Course 35.694

NATURAL LANGUAGE PROCESSING WITH PROLOG

by

Douglas Lyon, BSCSE

A report submitted to
Professor Franklin

April 1985

Electrical Computer and Systems Engineering Department

Rensselaer Polytechnic Institute

Troy, New York, 12180

Phone (518)266-6248

ABSTRACT

A method is described for transforming a minimal English subset into Prolog. The transformation is divided into three stages:

1. Simple limited English into predicate calculus.
2. Predicate calculus into conjunctive normal form.
3. Conjunctive normal form into Prolog.

Sentences which are correctly transformed include:

"Every man loves a woman." and

"John loves a woman."

Most permutations of the above sentences are permitted provided the sentences remain simple. Constraints on the system include context free grammer, noun phrase followed by verb phrase, and zero deviation between semantic and pragmatic meaning.

Applications include creation and consultation of data bases, theorem proving, and expert systems.

Acknowledgment

The research reported here was supported by Professor Randolph Franklin.

Many thanks to Professor David Wieck for verifying the examples on conjunctive normal form and for providing thoughts on the subject of mechanical theorem proving.

This material is based upon work supported by the National Science Foundation under grant no. ECS-8351942, the Rome Air Development Center, contract number, F30602-85-C-0008, subcontract 353-9023-7, and by the Data System Division of the International Business Machines Corp.

Table of Contents

Chapter 1: Introduction

Natural language processing is concerned with theories and techniques that address the problem of natural language communication with computers. Being able to translate English into a computer program is an example of natural language processing. It is a human activity which is attributed to a "programmer". Telling the computer what we mean is the name of the programming game and is a central issue of natural language processing.

The aim of this research is to translate a limited subset of English into Prolog. The English subset has a natural syntax (grammar) and rigorously defined semantics. Examples of properly transformed sentences include "john loves a woman.", "a woman loves john", "every man loves a woman" and so on. The topic is one of machine translation. It is significant to the natural language processing sphere of interest because it ignores the gap between what is intended and what is understood. The study of the gap between what is meant (semantics) and what is understood (pragmatics) is called semiology. By avoiding semiological considerations we can keep the topics considered in an analytic domain.

Consider the class of languages called computer languages (ie. FORTRAN). In a computer language there is no gap between the semantic and pragmatic meaning. Since the

English used by this research has rigorously defined semantics it is like a computer language because there is no gap between the semantic and pragmatic meaning. For example, if the program were to accept the following English statements as rules for its database:
"God is love. Love is blind. Ray Charles is blind."
then it would conclude (perhaps incorrectly) that
Ray Charles is God.

Researchers in the field have thus concluded that a natural language system that understands the semantic and pragmatic meaning of a sentence must "come to an understanding of the user's motivation and the multiple purposes served by any given piece of language." [LEHNERT 82]

Many different approaches have been used in the past to arrive at a workable natural language processing system. Top-down (ie. hypothesis-driven) parsing uses a grammar and tries to fit it to a string [WINOGRAD 83]. A context-free grammer is an example of a top-down parsing scheme where one starts with productions for the initial symbol, and builds an expansion by substitution which will get to the symbols in the string [GAZDAR 81].

The work described in this paper uses a definite-clause-grammar notation [PEREIRA and WARREN 80] to generate Predicate Calculus directly from English [COLMERAUER 82].

> "Predicate Calculus is a formal language which can express statments about limited domains. It comprises a set of symbols, and rules for combining these into terms and formulae." [CHANG and Keisler 37]

See [CROSSLEY et al. 72] for further information on Predicate Calculus. The Predicate Calculus is transformed into conjunctive normal form [CLOCKSIN 81] and then into Prolog.

Chapter 2: English to Predicate Calculus

The conversion of English into predicate calculus is described by functional specification of the main predicates.

Read_in(S) returns a Prolog list (here after referred to as a list) in the variable S. The list is typed in at the terminal in free form by the user. Thus:

```
:- read_in(S).
the quick brown fox.
S = [the, quick, brown, fox]
```

Please note that Prolog responses are underlined.

In the following notation :

a & b means a AND b.

a # b means a OR b.

a -> b means a implies b.

a <-> b means a is equivalent to b.

~a means not a.

all(a,b) means b is true no matter what a stands for.

exists(a,b) means there is something that a can stand for, such that b is true.

The predicate "sentence" is used to transform the list returned by read_in into predicate calculus.

Example 1 - Translation to predicate calculus.

```
/* '***' is the main file */

:- consult(***).

g :- read_in(S),

/* S is the Sentence in list form */

sentence(S,F,Pc),

/* Pc is the predicate calculus */

write(Pc),

nl,

nl,

g.


] ?g.


every man loves a woman.

all(X,man(X) -> exists(Y,woman(Y) & loves(X,Y)))


every woman loves a man.

all(X,woman(X) -> exists(Y,man(Y) & loves(X,Y)))


john loves a woman.

exists(X,woman(X) & loves(john,X))


a woman loves john.

exists(X,woman(X) & loves(X,john))
```

john loves every woman that loves a man.

<u>all(X,woman(X) & exists(Y,man(Y) & loves(X,Y)) -> loves(john,X))</u>

every woman that john loves loves a woman that loves a man.

<u>all(X,woman(X) -> loves(john,X))</u>

The last example will not yield a correct answer because it is in violation of the grammer. A sentence must be a noun phrase followed by a verb phrase. In the sentence "every woman that John loves loves a woman that loves a man." the object of the sentence is "every woman that John loves". A good rule of thumb to avoid this error would be to keep transitive verbs (in this case "loves") from following each other. A more elegant solution is to program the system to recognize the use of a comma in the sentence.

Chapter 3: Predicate Calculus to Conjunctive Normal Form

Conjunctive normal form is a rearrangement of a logical expression so that all the conjunctions (ANDs) appear outside of the disjunctions (ORs). In our notation the symbol '&' is a conjunction and the symbol '#' is a disjunction.

In the example which follows Predicate Calulus formulae are rewritten automatically in conjunctive normal form. A five stage approach is taken in order to achieve the transformation. In stage one we replace a -> b (a implies b) with (~a) # b

and we replace

a <-> b (a is equivalent to b) with (a & b) # (~a & ~b).

In stage two we move negation inward thus,

~(a & b) is changed to (~a) # (~b),

~(a # b) is changed to (~a) & (~b),

~exists(a,p) is changed to all(a,~p),

and

~all(a,p) is changed to exists(a,~p).

Stage three removes existential quantifiers by introducing Skolem constants so

exits(X, female(X) & motherof(X, eve)) is changed to

female(gl) & motherof(gl,eve).

Stage four moves the universal quantifiers outwards. For example:

all(X, m(X) -> all(Y,w(Y) -> p) is changed to

```
   all(X, all(Y, m(X) -> w(Y) -> p)))
```

Finally, stage 5 distributes AND over OR. For example:

```
   (A & B) # C is the same as (A # C) & (B # C)

   A # (B & C) is the same as (A # B) & (A # C)
```

See [CLOCKSIN and MELLISH 81] for further details.

```
] ?listing(g).
g() :-
   read_in(S),
   sentence(S,F,Pc),
   write(the predicate calculus is),nl,
   write(Pc),nl,nl,
   implout(Pc,X1),
   write(with implications removed),nl,
   write(X1),
   nl,nl,
   negin(X1,X2),
   write(with negation moved inward),nl,
   write(X2),nl,nl,
   skolem(X2,X3,[]),
   write(with Skolem constants introduced),nl,
   write(X3),nl,nl,
   univout(X3,X4),
   write(with universal quantifiers moved out),nl,
   write(X4),nl,nl,
   conjn(X4,X5),
   write(with & distributed over #),nl,
   write(X5),nl,nl,
   g.
] ?g.

every man loves a woman.
the predicate calculus is
all(X,man(X) -> exists(Y,woman(Y) & loves(X,Y)))

with implications removed
all(X,~ man(X) # exists(Y,woman(Y) & loves(X,Y)))

with negation moved inward
all(X,~ man(X) # exists(Y,woman(Y) & loves(X,Y)))

with Skolem constants introduced
all(X,~ man(X) # woman(g1(X)) & loves(X,g1(X)))
```

```
with universal quantifiers moved out
~ man(X) # woma[illegible]ves(X,g1(X))

with & distributed over #
(~ man(X) # woman(g1(X))) & ~ man(X) # loves(X,g1(X))

every woman loves a man.
the predicate calculus is
all(X,woman(X) -> exists(Y,man(Y) & loves(X,Y)))

with implications removed
all(X,~ woman(X) # exists(Y,man(Y) & loves(X,Y)))

with negation moved inward
all(X,~ woman(X) # exists(Y,man(Y) & loves(X,Y)))

with Skolem constants introduced
all(X,~ woman(X) # man(g2(X)) & loves(X,g2(X)))

with universal quantifiers moved out
~ woman(X) # man(g2(X)) & loves(X,g2(X))

with & distributed over #
(~ woman(X) # man(g2(X))) & ~ woman(X) # loves(X,g2(X))

john loves a woman.

the predicate calculus is
exists(X,woman(X) & loves(john,X))

with implications removed
exists(X,woman(X) & loves(john,X))

with negation moved inward
exists(X,woman(X) & loves(john,X))

with Skolem constants introduced
woman(g2) & loves(john,g2)

with universal quantifiers moved out
woman(g2) & loves(john,g2)

with & distributed over #
woman(g2) & loves(john,g2)

a woman loves john.

the predicate calculus is
exists(X,woman(X) & loves(X,john))

with implications removed
```

```
exists(X,woman(X) & loves(X,john))

with negation moved inward
exists(X,woman(X) & loves(X,john))

with Skolem constants introduced
woman(g2) & loves(g2,john)

with universal quantifiers moved out
woman(g2) & loves(g2,john)

with & distributed over #
woman(g2) & loves(g2,john)

john loves every woman that loves a man.
the predicate calculus is
all(X,woman(X)  &   exists(Y,man(Y)   &   loves(X,Y))    ->
loves(john,X))

with implications removed
all(X,~ (woman(X)  &  exists(Y,man(Y)  &   loves(X,Y)))   #
loves(john,X))

with negation moved inward
all(X,(~ woman(X) #  all(Y,~  man(Y)  #  ~  loves(X,Y)))  #
loves(john,X))

with Skolem constants introduced
all(X,(~ woman(X) #  all(Y,~  man(Y)  #  ~  loves(X,Y)))  #
loves(john,X))

with universal quantifiers moved out
(~ woman(X) # ~ man(Y) # ~ loves(X,Y)) # loves(john,X)

with & distributed over #
(~ woman(X) # ~ man(Y) # ~ loves(X,Y)) # loves(john,X)
```

Chapter 4: Conjunctive Normal Form to Prolog

Prolog (Programming in logic) is a programming language which uses logical clauses and a resolution theorem prover to make assertions and rules about how a problem may be solved [BOWEN, et al. 82]. If English can be transformed into Prolog properly (and only very limited English subsets can) we should find English specifications of programs to be themselves 'runnable' after only machine translation. It could then be said that if the result is not what was wanted the specification was incomplete or not correct. Naturally this is quite a lofty goal and is quit beyond the scope of this research, it is simply a suggestion of what is possible.

```
:- consult(***).
g :- repeat,
write('>>>'), /* prompt the user */
read_in(S), /* S returns a list which was typed at the
terminal */
sentence(S,F,Pc),/* Pc is the predicate calculus version
of the list S */
write('the predicate calculus is'),nl,write(Pc),nl,nl,
implout(Pc, X1), /* implications removed
negin(X1,X2), /* negation moved inward
skolem(X2,X3,[]),/* Skolem constants introduced
univout(X3,X4), /* universal quantifiers moved out
conjn(X4,X5), /* & distributed over #
write('the     conjunctive      normal        form
is'),nl,write(X5),nl,nl,
clausify(X5,Clauses, []),
write('the clauses are'),nl,pclauses(Clauses),
g.
] ?g.


>>>every man loves a woman.
the predicate calculus is
all(X,man(X) -> exists(Y,woman(Y) & loves(X,Y)))

the conjunctive normal form is
(~ man(X) # woman(g1(X))) & ~ man(X) # loves(X,g1(X))
```

```
the clauses are
woman(g1(X)) :- man(X).
loves(X,g1(X)) :- man(X).


>>>every man loves every woman.
the predicate calculus is
all(X,man(X) -> all(Y,woman(Y) -> loves(X,Y)))

the conjunctive normal form is
~ man(X) # ~ woman(Y) # loves(X,Y)

the clauses are
loves(X,Y) :- man(X), woman(X).


>>>every woman loves a man.
the predicate calculus is
all(X,woman(X) -> exists(Y,man(Y) & loves(X,Y)))

the conjunctive normal form is
(~ woman(X) # man(g2(X))) & ~ woman(X) #  loves(X,g2(X))

the clauses are
man(g2(X)) :- woman(X).
loves(X,g2(X)) :- woman(X).


>>>john loves a woman.
the predicate calculus is
exists(X,woman(X) & loves(john,X))

the conjunctive normal form is
woman(g2) & loves(john,g2)

the clauses are
woman(g2).
loves(john,g2).


>>>a woman loves john.
the predicate calculus is
exists(X,woman(X) & loves(X,john))

the conjunctive normal form is
woman(g2) & loves(g2,john)

the clauses are
woman(g2).
loves(g2,john).
```

Chapter 4: Conjunctive Normal Form to Prolog

Conclusion

The next step in the progression from English to Prolog is to use the Prolog produced by the system described in order to make inferences from transformations of English statements. A step beyond is to recognize uncertainty by modifying the behavior of the inference engine to recognize probabilistic properties of objects. It is conceivable that persons without technical expertise may generate databases for expert systems [HENDRIX 77]. An application of natural language processing used currently [REBOH 79] is the creation of a knowledge acquisition system as a tool to build expert systems [WATERMAN and HAYES-ROTH 82]. In the rule oriented programming environment of Prolog, the rules may be treated as properties of an object. The atoms of an English string may then indicate meta-rules [THOMPSON 82] needed to transform them into the Predicate Calculus. A paradigm may then be organized around recursively composable sets of pattern-action rules (ie. frames [MINSKY 75], conceptual dependencies [SCHANK 75] , knowledge structure, scripts [SCHANK and ABELSON 77]...) where behavior can include the side effect of accessing properties.

More work is required on the subject of transformation in order to the transformation from English to Prolog more robust. The work described in this report only scratches the surface of the English to Prolog conversion problem.

Conclusion

Even when ignoring the semiological consideration, a good amount of "sentence sense" is required to make a general transformation. Considering the difficulty of the English to Prolog transformation, it is evident that an expert system should be used.

Bibliography

[BOBROW and STEFIK]
Bobrow, D.G. and Stefik, M. The Loops Manual, available from David Catton, Artificial Intelligence Ltd., 62-78 Merton Rd., Watford WD1 7BY, England. Tel: 0923-47707

[BOWEN, et al. 82]
Bowen et al. Decsystem-10 PROLOG User's Manual Dept. of Artificial Intelligence, Edinburgh, 1982. Occasional Paper 27.

[BUNDY 84]
Bundy, Alan, Catalogue of Artifical Intelligence Tools, Springer-Verlag, New York, 1984.

[CHANG and KEISLER 37]
Chang, C.C. and Keisler, A.J. Model Theory, North-Holland, 1937.

[CLOCKSIN and MELLISH 81]
Clocksin, W.F. and Mellish, C.S. Programming in Prolog Springer Verlag, 1981.

[COLMERAUER 82]
Colmerauer, Alain "An interesting Subset of Natural Language", Logic Programming, Clark and Tarnlund (editors), Academic Press New York, 1982.

[CROSSLEY et al. 72]
Crossley et al. What is Mathematical Logic?, Oxford University Press, 1972.

[GAZDAR 81]
Gazdar, G. "Phrase Structure Grammar" The Nature of Syntactic Representations, Jacobson and Pullum (editor), Reidal, Dordrecht, 1981.

[HENDRIX 77]
The LIFER Manual TN 138 edition, SRI International, Menlo Park, 1977.

[LEHNERT 82]

Lehnert and Ringle, Strategies For Natural Language Processing, Lawrence Erlbaum Associates, Hillsdale, NJ, 1982.

[LEWIS 80]
Lewis, Anthony The FORTRAN Reference Guide, Prime Computer, Inc. 1980

[MINSKY 75]
Minsky, M. "A framework for representing knowledge." is P.H. Winston (editor), The Psychology of Computer Vision McGraw-Hill, 1975.

[PEREIRA and WARREN 80]
Pereira, F. and Warren, D.H.D. "Definite Clause Grammers for Language Analysis - A survey of the Formalism and a Comparison with Augmented Transition Networks", Artificial Intelligence 13:231-278, 1980.

[REBOH 79]
Reboh, R. "The Knowledge Acquisition System" in Duda, R.O. (editor), A Computer-Based Conssultant foor Mineral Exploration. SRI international, Artificial Intelligence Center, Menlo Park, September, 1979. Final Report, SRI Project 6415.

[SCHANK 75]
Schank, R.C. (editor), Conceptual Information Processing North-Holland, 1975.

[SCHANK AND ABELSON 77]
Schank, R. and Abelson, R., Scripts, Plans, Goals and Understanding Lawrence Erlbaumm Associates, 1977.

[THOMPSON 82]
"Handling Metarules in a Parser for GPSG." In Barlow, M., Flickinger, D. and Sag, I. (editor), Developments in Generalised Phrase Structure Grammer. Bloomington, Indiana University Linguistics Club, 1982. Stanford Working Papers in Grammatical Theory Volume 2.

[WALKER 78]
Walker, Donald E., Understanding Spoken Language,

Elsevier North-Holland, Inc., New York, NY, 1978.

[WATERMAN AND HAYES-ROTH 82]
Waterman, D. and Hayes-Roth, F. An Investigation of Tools for Building Expert Systems. Technical Report R-2818-NSF, Rand Corporation, June, 1982.

[WARREN, PEREIRA and PEREIRA 77]
Warren, D.H.D., Pereira, L.M., and Pereira, F. "Prolog - the language and its implementation compared with Lisp." In ACM Symposium on AI and Programming Languages. Association for Computing Machinery, 1977.

[WINOGRAD 83]
Winograd, T., Language as a Cognitive Process, Addison-Wesley, 1983

Appendix A: Prime ASCII

The ASCII used in the software of this report is specific to a Prime computer. "The Prime internal standard for the parity bit is one", this means that 128, decimal, is added to all ASCII characters [LEWIS 80]. Thus digits 0,1,2, etc are 176, 177, 178, etc in that order. This sort of code dependent thing becomes important in the Gensym predicate (where the number 176 should be changed to 48 for most machines, and in read_in, where numbers are being compared.

```
/* given a word and the character after it read
   in the rest of the sentence */
/* S returns a list which was typed at the terminal *
/
read_in([W|Ws]) :-
     get0(C),
     readword(C,W,C1),
     restsent(W,C1,Ws).

restsent(W,_,[]) :-
     lastword(W),!.
restsent(W,C,[W1|Ws]) :-
     readword(C,W1,C1),
     restsent(W1,C1,Ws).

/* read in a single word, given an initial character
*/
/* and rember what character came after the word */
readword(C,W,C1) :-
     single_character(C),!,
     name(W,[C]),
     get0(C1).
readword(C,W,C2) :-
     in_word(C,NewC),!,
     get0(C1),
     restword(C1,Cs,C2),
     name(W,[NewC|Cs]).
readword(C,W,C2) :-
     get0(C1),
     readword(C1,W,C2).

restword(C,[NewC|Cs],C2) :-
     in_word(C,NewC),!,
     get0(C1),
     restword(C1,Cs,C2).
restword(C,[],C).

/* these characters should form words on there own
 all number should have 128 base ten subtracted from
them
 for transportation to standard ASCII systems */

single_character(172). /* , */
single_character(187). /* ; */
single_character(188). /* : */
single_character(191). /* ? */
single_character(161). /* ! */
single_character(174). /* . */

/* these characters can appear within a word */
in_word(C,C) :-
```

```
     C > 224,
     C < 251.     /* a b ... z */
in_word(C,L) :-
     C > 192,
     C < 219,
     L is C + 32. /* A B ... Z */
in_word(C,C) :-
     C > 175,
     C < 186.     /* 1 2 ... 9 */
in_word(167,167). /* ' */
in_word(173,173). /* - */

lastword(.).
lastword(!).
lastword(?).

/* This prints out a list in a pretty format */
print(D) :-
     pp(D,1).
print(D) :-
     pp(D,1).

/* pp helps to pretty print a list */
pp([H|T],I) :- !,
     J is I + 3,
     pp(H,J),
     ppx(T,J),
     nl.
pp(X,I) :- !,
     tabulate(I),
     write(X),
     write( ).
pp([H|T],I) :- !,
     J is I + 3,
     pp(H,J),
     ppx(T,J),
     nl.
pp(X,I) :- !,
     tabulate(I),
     write(X),
     write( ).

ppx([],_).
ppx([H|T],I) :-
     pp(H,I),
     ppx(T,I).
ppx([],_).
ppx([H|T],I) :-
     pp(H,I),
     ppx(T,I).
```

```
/* Z is the predicate calculus version of the list X
*/
sentence(X,Y,Z) :-
     noun_phrase(X,U,V,X1,Z),
     verb_phrase(U,Y,V,X1).

noun_phrase(X,Y,Z,U,V) :-
     determiner(X,X1,Z,Y1,U,V),
     noun(X1,Z1,Z,U1),
     rel_clause(Z1,Y,Z,U1,Y1).
noun_phrase(X,Y,Z,U,U) :-
     proper_noun(X,Y,Z).

verb_phrase(X,Y,Z,U) :-
     trans_verb(X,V,Z,X1,Y1),!,
     noun_phrase(V,Y,X1,Y1,U).
verb_phrase(X,Y,Z,U) :-
     intrans_verb(X,Y,Z,U).

rel_clause(X,Y,Z,U,U & V) :-
     X = [that|X1],
     verb_phrase(X1,Y,Z,V).
rel_clause(X,Y,Z,U,U) :-
     X = Y.

determiner(X,Y,Z,U,V,all(Z,U -> V)) :-
     X = [every|Y].
determiner(X,Y,Z,U,V,exists(Z,U & V)) :-
     X = [a|Y].

noun(X,Y,Z,man(Z)) :-
     X = [man|Y].
noun(X,Y,Z,woman(Z)) :-
     X = [woman|Y].
noun(X,Y,Z,computer(Z)) :-
     X = [computer|Y].

proper_noun(X,Y,john) :-
     X = [john|Y].
proper_noun(X,Y,kevin) :-
     X = [kevin|Y].

trans_verb(X,Y,Z,U,loves(Z,U)) :-
     X = [loves|Y].
trans_verb(X,Y,Z,U,hates(Z,U)) :-
     X = [hates|Y].
trans_verb(X,Y,Z,U,Z is U) :-
     X = [is|Y].

intrans_verb(X,Y,Z,lives(Z)) :-
     X = [lives|Y].
```

```
/* This permits one list too be subtracted from anoth
er. */
/* subst(a,b,[a,b,c],[a,a,c]) */
subst(V1,V2,F1,F2) :-
     substitute(F1,V1,V2,F2).

/* append allows one list to be appended to another *
/
/* append([], atom, atom) */
/* append([d],[oo],[d,oo])
append([],L,L).
append([X|L1],L2,[X|L3]) :-
     append(L1,L2,L3).
append([],L,L).
append([X|L1],L2,[X|L3]) :-
     append(L1,L2,L3).

/* Create a new atom starting with a root provided an
d */
/* finishing with a unique number */
gensym(Root,Atom) :-
     get_num(Root,Num),
     name(Root,Name1),
     integer_name(Num,Name2),
     append(Name1,Name2,Name),
     name(Atom,Name).

get_num(Root,Num) :-
     /* this root encountered before */
     retract(current_num(Root,Num1)),!,
     Num is Num1 + 1,
     asserta(current_num(Root,Num1)).
     /* first time for the root */
get_num(Root,1) :-
     /* Convert from an integer to a list of characte
rs */
     asserta(current_num(Root,1)).

integer_name(Int,List) :-
     integer_name(Int,[],List).
integer_name(I,Sofar,[C|Sofar]) :-
     I < 10,!,
     C is I + 176.
integer_name(I,Sofar,List) :-
     Tophalf is I / 10,
     Bothalf is I mod 10,
     C is Bothalf + 48,
     integer_name(Tophalf,[C|Sofar],List).

/* concatenate will concatenate to lists */
```

```
concatenate([],L,L).
concatenate([E|R],L2,[E|L]) :-
     concatenate(R,L2,L).

/* reverse([hi,there,mr,potato,head],[head,potato,mr,
there,hi]).
reverse([],[]).
reverse([E|R],L) :-
     reverse(R,R1),
     concatenate(R1,[E],L).

/* get_till_n */
/* Pickup the first n members of a list.*/
/* get_till_n([a,b,c,d,e,f,g],[a,b,c,d],L,4)./*
get_till_n([],[],_,_).
get_till_n([X|L],[X|P],C,N) :-
     (var(C) , C = 1 ; true),
     C =< N,
     C1 is C + 1,
     get_till_n(L,P,C1,N).
get_till_n(_,[],_,_).

unlist([I],I).

list(I,[I]).

cdr([H|T],T).

car([H|T],H).

/* stack */
/* This is a logic program which keeps a list in the
/* for of a stack.
/* push([hi]),push(1),push(a).
/* pop(X),pop(Y),pop(Z), z = [hi], y = 1, x = a
push(X) :-
     asserta((stack(Y) :- Y = X)).

pop(X) :-
     stack(X),
     retract((stack(Y) :- Y = X)).

press(X) :-
     assertz((stack(Y) :- Y = X)).

/* this defines the subtraction relation between two
lists.
   for example: */
/* subtract([a,b,[a,b,c],[a,a,c]). */
subtract(L,[],L) :- !.
subtract([H|T],L,U) :-
```

```
     member(H,L),!,
     subtract(T,L,U).
subtract([H|T],L,[H|U]) :- !,
     subtract(T,L,U).
subtract(_,_,[]).

translate(X) :-
     implout(X,X1),         /* implications removed */
     negin(X1,X2),          /* negation moved inward */
     skolem(X2,X3,[]),      /* Skolem constants introdu
ced */
     univout(X3,X4),        /* universal quantifiers mo
ved out */
     conjn(X4,X5),          /* & distri[illegible] */
     print(clauses passed to clausify are),
     nl,
     write(Clauses),
     nl,
     clausify(X5,Clauses,[]),
     print(clausify returned:),
     print(Clauses),
     nl,
     pclauses(Clauses),
     printclauses(Clauses),
     make_prolog(Clauses).

implout(P <-> Q,(P1 & Q1) # ~ P1 & ~ Q1) :- !,
     implout(P,P1),
     implout(Q,Q1).
implout(P -> Q,~ P1 # Q1) :- !,
     implout(P,P1),
     implout(Q,Q1).
implout(all(X,P),all(X,P1)) :- !,
     implout(P,P1).
implout(exists(X,P),exists(X,P1)) :- !,
     implout(P,P1).
implout(P & Q,P1 & Q1) :- !,
     implout(P,P1),
     implout(Q,Q1).
implout(P # Q,P1 # Q1) :- !,
     implout(P,P1),
     implout(Q,Q1).
implout(~ P,~ P1) :- !,
     implout(P,P1).
implout(P,P).

negin(~ P,P1) :- !,
     neg(P,P1).
negin(all(X,P),all(X,P1)) :- !,
     negin(P,P1).
negin(exists(X,P),exists(X,P1)) :- !,
     negin(P,Pl).
negin(P & Q,Pl & Ql) :- !,
     negin(P,Pl),
     negin(Q,Ql).
negin(P # Q,Pl # Ql) :- !,
     negin(P,Pl),
     negin(Q,Ql).
negin(P,P).

neg(~ P,Pl) :- !,
     negin(P,Pl).
neg(all(X,P),exists(X,Pl)) :- !,
     neg(P,Pl).
neg(exists(X,P),all(X,Pl)) :- !,
     neg(P,Pl).
neg(P & Q,Pl # Ql) :- !,
     neg(P,Pl),
     neg(Q,Ql).
neg(P # Q,Pl & Ql) :- !,
     neg(P,Pl),
     neg(Q,Ql).
neg(P,~ P).

skolem(all(X,P),all(X,Pl),Vars) :- !,
     skolem(P,Pl,[X|Vars]).
skolem(exists(X,P),P2,Vars) :- !,
     gensym(g,F),
     makesk(F,Vars,Sk),
     subst(X,Sk,P,Pl),
     skolem(Pl,P2,Vars).
skolem(P # Q,Pl # Ql,Vars) :- !,
     skolem(P,Pl,Vars),
     skolem(Q,Ql,Vars).
skolem(P & Q,Pl & Ql,Vars) :- !,
     skolem(P,Pl,Vars),
     skolem(Q,Ql,Vars).
skolem(P,P,_).

makesk(F,Vars,Sk) :-
     (Vars = [] , Sk = F ; Sk =.. [F|Vars]).

univout(all(X,P),Pl) :- !,
     univout(P,Pl).
univout(P & Q,Pl & Ql) :- !,
     univout(P,Pl),
     univout(Q,Ql).
univout(P # Q,Pl # Ql) :- !,
     univout(P,Pl),
     univout(Q,Ql).
univout(P,P).
```

```
conjn(P # Q,R) :- !,
     conjn(P,P1),
     conjn(Q,Q1),
     conjn1(P1 # Q1,R).
conjn(P & Q,P1 & Q1) :- !,
     conjn(P,P1),
     conjn(Q,Q1).
conjn(P,P).

conjn1((P & Q) # R,P1 & Q1) :- !,
     conjn(P # Q,P1),
     conjn(Q # R,Q1).
conjn1(P # Q & R,P1 & Q1) :- !,
     conjn(P # Q,P1),
     conjn(P # R,Q1).
conjn1(P,P).

clausify(P & Q,C1,C2) :- !,
     clausify(P,C1,C3),
     clausify(Q,C3,C2).
clausify(P,[cl(A,B)|Cs],Cs) :-
     inclause(P,A,[],B,[]),!.
clausify(_,C,C).

inclause(P # Q,A,A1,B,B1) :- !,
     inclause(P,A2,A1,B2,B1),
     inclause(Q,A,A2,B,B2).
inclause(~ P,A,A,B1,B) :- !,
     notin(P,A),
     putin(P,B,B1).
inclause(P,A1,A,B,B) :-
     notin(P,B),
     putin(P,A,A1).

notin(X,[X|_]) :- !,
     fail.
notin(X,[_|L]) :- !,
     notin(X,L).
notin(X,[]).

putin(X,[],[X]) :- !.
putin(X,[X|L],L) :- !.
putin(X,[Y|L],[Y|L1]) :-
     putin(X,L,L1).

pclauses([]) :- !,
     nl,
     nl.
pclauses([cl(A,B)|Cs]) :-
     pclause(A,B),
     nl,
     pclauses(Cs).

pclause(L,[]) :- !,
     pdisj(L),
     write(.).
pclause([],L) :- !,
     write(:- ),
     pconj(L),
     write(.).
pclause(L1,L2) :-
     pdisj(L1),
     write( :- ),
     pconj(L2),
     write(.).

pdisj([L]) :- !,
     write(L).
pdisj([L|Ls]) :-
     write(L),
     write(; ),
     pdisj(Ls).

pconj([L]) :- !,
     write(L).
pconj([L|Ls]) :-
     write(L),
     write(, ),
     pconj(Ls).

printclauses([]).
printclauses(Clauses) :-
     print(Clauses are :),
     print(Clauses).

make_prolog([]).
make_prolog(Clauses) :-
     reverse(Clauses,Reversed_clauses),
     get_till_n(Reversed_clauses,First_clause,L,1),
     print(the equivalent Prolog program is :),
     nl,!,
     print_clause_1(First_clause),
     cdr(Reversed_clauses,Rest_clauses),!,
     print_rest_clauses(Rest_clauses).

print_clause_1(X) :-
     unlist(X,C),
     arg(1,C,H),
     unlist(H,H1),
     arg(2,C,T),
     unlist(T,T1),
     write(H1,1),
     write( :- ,1),
     write(T1,1).

print_rest_clauses([]) :-
     write(.,1),
     nl(1).
print_rest_clauses([H|T]) :-
     write(,   ,1),
     arg(1,H,H1),
     arg(2,H,H2),
     unlist(H1,H10),
     write(H10,1),
     write( , ,1),
     unlist(H2,H3),
     write(H3,1),
     print_rest_clauses(T).

/* This is the main loop */
go() :-
     repeat,
     write(enter a sentence :),
     read_in(S),
     print(has the same meaning as : ),
     nl,
     sentence(S,F,Pc),
     print(Pc),
     nl,!,
     translate(Pc),!,
     nl,
     nl,
     go.
```

Vita

Douglas Lyon is the Chief Scientist for Raytal Inc., a company involved with laser imaging in real time. He is also Chief Engineer for WRPI a college run 10000 watt radio station. Born in New York City in 1960 Doug has published in Kim one user's note, and assorted hobbiest journels. Doug posseses a B.S. from RPI and is working towards an M.E. in Computer and Systems Engineering.