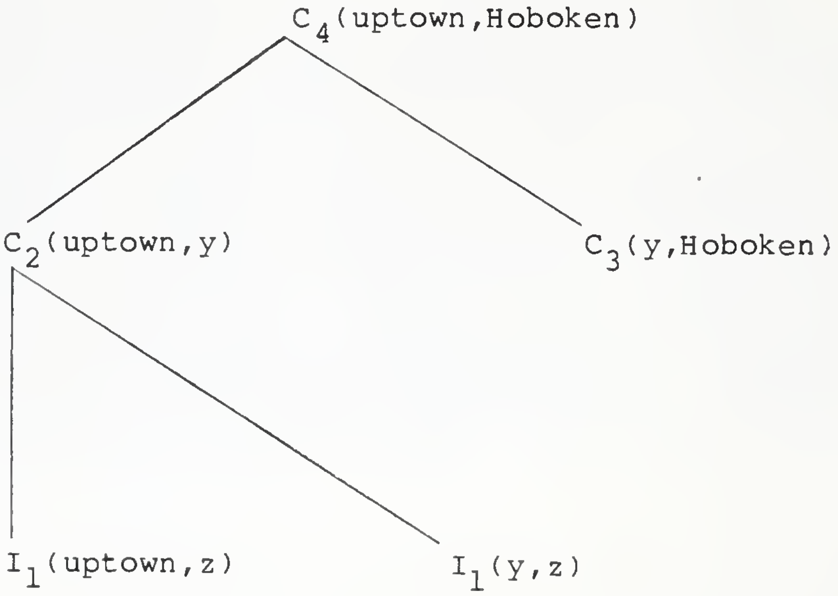
$C_2(x,y) <- I_1(x,z)I_1(y,z)$

$C_4(x,z) <- C_2(x,y)C_3(y,z)$.

A query

```
can go from uptown to Hoboken
-----------------------------
```

is translated into C_4(uptown,Hoboken), and causes the following tree to be constructed from the rules:



The tree is constructed using only the query and the rules, without reference to the facts in the data base. Next, the tree is interpreted as a query program, and executed as follows. Each node of the tree is assigned an initially empty set, called its <u>extension</u>. Then, each leaf node extension is made equal to the set of tuples, from the corresponding asserted data, relevant to the predicate at the node. For example, the leftmost leaf in the tree gets the rows from the "is in" relation which start with "uptown". Next, the lowest level of the tree is executed, in this case using an operation equivalent to the relational algebra **join** (we write * for join) as

$$I_1(\text{uptown},z) * I_1(y,z)$$

and the result

uptown	uptown
uptown	village

is placed in the extension of C_2. Then, the upper level of the tree is executed, placing

uptown	Hoboken

in the extension of C_4. Now the extension of the root has been computed, and it is printed as the answer.

In the example, the tree represents a conjunctive query. In general, a query may contain disjuncts, in which two or more rules contribute to the extension of a node, and negations, in which case a node is extended by adding tuples which are not in the set calculated by a rule. Thus the backchain procedure yields an and-or-not tree. Note that, in the construction of the tree, selection arguments in the query (e.g. uptown, Hoboken) are propagated downwards.

3.2 Definitions

In this section, we set down the definitions which are needed to prove the correctness of the backchain-iteration algorithm.

We use x,...,z as individual variables, a,...,d as constants, and x,...,z with subscripts to denote ordered lists of variables and constants. A substitution is a function s, from variables and constants to variables and constants, such that s(a)=a for each constant a.

A knowledge base K is a finite set of clauses, each of the form

$$A(x_0)<-B_1(y_1)..B_m(y_m)-C_1(z_1)..-C_n(z_n)$$

where m+n is greater or equal 0, $A(x_0)$ and $B_i(y_i)$, i=1..m, are positive literals, (e.g P(x,y)), and $-C_j(z_j)$, j=1..n, are negative literals (e.g. -P(x,y)).

If m+n > 0 the clause is a rule. If m+n = 0, then the clause in an assertion. We assume that assertions contain no variables, and, if

$$A(x_0)<-$$

is an assertion in K, then K contains no rule with $A(x_0)$ on the left.

Each rule is such that, if a variable appears in x_0, then it appears in some y_i or z_j. Also, if a variable appears in some z_j, then it also appears in some y_i.

Note that given a clause in which some variable appears in a z_j but not in a y_i, we can often replace the clause by a set of clauses in which each variable in a z_j does appear in a y_i. For example, we can replace

$$P(x,y) <- -Q(x,y)$$

by the clauses

$$P(x,y) <- Q_1(x)Q_2(y)-Q(x,y)$$

$$Q_1(x) <- Q(x,y)$$

$$Q_2(y) <- Q(x,y).$$

Let s be a substitution. We say that $A(s(x_0))$ follows from K, written $A(s(x_0))$ -! K, if

(i) $A(s(x_0))$ <- , or

(ii) there is a rule

$A(x_0)$<-$B_1(y_1)..B_m(y_m)$-$C_1(z_1)$..-$C_n(z_n)$

in K such that

a) $B_i(s(y_i))$ -! K for i = 1..m, and

b) it is not the case that $C_j(s(z_j))$ -! K

for any j in {1,..,n}.

Where K is understood, we write -! instead of -! K.

A program for $A(x_0)$ is a tree with root $A(x_0)$ defined by the following. If there exists a rule

$A(x_0')$<-$B_1(y_1)..B_m(y_m)$-$C_1(z_1)$..-$C_n(z_n)$

and a substitution s such that $s(x_0') = x_0$, then (using s to rename variables which are already in the tree) add

$A(x_0)$
<-$B_1(s(y_1))..B_m(s(y_m))$-$C_1(s(z_1))$..-$C_n(s(z_n))$

to the tree below the root. If there is a program for $B_i(s(y_i))$ or $C_j(s(z_j))$ then add that to the tree also.

Note that the program tree is finite only if no rule eventually calls itself as a program. We assume this to be the case for now.

The extension of a program tree is defined as follows:

(1) Each node is assigned an empty set, called its extension.

(2) For each leaf $C(y_0)$, if $C(s(y_0))$<- is in K for some substitution s, then place $C(s(y_0))$ in the extension of the leaf. Then mark the leaf extended.

(3) If

$$A(x_0)<-B_1(y_1)..B_n(y_n)-C_1(z_1)..-C_n(z_n)$$

is a rule in the program tree, the nodes $B_i(y_i)$ and $C_j(z_j)$ have been extended, $B_i(s(y_i))$ is in the extension of $B_i(y_i)$ and $C_j(s(z_j))$ is not in the extension of $C_j(z_j)$, then place $A(s(x_0))$ in the extension of $A(x_0)$. When all such placings have been made, mark $A(x_0)$ extended.

(4) Repeat (3) until the root of the tree is marked extended.

3.3 Correctness of Backchain-Iteration

The definitions in the last section allow us to prove that an answer tuple is in the output from a query if, and only if, it follows from the knowledge base. Formally, this is stated as:

Theorem 1 Let T be an extended program tree with root $A(x_0)$, and let s be a substitution. Then $A(s(x_0))$ is in the extension of the root of T iff $A(s(x_0))$-!.

Proof Without loss of generality, assume that there is just one clause at the root of the tree, and let it be

$A(x_0)$<-$B_1(y_1)..B_n(y_n)$-$C_1(z_1)..$-$C_n(z_n)$

Case m+n=0: The extension of $A(x_0)$ is defined as

{$A(x_0')$! there exists a substitution s such
that $s(x_0)=x_0'$ and $A(x_0')$<-}

so, from the definition of -!, $A(x_0')$ is in the extension of $A(x_0)$ at the root of the tree iff $A(x_0')$-!.

Case m+n>0: Suppose $A(s(x_0))$ is in the extension of the root of T, for some s. Then, by definition of the extension of T, $B_i(s(y_i))$ is in the extension of the node marked $B_i(y_i)$, and $C_j(s(z_j))$ is not in the extension of the node marked $C_j(z_j)$. As an inductive hypothesis, assume that the presence of $B_i(s(y_i))$ in the extension of the node marked $B_i(y_i)$ implies that $B_i(s(y_i))$-!, and that the absence of $C_j(s(z_j))$ from the extension of the node marked $C_j(z_j)$ implies that it is not the case that $C_j(s(z_j))$-!. Then, from the definition of -!, $A(s(x_0))$-!.

A similar argument establishes that if $A(s(x_0))$-! then $A(s(x_0))$ is in the extension of the root of T. []

The theorem applies to the case in which no rule calls itself, i.e. in which the set of rules is such that it is not possible for a literal to be repeated along a path in a program tree. A more general case is discussed in section 4.

3.4 Power of Non-Recursive Backchain-Iteration is that of the Relational Algebra

This section shows that if a query can be written in the relational algebra [5], then it can also be written in SYLLOG. (We shall sometimes refer to the relational algebra simply as the algebra). Since algebra expressions are (tacitly) formulated to be non-recursive, we shall see that the corresponding sets of syllogisms are also free of recursion. Thus SYLLOG, without recursive syllogisms, has at least the power of the algebra.

Following [9], we take the five operations **union**, **set difference**, **cartesian product**, **project**, and **select** to define the algebra. So SYLLOG is as powerful as the algebra if it can be shown to simulate any inductive combination of these five operations. However it would be quite inconvenient in practice to use only SYLLOG equivalents of these, so we also show how the relational algebra operations **natural join**, **intersection**, and **quotient** can be written in SYLLOG.

Theorem 2 If a query can be written in the relational algebra, then it can also be written in SYLLOG.

Proof We shall use the internal rule form, since the translation between this and the syllogism form is straightforward. For each of the algebra operations, we show an equivalent rule, or set of rules.

1. Union In the algebra,

$$R = R_1 \text{ U } R_2 = \{ x_k \ ! \ R_1(x_k) \text{ v } R_2(x_k) \}$$

In rules

$$R(x_k) <- R_1(x_k)$$

$$R(x_k) <- R_2(x_k).$$

2. Set difference

$$R=R_1-R_2=\{x_k!R_1(x_k) \text{ and not } R_2(x_k)\}$$

$$R(x_k) <- R_1(x_k) -R_2(x_k)$$

3. Cartesian product

$$R=R_1 \text{ X } R_2=\{<x_k,x_j>!R_1(x_k) \text{ and } R_2(x_j)\}$$

$$R(x_k,x_j)<-R_1(x_k)R_2(x_j)$$

4. Project

$R = PROJ_{i_1,..,i_m}\ R_1 =$

$\{<x_{i_1},..,x_{i_m}>$!

there exists $<y_1,..,y_n>$ in R_1

such that $x_{i_j}=y_{i_j}$ for j=1..m }

where x_{i_j}, y_k and y_{i_j} are domain variables [10].

$R(x_{i_1},..,x_{i_m}) <- R_1(y_1,..,y_n)$.

5. Select

$R = SEL_P\ R_1(x_k) =$

$\{\ x_k$! $R(x_k)$ and $P(x_k)\ \}$

where P is a predicate defined in terms of:

(i) operands that are constants or variables,

(ii) lexical or arithmetic comparison operators

<, =, >, < or =, $\neq$, > or =,

(iii) logical operators **and**, **or**, **not**.

We give some examples of translation of select expressions into SYLLOG. Using these, it is straightforward to inductively decompose, then translate, an arbitrary select. Let @ and % be comparison operators.

$R(x,y) = SEL_{x@y}\ R_1(x,y)=$

$\{<x,y>$! $R_1(x,y)$ and $x@y\}$

$R(x,y) <- R_1(x,y)\ @(x,y)$

$R(x,y) = SEL_{x@y \text{ and } x\%y}\ R_1(x,y) =$

$\{<x,y>!\ R_1(x,y)$ and $x@y$ and $x\%y\}$

$R(x,y)<-R_1(x,y)@(x,y)\%(x,y)$

$$R(x,y) = \mathrm{SEL}_{x@y \text{ or } x\%y}\ R_1(x,y) =$$

$$\{<x,y>!\ R_1(x,y) \text{ and } (x@y \text{ or } x\%y)\}$$

$$R(x,y) <- R(x,y)\ @(x,y)$$

$$R(x,y) <- R(x,y)\ \%(x,y)$$

$$R(x) = \mathrm{SEL}_{\mathrm{not}(x@c)} R_1(x) =$$

$$\{\ x\ !\ R_1(x) \text{ and not } x@c\ \}$$

$$R(x) <- R_1(x)\ -@(x,c)$$

The above five operations are sufficient to define a basis for the relational algebra [10], and it is clear that the translation of any algebraic expression into a set of SYLLOG rules can be constructed in a straightforward way using the indicated translations for individual operations. It follows easily from Theorem 1 that the constructed SYLLOG rules will yield the same result as the relational algebra expression. This completes the proof of Theorem 2. []

We now give examples of how to write the algebra operators **natural join**, **intersection**, and **quotient** as SYLLOG rules (and hence as syllogisms).

The **natural join**

$$R = R_1 * R_2 =$$

$$\{<x,y,z>\ !\ R_1(x,y) \text{ and } R_2(x,z)\}$$

is written as the SYLLOG rule

$$R(x,y,z) <- R_1(x,y)\ R_2(x,z).$$

The **intersection**

$$R = R_1\ \&\ R_2 =$$

$$\{\ x_k\ !\ R_1(x_k) \text{ and } R_2(x_k)\ \}$$

is written

$$R(x_k) <- R_1(x_k) R_2(x_k)$$

The quotient

$$T = R/S$$

which is defined by

$$\{ x \;!\; y \text{ in } S \text{ implies } \langle x,y\rangle \text{ in } R \}$$

yields the set of rules

$$T(x) \leftarrow R_1(x) \; -R_2(x)$$

$$R_1(x) \leftarrow R(x,y)$$

$$R_2(x) \leftarrow R_1(x) \; S(y) \; -R(x,y)$$

Thus, not only is non-recursive SYLLOG formally as powerful as the relational algebra, but all of the algebra operators except division have simple transliterations in SYLLOG. Since the quotient operator is not widely used, its indirect expression in SYLLOG is a minor disadvantage.

4. BACKCHAIN-ITERATION and RECURSIVE SYLLOGISMS

In section 2.5 we used a syllogism

```
can go by train from _village to _ Hoboken
can go by train from _Hoboken to _Newark
------------------------------------------
can go by train from _village to _Newark
```

Stated as a rule, this is

```
C(x,z) <- C(x,y) C(y,z)
```

and it is clearly recursive. In fact, the rule expresses the transitive closure of the asserted tuples in C, an operation which cannot be expressed in the relational algebra [1]. Thus, if we allow such rules, SYLLOG is strictly more powerful than the relational algebra. This section describes a technique for extending the definitions of section 3.2 to cover the recursive case.

4.1 An Example

Suppose we have a knowledge base containing the rule shown above, and we relax our constraint that a predicate should not be both the subject of an assertion and the left side of a rule. Then we can also assert that some tuples are in C. Suppose we assert that the tuples

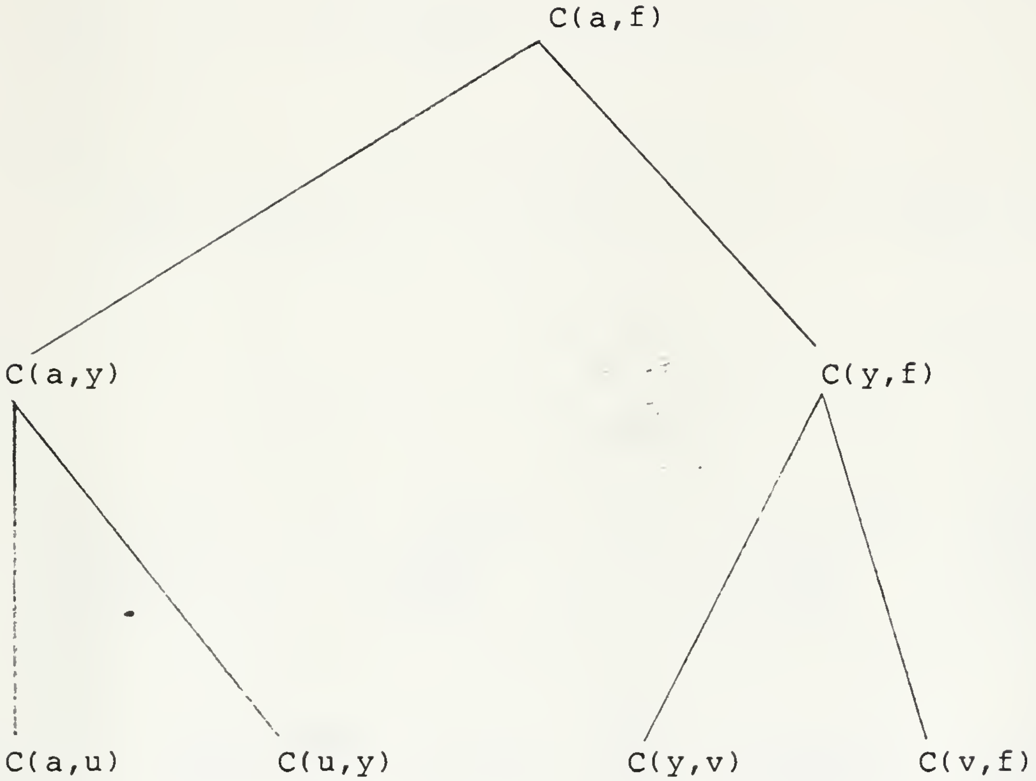
a	b
b	c
c	d
d	e
e	f

are in C, so that the knowledge base contains the assertions C(a,b)<-, ..., C(e,f)<-, together with the rule mentioned above.

If we now ask the question

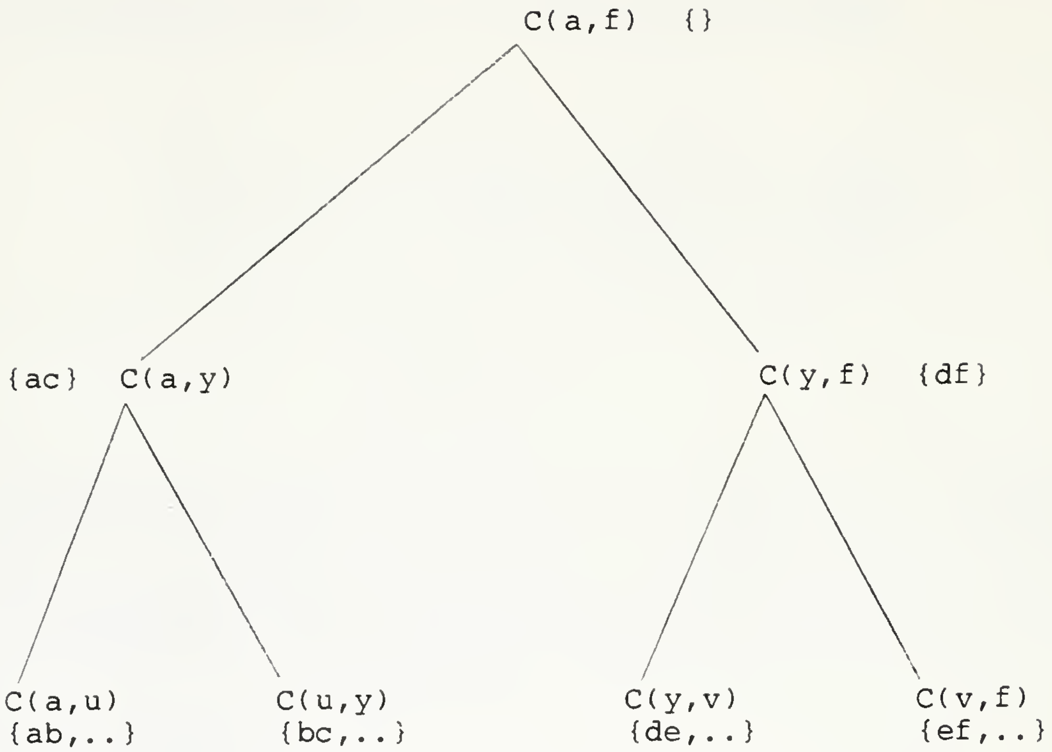
```
can go by train from a to f
---------------------------
```

SYLLOG will proceed to backchain from C(a,f). Rather than producing an infinite tree, the backchain produces the program tree



In growing the tree downwards, the stopping criterion is to not add to the tree a rule which is isomorphic (in a sense to be defined in section 4.2) to a rule above it.

Next, we do the iteration part of backchain-iteration, as described in section 3.2. Writing the extensions next to the tree gives



where {} denotes the empty set. Now suppose that, each time a tuple is placed in the extension of a node, it is also placed in the set of tuples of C. Then, after the first iteration, C contains the extra rows

a c
d f

It is easy to see that, if we now repeat the computation of the extension, the root extension will be made to contain the tuple a f, yielding the required positive answer.

So, a method of dealing with a recursive rule is to compute the extension of a finite tree not once, as in the non-recursive case, but repeatedly, until it is noted that no change occurs in the extension of any node. Clearly, there are ways of refining this to make it more efficient, but the principle is straightforward.

Note that, in this example, backchain-iteration instantiates to a program which is a two-sided graph search with specified end nodes. Thus if the graph of the relation C contains subgraphs which are not connected to the end nodes a or f, then the corresponding subrelations are not searched.

4.2 Definition of Finite Backchaining for Recursion

In our example, we mentioned that backchaining is halted when we are about to add to the program tree a rule which would be isomorphic to one above it. Our working definition of isomorphic is as follows:

Let Rule 1

$$A(x_0)<-B_1(y_1)..B_n(y_n)-C_1(z_1)..-C_n(z_N)$$

be part of the tree, and let Rule 2, which we are deciding whether or not to add, be

$$A'(x_0')<-$$
$$B'_1(y_1')..B'_n(y_n')-C'_1(z_1')..-C'_n(z_N')$$

If $A \neq A'$, $B_i \neq B_i'$, or $C_j \neq C_j'$ for some i and j

(i.e. if the rules cannot be arranged, preserving negation, to have the same predicate names in the same left to right order), then the rules are not isomorphic. If the rules do have the same sequence of predicate names, but there is no substitution s such that $s(x_0')=x_0$, $s(y_i')=y_i$, and $s(z_j')=z_j$ for all i and j, then the rules are still not isomorphic. If there is such an s, let

$$c = CARD\{ x ! s(x)=x, x \text{ is a constant}\}$$

and

$$v = CARD\{<x,b> ! s(x)=b, x \text{ is a variable and } b \text{ is a constant}\},$$

where CARD denotes the cardinality of a set, and say that the rules are isomorphic unless c is greater equal 1 and v is greater equal 1.

To see how this definition works in halting the backtrack in the example in section 4.1, suppose we are at a stage when the program tree contains only

C(a,f) <- C(a,y) C(y,f) (Rule 1)

and that we are considering whether or not to add some instance of

C(x,z) <- C(x,u) C(u,z)

below C(a,y). Clearly, the required instance is

C(a,z) <- C(a,u) C(u,z) (Rule 2).

Rules 1 and 2 become identical under the substitution s(a)=a, s(z)=f, s(u)=y. This substitution has c greater equal 1 by reason of s(a)=a and v greater equal 1 by reason of s(z)=f, so the two rules are not isomorphic, and Rule 2 is added to the tree.

Next, suppose that Rule 2 is in the tree, and that we are considering whether to add

C(a,v) <- C(a,w) C(w,v) (Rule 3)

below the C(a,u) in Rule 2. Rule 3 and Rule 2 become identical under the substitution s(a)=a, s(v)=z, s(w)=u. For this s, c is greater equal 1, but v is less than 1, so Rules 2 and 3 are isomorphic, and Rule 3 is not added to the program tree.

4.3 Iteration for Recursion

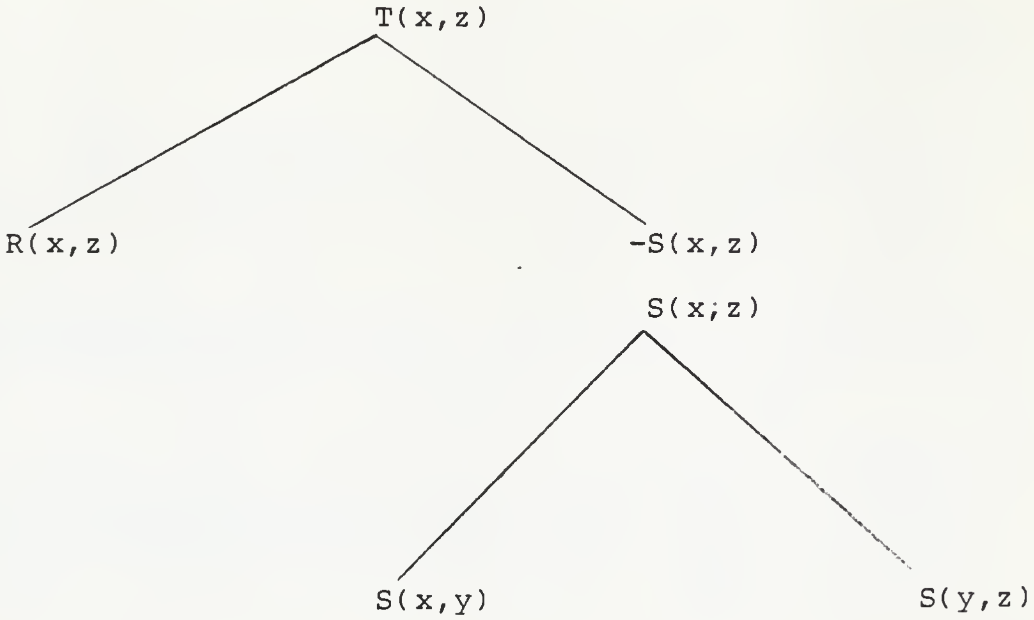
In the present, experimental, version of SYLLOG, the program tree extension iteration is repeated until no extension is changed in a full bottom to top scan of the tree. This is wasteful in the absence of recursion, since twice as much computation may be done as is needed; a first scan is made to get the answer, and a second scan is made to check that it is indeed the whole answer. Theorem 1 assures us that, if we detect, at backchain time, that there is no recursion, then a single extension scan is sufficient. On the other hand, if recursion is present, it is easy to adapt the example in section 4.1 to show that we cannot limit the number of extension scans in advance.

The repeated scan of the whole tree, each rule at each level being executed once at each scan, can actually be incorrect if both recursion and negation are present, (although there is a simple way making it correct). To see this, consider the rules

```
T(x,z) <- R(x,z) -S(x,z)

S(x,z) <- S(x,y) S(y,z)
```

together with the data R(a,d), S(a,b), S(b,c), S(c,d). The correct answer to the query T(x,z) is EMPTY, but the first scan of the program tree



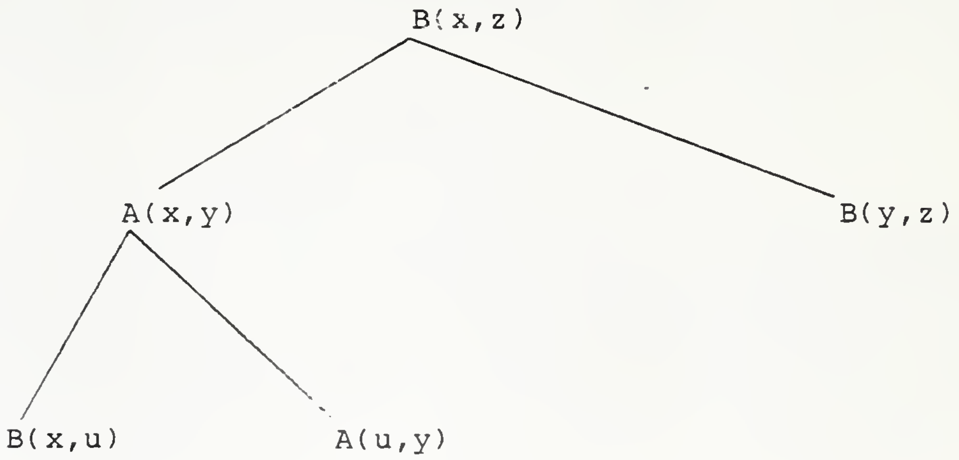
will place the tuple **a d** in the extension of the root, and subsequent scans will not remove it. One way of correcting this is to repeatedly extend the S subtree before extending the root of the tree.

It can also be necessary, in some cases, to repeatedly scan a local subtree. For example, if the rules are

A(x,z) <- B(x,y) A(y,z)

B(x,z) < - A(x,y) B(y,z)

and the data are A(a,b), A(c,d), B(b,c), B(d,e), then the tree



must be scanned twice to determine that B(a,e). If this tree is a subtree, and its root is the subject of negation higher in the main tree, then the local repeated scan must be made before scanning higher.

Since it is not easy to find real examples of data base retrievals which require recursion beyond the simple form

R(x,z) <- R(x,y) R(y,z)

needed for transitive closure, a good compromise between generality and computational cost appears to be to reject backchain trees which contain more complicated recursions, and to execute the admissible ones by repeated local scanning only.

5. CONCLUSIONS

The SYLLOG system, which has been prototyped in SETL, provides a simple, English-like language in which a non-programmer can set up and use a data base. The language prompts the user by showing a set of standardized English sentences on the screen, and the user makes a command by choosing one of these sentences and modifying it. The language is designed for interactive use at a screen, and would be most suitable for use with a light pen plus occasional key strokes. So far, the language has been implemented using a line editor, and separately using a visual editor. The number of key strokes needed is quite small.

The standardized English-like sentences are grouped into syllogisms, which function as a way of encoding knowledge about a particular domain,- e.g travel, dentistry etc.,- for use in query processing. A query is a single sentence, and it triggers a search of the domain knowledge, followed by a search of a relational data base. Thus the domain knowledge mediates between the user and the data base.

The order in which syllogisms are made known to SYLLOG has no effect on the result of a query, so that the language may fairly be said to be non-procedural. Recursive syllogisms are allowed, hence the power of SYLLOG exceeds that of the relational algebra (and of the relational calculus), yet there is no possibility for an inexperienced user to take the system into an infinite loop.

The two preprocessing stages for a SYLLOG query, namely the translation from English to predicate form followed by the construction of a program tree, are reasonably straightforward, and require little space or time in the computer. Once a program tree has been constructed, the computer resources needed for its execution are similar to those needed for any relational data base system.

We note that a method for converting recursive rules in a data base intension into iterative programs over the extension has also been suggested in [2]. In that method, relations are partitioned into those which are derived and those which are asserted, whereas we find it useful to mark individual tuples as either derived or asserted. Also, in [2], a rule must be regular, in the sense that the premise may contain at most one derived relation; hence either the user must be restricted to only declare regular rules, or a general method must be found to convert irregular rules into regular ones. As stated in [2], "finding a good program from a recursive query (graph) is a fruitful area of research". Our finite backchain algorithm appears to be a step in this direction.

Although the knowledge contained in a set of syllogisms greatly simplifies matters for the user, this paper has only described the use of the knowledge for query processing. Syllogisms can also be used for type-checking, and to express constraints which can be automatically enforced whenever an update is made. The use of syllogisms to express constraints is discussed in [11]. The related matter of updates into syllogistically defined views of a data base remains as an interesting topic for future work.

6. ACKNOWLEDGEMENTS

It is a pleasure to acknowledge many friendly conversations with the SETL group at New York University. The SETL language allowed the experimental version of SYLLOG to be developed with far less time spent on programming and debugging than would have been needed in other languages. The particular features of SETL which seem to have helped the most are

(1) the primitive, nestable data types **tuple**, and (of course!) **set**,

(2) the non-procedural **such that** notation for defining tuples and sets, and

(3) the provision of the omega value **undefined**, which was very useful in debugging.

Special thanks go to J. Schwartz for timely advice, and to D. Shields for providing some SETL tools and plenty of in depth knowledge about the VAX implementation of SETL.

7. REFERENCES

[1] Aho, A.V., and Ullman, J.D. Universality of data retrieval languages. Proc. 6th Annual Symp. Princ. Prog. Lang., 1979,110-119.

[2] Chang, C.L. On evaluation of queries containing derived relations in a relational data base. Advances in Data Base Theory, Vol 1, Eds H. Gallaire, J. Minker, J.M. Nicolas, Plenum, New York, 1981, 235-260.

[3] Chang, C.L. and Lee, C.T. Symbolic Logic and Mechanical Theorem Proving. Academic Press, 1973.

[4] Codd, E.F. A relational model of data for large shared data banks. CACM **13**, 6, 1970, 377-387.

[5] Codd, E.F. Further normalization of the data base relational model. Courant Computer Science Symposium 6: Data Base Systems, Prentice-Hall, Englewood Cliffs, New Jersey, 1971, 33-64.

[6] Codd, E.F. Relational completeness of data base sublanguages. ibid, 65-98.

[7] Fagin, R. Multivalued dependencies and a new normal form for relational data bases. ACM TODS **2**:3, 1977, 262-278.

[8] Kowalski, R. Logic for Problem Solving, Elsevier North Holland, New York, 1979.

[9] Shortliffe, E. Computer Based Medical Consultations: MYCIN, American Elsevier, New York, 1976.

[10] Ullman, J.D. Principles of Data Base Systems. Computer Science Press, Potomac, Maryland, 1980.

[11] Walker, A.D., and Salveter S. A transaction scheme transform which preserves data base integrity without undoing updates. Report, Computer Science Department, SUNY at Stony Brook, NY, July 1981.

[12] Zloof, M.M. Query-by-Example: a data base language. IBM Systems Journal, **16**:4, 324,343.

NYU c.1
Comp. Sci. Dept.
TR-034
Walker
SYLLOG: a knowledge based
data management system.

NYU c.1
Comp. Sci. Dept.
TR-034 Walker

AUTHOR

SYLLOG: a knowledge based

TITLE

data management system.

DATE DUE	BORROWER'S NAME
JAN 14 1986	M V Chitrao

JAN 1 4 1986