File No. UIUCDCS-F-86-954

A TRANSLATION SYSTEM *from* CLAUSAL FORM *into* ANNOTATED PREDICATE CALCULUS

Mark S. Goldfain

Department of Computer Science

University of Illinois at Urbana-Champaign

December 1985

ISG 86-2

This research was supported in part by The National Science Foundation under grant: NSF DCF 84-06801, the Office of Naval Research, under grant: N00014-82-K-0186, and the Defense Advanced Research Project Agency under grant: N00014-K-85-0878.

ABSTRACT

This paper discusses a Prolog program which performs translation from *Clausal Form (CF)* into *Annotated Predicate Calculus (APC)* expressions. A complementary system which translates from *APC* into *Clausal Form* also enters into the discussion. A number of theorems are given in the course of a detailed discussion of the translation algorithms. A final section gives some sample test data, from which a general critique of the system's capabilities and performance are drawn. This leads into a discussion of possible further research work.

ACKNOWLEDGEMENTS

Professor Ryszard Michalski first conceived of the CF → APC → CF translator in the spring of 1984, assigning it as a term project. Thanks go first of all to him, both for his helpful insights and explanations and for his patience when we discovered just how much work the CF → APC half involved. Secondly, Igor Mozetic and Professor Nachum Dershowitz were kind enough to read this paper and both gave a number of corrections and helpful suggestions. Tony Nowicki and Larry Rendell also reviewed the paper prior to publication. Of course the responsibility for any errors it may still contain is certainly my own.

Kah - Eng Pua, who wrote an APC → CF translator, was a welcome resource in my implementation of the inverse. The translator was written in the Prolog programming language, so I must thank Professor Claude Sammut at the University of New South Wales, whose elegant interpreter is in use here at the University of Illinois. Thanks are also due to Tony Nowicki for his support of our local version. Finally, many thanks to Valerie, for her accurate assistance entering code and her patience with me during that most trying of periods ... debugging.

SOME NOTES ON NOTATION

This paper discusses several different logical formalisms, or languages. It is hoped that the brief introduction in the first section will suffice for the reader to sort them out. Each of these languages has its own peculiar set of symbols, but at least they are consistent in the usage of symbols they share. The common symbols which appear in almost any present-day logic textbook, such as "&" for logical conjunction, or "∀" for the universal quantifier, are used without much introduction. The special symbols used by the Annotated Predicate Calculus are described in the course of section 2. It is assumed that the reader is familiar with the symbols used in Prolog programs, such as ":-" and "!" . There are nonetheless three symbols which might give the reader pause, and their use is as follows:

→	This is not used as a logical symbol. It is used here simply as an arrow indicating direction.
∇	This is used to denote the exception operator of APC, which is also called the exclusive or (see also page 7).
⊨⊣	This is used to indicate that two statements are logically equivalent. It is used rather than "⇔" when we want to distinguish that we are making a *comparison* of statements in a language, rather than one statement in the language.

CF to APC Translation

1. GENERAL BACKGROUND

The computer programming language *Prolog*, for Programming in logic, is a first attempt at a new style of programming language, commonly called a *declarative* language, as opposed to *procedural* languages. It uses a restricted subset of the *First-Order Predicate Calculus*, known as the *Horn Clauses*, with which the programmer describes various objects and their relationships. A user can then put questions to the Prolog interpreter, (a computer program) once the information has been stored in its database. If it can find an answer to a user's question stored as a fact in its database, it will give that fact as an answer. If it cannot find an answer directly, but has been given enough information to "deduce" the answer, then it will find and report such an answer. This is because of the structure of Horn clauses, which allow for a simple style of logical inference to proceed through a chain of implications.

There are at least two ways to view logic programming, both of which are helpful at times. First, a set of Prolog Horn clauses is often considered a *program* in the sense that it tells the interpreter what action to take in breaking down and solving a complex problem, just as conventional procedural programs tell an interpreter how to step through a problem. (Or, if they are compiled, which is a possibility for Prolog programs as well, then the equivalent machine code tells the machine what to do.) Alternatively, in a declarative interpretation, one can view the Prolog Horn clauses as a set of axioms through which an interpreter (or theorem-prover, or theorem-seeker) scans as it searches for answers to questions. For the present paper, this latter view will probably be more helpful. We will be looking at a superset of the set of Horn clauses, called *clausal forms*, as statements containing information. We will be comparing clausal forms (CF) to First-Order Predicate Calculus (FOPC) and Annotated Predicate Calculus (APC) statements containing the same information.

Robert Kowalski [1] first advocated that logic would be useful for certain classes of programming problems, many of which are current topics in artificial intelligence research. First-Order Predicate Calculus (FOPC) may yet prove best for this, but it has been shown that a simpler set of expressions, clausal forms (CF), have in one sense the same *expressive power* (defined below) as FOPC. The CF language can be viewed as a subset of the language FOPC, but it is written in a different fashion. Because the CF set has important properties for automatic theorem proving, it is of interest in the field of artificial intelligence, and a number of authors have compared the expressive powers of FOPC and CF. A scheme for translation from FOPC to CF can be found in a recent text by Clocksin and Mellish, *Programming in Prolog* [2], chapter 10. To avoid adding unnecessarily to the length of this paper, we will assume the reader is familiar with CF and FOPC as Clocksin and Mellish have presented in their text. It will further be assumed that the reader is familiar with the algorithms they have given for a translation from FOPC into the appropriate CF in pages 208-217.

Expressive power, as used by Clocksin and Mellish, is actually an indication of what class of statements can be made in a language. Consider two languages, $\mathcal{L}$ and $\mathcal{M}$. If for every statement one could make in language $\mathcal{L}$, there were some set of statements in language $\mathcal{M}$ that expressed the same thing, then language $\mathcal{M}$ is equal to or greater than language $\mathcal{L}$ in expressive power. If it also holds that every statement one can make in $\mathcal{M}$ can be made by some set of statements from $\mathcal{L}$, then they are equivalent in expressive power. Clocksin and Mellish are careful to point out that there is a difference in expressive power between CF and FOPC, in that one must use "Skolem constants" in CF to represent the existential quantifiers of FOPC. They admitted that this is not strict equivalence, but felt that it was not critical in this context, since the device of Skolem constants satisfied their needs. This will also suffice for our purposes. Thus we have from their work that it is possible to translate every FOPC statement into a set of CF statements which still retain the logical information. Indeed, if one does any *legal* reordering or renaming of parts in the CF obtained, the information survives and still represents statements that are equivalent to the FOPC expression(s). Not all of the literature agrees on the use of

some of the above terminology, but we are following the definitions given by Clocksin and Mellish.[1]

Another issue besides expressive power should be mentioned. To say that two languages are equivalent in expressive power can be very misleading, since the technical logic term is quite different in meaning than the average English speaker would expect. For example, FOPC is equivalent in expressive power to the language APC, which we shall discuss shortly. Thus CF is also of equivalent expressive power to APC as well. Certainly anything that one can write in APC can be translated into an expression in CF. However, statements which may be succinct in one language may require cumbersome expressions in another. For example, in APC it is easy to express notions of counting. The statement below indicates that there are 25 objects "x" which satisfy a relation "mysterious(x,y)" :

$$\exists(25)x\ \exists y\ [\ \text{mysterious}(x,y)\ ] \qquad (1.1)$$

This has equivalent expressions in FOPC and CF notation, but the shortest FOPC expression the author has seen for this requires about two dozen lines to write (and looks rather strenuous for a human reader to make any sense of). In Clausal Form (CF), the expression grows larger still. In many contexts, *relative conciseness* should prove to be just as important a measure of a language as expressive power. A language which is more concise can generally display higher concepts more clearly than a less concise language, even though they are of equivalent expressive power in the technical sense.

Whereas CF represents a more "primitive" cousin of FOPC, a body of research has looked at ways in which FOPC may be extended, to capture more of the information which natural languages can convey *without loss of rigor*. A key example of this has been given by Michalski et al, in *Machine Learning* [4], chapter 4. That chapter describes an extension called Annotated Predicate Calculus (APC) that can make statements about structured environments in a concise way. We list these extensions here, then give a more detailed description in the next section.

First, APC includes an additional quantifier, which we shall call a "numeric existential quantifier", which allows one to easily make statements of counting. Second, APC introduces compound predicates, which correspond to a contraction that users of natural languages commonly make. Third, APC includes a rich set of operators (see the *exception* operator). Finally, APC introduces the concept of separate annotations, which describe a term and which are meant to be applied wherever the term occurs. This allows a set of constraints on a term to be written just once, without requiring that they be repeated everywhere that one uses the term.

As an example of an annotation, one might define in one place that a "dining set" consists of a table and at least four chairs. Thereafter, wherever the term "dining set" is used, we are constrained by this definition. At present this last concept is not a part of our translation system: we translate numeric existential quantifiers, compound predicates, and exception operators, but have left out annotations. Perhaps a later version of the translator will include these, but we will not discuss them further in this paper.

2. RECENT BACKGROUND

We have already mentioned that one of the goals envisioned for Prolog was that it could be an effective environment to store logical information, which could then be used to deduce further facts using an efficient inference procedure. An automatic translation scheme between APC and CF could potentially allow one to write programs in APC, which could then be translated to Prolog and used. This goal has not been fully realized since Prolog systems in use today allow only Horn clauses, not the full clausal form, and they rely heavily on a control scheme which is order-dependent. This application may become available in concurrent Prolog environments, where the order of clauses is not important.

---

[1] It would be misleading to leave the impression that this is the first such demonstration. The work of Lowenheim, Skolem, Herbrand and Godel in the 1920's and early 1930's laid this groundwork. See their papers in reference [3]. Of course Clocksin and Mellish give an excellent and modern exposition.

There is still plenty of motivation to have such a translator, simply because it can provide an APC "front-end" for users of Prolog systems. Having this, one could inform a Prolog system of whatever one had in the way of *facts* in the powerful APC formalism, that would be rewritten automatically into CF. Next, one could *query* the system in APC. The query would also be rewritten, producing Prolog goals to be tried, from which an answer could be returned. If a theorem-prover made some manipulations or simplifications in the set of clausal forms, then instead of giving its result as a mass of CF, a reverse-translator can look for as concise as possible an APC statement to express the result. Such a statement would likely be a great deal more meaningful to a person using the system than the raw CF output.

Thus, in the spring of 1984 we embarked on a project to translate from APC to CF and vice versa. We chose to limit this goal to only dealing with the simpler kinds of numeric quantifiers and not to deal with annotations, in order to cut down the size of the task. The two parts turned out to be about a factor of 10 different in difficulty. The first part, APC → CF, simply needs to follow a few rules of expansion and convert an APC expression into its FOPC equivalent. Then the translator can simply execute the algorithm described by Clocksin and Mellish [2] and produce its (possibly massive) result at the terminal. This half of the translator was written by Kah Eng Pua, and was accomplished with a Prolog program of some 360 lines [5]. The other direction proved to be about as much harder to achieve as integrating functions is harder than taking derivatives. Indeed, for those familiar with the calculus, those tasks provide a close analogy. The CF → APC translator involves a Prolog program of more than 3600 lines which, while it can recognize a wide variety of forms, still may not succeed in giving the best result.

The goal in constructing the CF → APC translator is to take the specialized clausal forms such as are given in [2] and to see if we can reverse the process, producing the correct higher-level expression from a set of clauses. The task is not as clearly defined in this direction. Usually *many* such translations are possible, each one a different way to say the same thing. Moreover, it is not always clear which of the alternatives is the best one, if any. The CF → APC translator is described beginning in the next section. First we give a short discussion of the problem of translating APC → FOPC. This should explain all of the concepts from APC with which the reader needs to be familiar. Once one has an APC expression converted to FOPC, one can use the algorithm in reference [2] to complete the translation to CF. (Pua gives a similar description in his paper, reference [5]). As mentioned earlier, there are three constructions in APC which we need to expand to form the equivalent FOPC.

The first APC construction I have loosely termed the *numeric existential quantifier*. Instead of only being able to say "There is some X such that ...", as one does in FOPC, one can also say the following in APC:

Concept	APC Form
(1) "There are at least 10 objects X such that ..."	∃ (>=10) X ...
(2) "There are exactly 5 X such that ..."	∃ (5) X ...
(3) "There are at least 3 and at most 7 X such that ..."	∃ (3..7) X ...
(4) "There are 2 or 5 X such that ..."	∃ (2 ∨ 5) X ...
(5) "There are distinct objects X, Y and Z such that ..."	∃.[X,Y,Z] ...

Figure 1 : the Existential Quantifiers in APC

These same concepts can be expressed in FOPC, but only in a more clumsy fashion. The FOPC existential quantifier corresponds to the statement: "There is at least 1 X such that ..."
So in FOPC, one can manage to say

"There are at least 3 X such that <statement(X)>" , **(2.1)**

but only by a more basic construction. One way to achieve this is with:

$$\exists X \exists Y \exists Z \left[\begin{array}{l} (\neg(X = Y) \,\&\, \neg(X = Z) \,\&\, \neg(Y = Z)) \,\& \\ \text{<statement(X)>} \,\&\, \text{<statement(Y)>} \,\&\, \text{<statement(Z)>} \end{array} \right] \qquad \textbf{(2.2)}$$

This unfortunately involves repeating the statement three times (with a different variable each time). An alternate way[2] to express this statement is:

$$\exists X \exists Y \exists Z \left[\begin{array}{l} (\neg(X = Y) \ \& \ \neg(X = Z) \ \& \ \neg(Y = Z)) \ \& \\ \forall U \left[((U = X) \vee (U = Y) \vee (U = Z)) \Rightarrow \langle \text{statement}(U) \rangle \right] \end{array} \right] \qquad (2.3)$$

This more concise form avoids repetition of the statement body, but one still needs a good deal of overhead around the statement, which grows as the square of the number in the numeric quantifier.

To make a statement of the second form in figure 1,

"There are exactly 5 X such that alpha(X)" (2.4)

in FOPC one could either combine two statements like those in (2.2) :

"There are at least 5 X such that alpha(X)"
and (2.5)
"There are not at least 6 X such that alpha(X)"

or, applying the idea from formula (2.3), one could change the implication to an *if and only if* form :

$$\exists V \exists W \exists X \exists Y \exists Z \left[\begin{array}{l} (\neg(V = W) \ \& \ \neg(V = X) \ \& \ \ldots \ \& \ \neg(Y = Z)) \ \& \\ \forall U \left[(U = V) \vee \ldots \vee (U = Z) \right] \Leftrightarrow \langle \text{statement}(U) \rangle \right] \end{array} \right] \qquad (2.6)$$

There are also constructions that represent the 3rd and 4th forms in figure 1, based on the constructions in statements 2.2 to 2.6 above. The fifth form of figure 1 is actually simpler than the others to represent. In FOPC it can be written :

$$\exists X \exists Y \exists Z \left[\neg (X=Y) \ \& \ \neg (X=Z) \ \& \ \neg (Y=Z) \ \& \ \langle \text{statement} \rangle \right] \qquad (2.7)$$

Here, unlike the 2nd through the 4th forms, the "statement" already contains proper reference to all of the variables, so it does not have to be repeated or modified. These statements are all more cumbersome and less clear written in FOPC. They become quite unwieldy when the number in the numerical quantifer is large (large meaning anything above 10 or so).

Indeed, they are so unwieldy that Mr. Pua was led to make an unfortunate choice in developing the APC → CF translator. Instead of producing the correct FOPC and then translating to CF, he decided that a shorter form could be written if a certain Prolog predicate were given a special meaning. He chose to use a functor of the form "number(Var, NumericPart)" which stood for the fact that there were to be a certain number of occurrences of the variable "Var". Thus for example, number(X, 5) would mean that there were 5 instances of X (for further details and the forms used for each type of APC quantifier, see reference [5]) .

Surprisingly, however, some information is lost in this translation. This was not recognized until I began looking at the problem of recovering the APC from the output of the APC → CF translator. Examples demonstrating the loss of information are given in section 4.3, entitled 'simplify'. Since an APC → CF translator such as Mr. Pua has written will likely prove to have more applications than the CF → APC translator which I have produced, it is important that this improper code be re-worked, as mentioned in later sections of this paper (see the discussion of the 'simplify' routine in the section titled "Implementation: The Specifics", and the final section, entitled "Performance: Recommendations Regarding Further Work".)

---

[2] This form was suggested to the author by Nachum Dershowitz.

The second APC construct is the compound predicate, which allows one to make clear statements about objects which share given properties or belong together in some other way. As with numeric quantifiers, every compound predicate has an exact meaning in FOPC, but once again it will take more FOPC statements to convey the same information - the number will depend on how thoroughly compounded the predicate has become. Consider the following three English sentences:

"Bob is tired and hungry."

"Yesterday you talked to Jane or Susan."

"Chris or Mike took the car and drove to Toronto."

In the first, one can view the English sentence as a contraction for the more lengthy: "Bob is tired and Bob is hungry." Any English user will know that this is what was meant. Similarly, an English user will understand that the second sentence corresponds to the statement:

"Yesterday you talked to Jane, or yesterday you talked to Susan."

The idea in both examples is that in English we will often contract an "and" or an "or" construction as far as possible. One cannot talk to something called 'Jane or Susan', but the meaning is nonetheless clear. The third example sentence shows that in English, we feel free to mix these two constructions. Though it takes a bit more thought, we soon recognize that sentence to mean the same as the cumbersome statement:

"Either Chris took the car and Chris drove to Toronto, or
Mike took the car and Mike drove to Toronto."

The general rule seems to be that when both an "and" and an "or" contraction appear together, the "or" goes outside of the "and". One could say that the "and" is more tightly bound to the objects as it appears lower in the parse tree for the sentence's meaning, closer to the nouns involved.

These examples do not show the whole story in English usage, for there seems to be another convention that if more than one connective appears, the first is to distribute around the second. These can conflict, such that the sentence:

"Bill and Jack like Sally or Jane"

is rather ambiguous in its precise meaning.

APC has a construction which reflects the loose conventions above in a rigorous way. Any predicate may have arguments of the forms shown by the following three examples:

predicate(arg1, arg2 & arg3, arg4) .

predicate(arg1 V arg2, arg3) .

predicate(arg1, arg2 V arg3, arg4 & arg5) .

which are interpreted respectively to mean:

predicate(arg1, arg2, arg4) & predicate(arg1, arg3, arg4) .

predicate(arg1, arg3) V predicate(arg2, arg3) .

{ predicate(arg1, arg2, arg4) & predicate(arg1, arg2, arg5) } V
{ predicate(arg1, arg3, arg4) & predicate(arg1, arg3, arg5) } .

Any number of arguments are allowed to be compound arguments, and each compound argument may contain any number of simple terms, so long as "and-compounding" and "or-compounding" are never mixed in the same argument. The last restriction requires that although pred(a & b, c V d) is acceptable, pred(a V b & c, d) is not legal APC.

We can now give the general form for an APC predicate :

predicate (arg_1 , ... , arg_n) ,

where for each i, arg_i is one of the five forms

- simple term (such as "alpha", or "pred(x, y)" ...)
- and-predicate (such as "pred(a & b, c, d)" ...)
- or-predicate (such as "pred(a, b V c, d)" ...)
- simple-or-orterm$_{i_1}$ V ... V simple-or-orterm$_{i_m}$
- simple-or-andterm$_{i_1}$ & ... & simple-or-andterm$_{i_m}$

Here, a "simple-or-orterm" is any term that is either a simple term or an or-predicate, while a "simple-or-andterm" is any term that is either a simple term or an and-predicate. Note that since

pred(f(a) & f(b) V g(c), second, third)

is not a legal form, its contraction to

pred(f(a & b) V g(c), second, third)

is also not legal, even though on the top level it looks fine. That is why we only allow "simple-or-orterms" to be "or-ed" together and only "simple-or-andterms" to be "and-ed" together. Such mixed connector argument compounding is not allowed, as this is deemed more confusing than helpful.

The same forms are allowed in arguments of functions as well as predicates. (Though predicates are in some senses different than functions in logic, they appear identical to the translator at all stages.)

Two non-obvious situations require comment. First, since any number of arguments may be either "or-compounded" or "and-compounded", the rule if there are more than two is that all "and" compounds should be expanded first, then all "or" compounds should be expanded.

Thus, f(p V q, r & s, t & u, v V w) becomes:

f(p V q, r, t, v V w) & f(p V q, r, u, v V w) &
f(p V q, s, t, v V w) & f(p V q, s, u, v V w)

which then gives:

{ f(p, r, t, v) & f(p, r, u, v) & f(p, s, t, v) & f(p, s, u, v) } V
{ f(p, r, t, w) & f(p, r, u, w) & f(p, s, t, w) & f(p, s, u, w) } V
{ f(q, r, t, v) & f(q, r, u, v) & f(q, s, t, v) & f(q, s, u, v) } V
{ f(q, r, t, w) & f(q, r, u, w) & f(q, s, t, w) & f(q, s, u, w) }

The second complication involves a bit of recursion which can develop in this situation. There is nothing to prevent the following APC clause from being used:

f(p, g(q V r, s), t & u)

This one is expanded by recognizing first that g(q V r, s) means g(q, s) V g(r, s), so the whole thing expands to:

[f(p, g(q,s), t) & f(p, g(q,s), u)] V [f(p, g(r,s), t) & f(p, g(r,s), u)]

The CF → APC translator was designed to handle these two situations correctly (though I am not going to *guarantee* that the code is bug-free!)

The third and final APC feature is a rich set of connectives. The APC connectives are shown in figure 2.

Connector	Meaning	Comments
&	and	
∨	or	
¬	not	
⇨	only if	also read "implies"
⇦	if	also read "is implied by"
⇔	if and only if	
∇	except if	also read "exclusive or"

Figure 2 : The Logical Connectives of APC

The reader will recall that A ⇨ B is logically equivalent to ¬A ∨ B , thus A ⇦ B is equivalent to A ∨ ¬B , and that various equivalents for A ⇔ B include (A ⇨ B) & (B ⇨ A) , and therefore (¬A ∨ B) & (¬B ∨ A) , as well as (A & B) ∨ (¬A & ¬B) . This last form is perhaps closest to the way one generally thinks of it.

The APC "exception" operator is not really new. It is more commonly called the "exclusive or", and is generally discussed at some point in basic texts on standard logic. It is given the following interpretation: A ∇ B means A ⇔ ¬B , which is (A ⇨ ¬B) & (¬B ⇨ A) , so it also equates with (¬A ∨ ¬B) & (B ∨ A) , or (A & ¬B) ∨ (B & ¬A) . This last form is helpful toward understanding its name, since it shows that the condition A holds, except when B does.

3. IMPLEMENTATION : OVERVIEW

To get a general idea of the task for a CF → APC translator, take the example output which Clocksin and Mellish give on page 217:

```
person(fl(X)); king(X) :- .
king(X) :- respects(fl(X), X) .
```

We might be given this as input, and asked to find out what it "means" in FOPC or APC. To determine this, we need to reverse the transformations that were made on it. One needs to replace the universal quantifiers which were removed, which in this case means putting a universal quantifier in for "X". One also needs to return any Skolem functions into existentially quantified variables. In order to do this, one needs to be able to recognize that some functions are designated Skolem functions and play this special role. Such a convention is used in the translator, as described later. This whole process takes a clause (or set of clauses) which we may say is *implicitly quantified*, and making the quantification explicit. If the translator is recovering an APC expression from some clauses which have been manipulated, we could refer to this as *re-quantification*. In this paper, for simplicity, we will simply refer to this as *quantification*. After quantification, the process continues - we look patterns that indicate we could make an expression into an implication, or for ways to collect negations higher in the expression, or to form an *if-and-only-if* expression from two simple implications. Also, since we are translating into APC, rather than just FOPC, we would also try to recognize patterns for numeric existential quantifiers or exclusive or's, etc.

There are three versions of the main routine of the CF → APC translator, providing for options the user may or may not want. The differences in the three are not essential, but cosmetic. In general, the main clause is as follows:

```
prologtoapc(InFile, OutFile) :-
        readinlist(InFile, ClauseList),
        quantify(ClauseList, Quantified),
        simplify(Quantified, [APCClause]),
        optimize(APCClause, BestForm),
        tell(OutFile), writeout(BestForm), told,
        writeout(BestForm), !.
```

As the main clause of the program shows, the program has 9 goals to accomplish to perform the translation (counting "!" as a goal). Some of these goals are not very significant, while others are groupings of a great deal of activity. Let us discard the insignificant ones immediately. just below "optimize", the job is really done. The last five goals :

```
        tell(OutFile),
        writeout(BestForm),
        told,
        writeout(BestForm), !.
```

merely open the output file, write out the structure obtained using a routine called 'writeout', close the file, then print the result also at the terminal, and finally "cut", to stop the program. The "writeout" routine is a straightforward recursive pretty-printer for APC expressions. It comprises only about one page of code, and we will not dwell on it further. The heart of the program is contained in the first four steps, which are detailed in the next section.

4. IMPLEMENTATION : THE SPECIFICS

4.1. 'readinlist'

Of the remaining four goals, 'readinlist' was the simplest. It uses a built-in procedure 'read' to get the next clause, and loads all of the clauses from an input file. It should be noted that because the CF counterpart of a typical APC expression can typically be very lengthy, an input file is the only reasonable way of giving the clauses to the translator. This produced some further difficulties that were not present in the APC → CF translator. (For example, the 'read' input routine turns all of the variables it encounters into literal strings, which proved a slight inconvenience in the duration of the program.)

Once the clauses are read in, they are put into a list structure which represents the "and" of all of the clauses. The head and tail of each clause are turned into a disjunction, so one sometimes has a conjunctive-normal form at this stage. The details of the list structure used by the program to internally represent logic statements are not of interest to us here. The reader who wishes to see the actual form used is referred to Appendix A. What is relevant to note is that an input of n CF statements will produce a list of n disjunctions, which are implicitly meant to be "anded" together when it becomes possible to join them. As an example, suppose our input consisted of the five clauses shown in figure 3a, on the following page. These would be converted into the list shown in figure 3b. (The numbers at the left are for reference only, and do not appear in either the input or the resulting list.)

(1) stable(X) :- onground(X), solid(X) .

(2) solid(brick) .

(3) faculty(A) ; staff(A) :- not(student(A)) .

(4) faculty(B), staff(B) :- dean(B) .

(5) (terrible(X, Y) ; wonderful(X, alpha, Z)) , beautiful(Z) :-
more(X, zero) ; (less(X, freezing([ice, cream]), cones),
not(simple(Y, Z))) .

(a) an input file with five Prolog clauses

(1) stable(X) V (¬ onground(X) V ¬ solid(X)) ,

(2) solid(brick) ,

(3) (faculty(A) V staff(A)) V student(A) ,

(4) (faculty(B) & staff(B)) V ¬ dean(B) ,

(5) ({ terrible(X,Y) V wonderful(X,alpha,Z) } & beautiful(Z)) V
(¬ more(X,zero) & {¬less(X,freezing([ice, cream]),cones) V simple(Y,Z)})

(b) the output of the 'readinlist' routine

Figure 3 : An example of the 'readinlist' routine in action

In the event that it is not clear how these are arrived at, let's briefly review the way clauses are interpreted logically. (See especially pages 214-215 of reference [2].) Any clause will consist of one or more terms, separated by commas, semicolons, or the symbol ":-", and ending with a period. Thus the first clause in our input consisted of the three terms: stable(X), onground(X), and solid(X). The separators are actually meant as logical connectors: the comma means "logical and", the semicolon means "logical or", and the ":-" symbol means "if", or if you prefer, "is implied by". Thus, the first clause can be read as: " 'stable(X)' is true if 'onground(X)' and 'solid(X)' are both true." It is not difficult to rewrite such a clause. Instead of making it an implication, we use the logical identity:

$$A \Leftarrow B1 \,\&\, B2 \,\&\, \ldots \,\&\, Bn \quad \models\!\dashv \quad A \vee \neg B1 \vee \neg B2 \vee \ldots \vee \neg Bn$$

As the exact inverse of the last step in APC or FOPC → CF translation. We prefer this to other forms since it contains only the connectives "and" and "or", joining simple terms or their negations. This type of logical statement is fairly easy to manipulate through the first stages of our translation.

Thus the first clause produces the logical statement: stable(X) V ¬(onground(X)) V ¬(solid(X)) . The second clause was simply one fact, so 'readinlist' does not need to do anything with it beyond placing it in the list. The third clause has two terms before the ":-" which produce an "or" inside the main disjunction. The last two clauses are less commonly seen. Clause four was

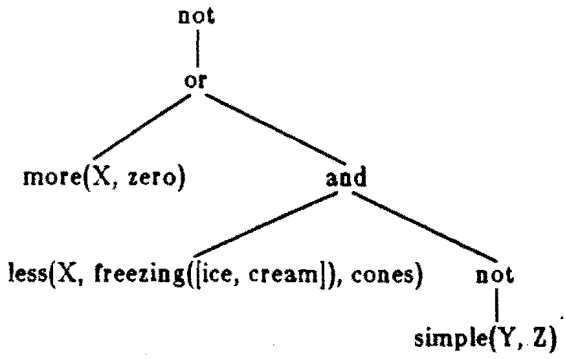
faculty(B), staff(B) :- dean(B) .

and produces a conjunction within a disjunction: (faculty(B) & staff(B)) V ¬ dean(B) .

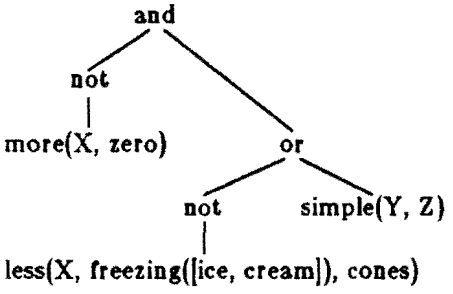
Finally, clause 5 is a fair mess. Its head is an "and" of two parts, the first of which is itself an "or" of two simple terms. Its tail is of an opposite construction, being an "or" of two parts, the second of which is an "and" of two terms:

```
(terrible(X, Y) ; wonderful(X, alpha, Z)) , beautiful(Z) :-
more(X, zero) ; (less(X, freezing([ice, cream]), cones),
                    not(simple(Y, Z))  ) .
```

The result produced shows these structures remaining within the overall disjunct which replaces the ":-" symbol. Note that here a bit of logical manipulation has taken place. The second half of our output might have had the structure:



But we do not wish to have any negations high in the tree. Thus we distribute negations down to the literals to produce:



Which finally resulted in our form:

({ terrible(X,Y) ∨ wonderful(X,alpha,Z) } & beautiful(Z)) ∨

(¬ more(X,zero) & {¬less(X,freezing([ice, cream]),cones) ∨ simple(Y,Z)}) .

Note that the first three clauses in the example all resulted in logical expressions containing only "or" and "not" connectors. If only these types of input are present, we get a conjunctive normal form from 'readinlist'. Although Clocksin and Mellish's algorithm always produces a conjunctive normal form, and clauses that are strictly disjunctions, it could not be assumed that our input would always satisfy that form. As the example illustrates, it is perfectly legal CF to use conjunction in the head and disjunction in the body of a clause. The standard input routine 'read' had no special problem with these forms, and they did not present a special difficulty down the line, so I allowed these to be handled as well. Thus, one may not

always get a conjunctive normal form at this stage, but 'readinlist' can produce the proper structure for any set of CF clauses that are legal.[3]

4.2. 'quantify'

All legal CF expressions are valid through the first step, which is fairly simple, only occupying about 2% of the program code. Now things begin to get interesting. Once one has the rudimentary logical form, one needs to quantify it. That is, variables need to be given universal quantifiers, while Skolem constants and Skolem functions need to be replaced with variables that have existential quantifiers. This needs to be done before many logical manipulations can be tried, since certain seemingly harmless actions, like distributing the negations to more appropriate places, will alter the sense of quantifiers and invalidate the result.

Note that a special predicate needs to be reserved for Skolem functions in the translator. I chose to reserve the letters "skf" followed by an integer as the set of unique identifiers to stand for Skolem constants or functions. Thus, "skf5" might be a Skolem constant, and "skf3(X, Y)" a Skolem function which depended on X and Y. Both Skolem constants and functions may be treated uniformly by considering Skolem constants to be "functions of zero arguments". We will henceforth generally only mention Skolem functions, letting it be understood that constants are included as well.

When we attempt to quantify our clause list, there are many sets of clauses which, although they are legal CF expressions, do not correspond to any well-formed FOPC expressions. For a frivolous example, suppose the list consisted of only one clause:

skf1(X) = skf1(Y) .

This is not a valid way of naming Skolem functions - a new function needs to be used for every new object, rather than re-using skf1. As another requirement, we insist the argument lists always be given in the same order. Thus the clause:

funnyresult(X) :- skf3(X, Y) = 0, skf3(Y, X) = 0.

is not properly formed. This may seem an unnecessary requirement, since at first it seems that only the *set of arguments*, not their *order* seems critical to the formation of Skolem functions. The legitimate need for this requirement should become clear by the end of this subsection.

In any case, we have seen that there are clauses for which no legal quantification is possible. Ideally, we would like an algorithm that could be sure of producing a correct quantification for the clause list if one existed, and which could determine that no legal quantification was possible, if that were the case. As we will see shortly, this is not likely to be found. We begin with a couple of definitions and remarks, then we will give a few theorems which discuss the question of *quantifiability*.

Definition. In Prolog, the scope of a variable consists of the clause in which it occurs, while the scope of an atom or functor is the entire set of clauses in which it occurs.

The meaning of this is made clear by the following illustration. Suppose that in a large Prolog program (or database), we had a thousand clauses. Suppose the 30th and 31st clauses were:

alphabet(X, Y) :- romancelang(X), Y = latinalphabet.
alphabet(A, X) :- grecianlang(A), X = greekalphabet.

Also, suppose the 930th clause was:

romancelang(english) .

---

[3] Note: in the Prolog interpreter in use here (UNSW Prolog), one type of standard clause was not able to be properly read by the 'read' predicate; those with a neck but no body, such as " alwaystrue(Clause) :- . " turn out to be considered illegal in UNSW Prolog. This is only a minor difference in style however, since the UNSW interpreter *does* accept the shortened form, " alwaystrue(Clause). " which means exactly the same thing.

In such clauses, the convention in Prolog is that names beginning with upper case are variables, while those beginning with lower case are atoms or the names of functors. (Perplexingly chosen to be the opposite of the usual convention in English.) Thus, 'X', 'Y', and 'A' are variables, 'latinalphabet', 'greekalphabet', and 'english' are atoms, meant to name objects, and 'alphabet', 'romancelang', and 'grecianlang' are functor names, meant to designate functions, which are also objects in some sense.

In the 30th clause, the variable X occurs twice. Both places that it occurs it is meant to refer to the same thing. We are using this same convention in our interpretation of clauses - all occurrences of a given variable in one clause must be unified together - they must refer to the same thing.[4] However, the variable X occurs again in the very next clause (the 31st). Although the two clauses are close together and have the same overall structure, we clearly do not require the use of X in clause 31 to be unified with the use of X in clause 30. Indeed, the names being used would indicate that we are talking about languages and alphabets, so in clause 30, X is probably used to designate a language and in clause 31 an alphabet.

Thus, if a *variable* occurs in different clauses, we cannot make any statement as to whether it needs to refer to the same thing in one clause as in another. The same variable is free to be used for different things in different clauses (and as is virtually always true of variables, different variables are quite free to be used for the same thing, either in different clauses or the same clause.) Meanwhile, the *function* 'alphabet' occurs in two clauses, and we see the function 'romancelang' occurring in two widely separated clauses. These *are* meant to refer to the same function wherever they occur. Similarly, wherever the *constant* 'latinalphabet' occurs in the database, it is always intended to refer to the same object.

Definition. Let **C** be a clause and let **S** be the set of Skolem functions (and constants) occurring in the clause. For example, if **C** is the clause:

$$\text{skf3(X)} = \text{f(X, V)} \; ; \; \text{skf4(X, Y)} = \text{X} \; :- \; \text{f(skf3(X), X)} = \text{g(V)}, \; \text{Y} < \text{X}, \; \text{skf5(X, Z)} = \text{Z}.$$

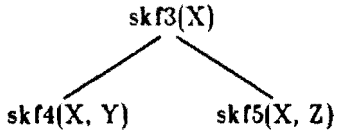
Then **S** = { skf3(X), skf4(X, Y), skf5(X, Z) } . Put the elements of **S** into a directed-graph structure as follows: For each Skolem function, consider the set of variables in its argument list. Partition **S** into subsets containing Skolem functions with identical argument lists. Let each such set be a node in the graph. Thus, each node **n** has associated with it an argument list, which we shall term *the a-list for* **n**. Now add edges to the graph which show the subset ordering on the a-lists. To be specific, edges are added according to the following rule: If the a-list for node **p** is a proper subset of the a-list for node **c**, and if no node **m** exists such that the a-list for **m** is a proper subset of the a-list for **c** and a proper superset of the a-list for **p**, then an edge is drawn from **p** to **c**. The structure so obtained is called the *skf-forest* for the clause **C**.

Remarks : We do not draw an edge from a node to itself. Hence the use of the word "proper" in the definition. We will use the terms "descendent" and "ancestor" to denote nodes for which there is a path of edges from the ancestor node to the descendent node, reserving the terms "father" and "child" for the special case where the path is a single edge. Note that it is not possible for a node to be its own descendent (no cycles), nor is it possible for a node to be a father to both a node and any of its descendents. It *is* possible for a node to have more than one father, so the term *forest* is a bit misleading, as the following examples show (technically, the second example is not a forest) but we will justify our use of the term later.

---

[4] Prolog has a special variable which need not unify with itself, the so-called *anonymous variable*. Anonymous variables are not used here since they have no meaning in this context. If they are given in the input, the translator will complain.

Example 1. From the set S mentioned above, the skf-forest would be:

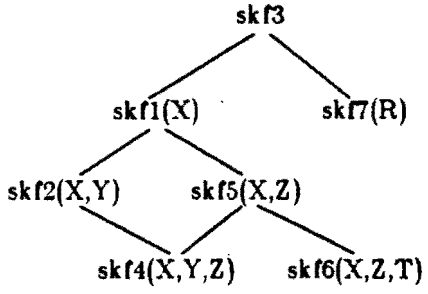


Note that the skf-forest for a given clause depends only on the set of Skolem functions found in the clause, not on any other structure information of the clause, nor on any of the variables in the clause which do not happen to occur in the argument lists of any of the Skolem functions.

Example 2. As a more complicated situation, if for some clause:

$$S = \{ skf1(X), skf2(X, Y), skf3, skf4(X, Y, Z), skf5(X, Z), skf6(X, Z, T), skf7(R) \}$$

then the skf-forest for this clause is:



We can begin with a simple theorem regarding quantifiability.

Lemma. Any clause containing *no* Skolem functions is quantifiable.

Proof: Any clause with only universal quantifiers is logically equivalent to the same clause with all of the quantifiers pulled to the outside. Thus we may quantify the clause simply by surrounding it with the sequence:

$$\forall X_1 (\ldots \forall X_n (\ \ldots \ \langle clause \rangle \ \ldots \ ) \ldots)$$

Where $X_1 , \ldots , X_n$ are the variables found in the clause. This will ensure that every variable in the clause is properly quantified. □

Theorem 1. Any clause for which all Skolem functions are constants is quantifiable.

Proof: In the special case that there are no Skolem functions, the lemma shows a valid quantification. If a list of Skolem constants is added to the agenda, then our only requirement is that none of the quantifiers for the Skolem constants fall inside of the (universal) quantifiers for the variables which occur in the clause. Thus, if $V = \{ X_1, \ldots ,X_n \}$ is the set of variables found, while $S = \{ skf1, \ldots , skfm \}$, then by giving these functions variable names of $Y_1, \ldots , Y_m$, respectively, we can quantify the clause with the scheme:

$$\exists Y_1 (\ldots \exists Y_m (\forall X_1 (\ldots \forall X_n (\ \ldots \ \langle clause \rangle \ \ldots \ ) \ldots)) \ldots) .$$

Again, this is a valid quantification for the clause. ■

Now let's look at some clauses that *cannot* be quantified. As a simple and stubborn example, consider the clause:

$$X = skf1(X) \ \vee \ function(skf2(Y), skf3(Z)) . \qquad \textbf{clause 1}$$

Imagine an attempt to quantify this clause. It would require a logical expression containing somewhere the indivisible part:

$$function(skf2(Y), skf3(Z))$$

No matter what logic we perform, this cannot be split up, because it is a literal. But this must be a part of an expression which has existential quantifiers surrounding it, one for skf2(Y) and one for skf3(Z). Suppose that we replace these two Skolem functions with A and B, respectively, so we have function(A,B). Somehow we must put an existential quantifier in for both. Thus, somewhere surrounding this term, we have the quantifier for A:

$$\exists A (\ \ldots \ function(A,B) \ \ldots \ )$$

Now A came from the function skf2(Y), so we know that outside of this we have the quantifier for Y. Our expression must then look like:

$$\forall Y [\ \ldots \ \exists A (\ \ldots \ function(A,B) \ \ldots \ ) \ \ldots \ ]$$

Next, we know that $\exists$ B must surround the term "function(A,B)". Suppose that it occurred somewhere within the expression we have built so far. If that were the case, then B could not have been given the Skolem function skf3(Z), because the variable Y was surrounding it. The absence of Y from the argument list of skf3(Z) forces us to conclude that the quantifier for B lies outside of the scope of the quantifier for Y. Thus we now have:

$$\exists B \{ \ \ldots \ \forall Y [\ \ldots \ \exists A (\ \ldots \ function(A,B) \ \ldots \ ) \ \ldots \ ] \ \ldots \ \}$$

But now we are stuck; the quantifier $\forall$ Z needs to surround the quantifier for B, but it may not surround the quantifier for A, once again because it is not listed in the argument list for skf2(Y) . The problem is that in both FOPC and APC all expressions must be "properly nested", so that no quantifiers can have scopes that only partially overlap - one must lie inside of the other. □

On the other hand, there are clauses which are not quantifiable without transformation, but are quantifiable if one can find the required trick. Consider a second example:

$$([X = skf1(X)] \ \& \ [Z > skf2(Z)]) \ \vee \ ([X = skf1(X)] \ \& \ [Z < skf2(Z)]) . \qquad \textbf{clause 2}$$

As it stands now, all four quantifiers required (one for X, Z, skf1(X), and skf2(Z)) need to be placed outside the entire original expression, since all four objects occur in both halfs of the clause. But no matter what order is chosen for the four quantifiers, either skf1(X) is quantified within the scope of the quantifier for Z, or skf2(Z) within the scope of the quantifier for X. All is *not* lost however, for the above expression is of the form (A & B) $\vee$ (A & C), which is logically equivalent to A & (B $\vee$ C) . Thus the clause can be transformed to:

$$[X = skf1(X)] \ \& \ [(Z > skf2(Z)) \vee (Z < skf2(Z))] , \quad \text{which is easily quantified as:}$$
$$\forall X \exists A [X = A] \ \& \ \forall Z \exists B [(Z > B) \vee (Z < B)] .$$

Definition. If for some clause quantification is possible without rearranging any of its parts, just simply inserting quantifiers at strategic locations, then we may call such a clause *quantifiable without transformation.* Of course any clause which is quantifiable at all must have some logically equivalent clause which is quantifiable without transformation. If at least this is possible (as it was in clause 2) then the clause may be termed *quantifiable with transformation.*

Theorem 2. The examples just given indicate a theorem that a clause is not quantifiable if it has an indivisible part, one that is not removable or separable by any transformation, and which contains two Skolem functions, for which neither one has an argument list that is a subset of the other. ■

To summarize what we have seen thus far, we've seen some clauses which are quantifiable immediately, some which are simple enough that we can prove they are not quantifiable, and others, often rather simple in appearance, which are quantifiable after some logical transformation, but for which it may be very difficult to find a quantification. (The reader may have felt the last example was "obviously quantifiable", but examples which are disguised to any degree of difficulty may be contrived.) Indeed, this shows the difficulty in finding any criterion that declares a general clause as non-quantifiable. To prove a clause non-quantifiable, one must prove that all of its logically equivalent forms are non-quantifiable.[5]

As we consider less trivial situations, the goal is to find a quantification which fits the constraints of the arguments in the Skolem functions. This is where the skf-forest will have its utility. The edges in the skf-forest show which quantifiers need to belong inside which others:

Theorem 3. If the clause **C** is quantifiable without transformation, then it can be arranged such that for every node in the skf-forest, the quantifiers for the Skolem functions in that node contain the quantifiers for all of their descendents within their scope.

That is to say, if the Skolem function skf1(...) is an ancestor to the Skolem function skf2(...), then we can quantify the clause such that " ∃ skf1" will contain " ∃ skf2" in its scope.

Proof : If the Skolem function skf1(...) is an ancestor to the Skolem function skf2(...), then we know that the arguments for skf1(...) are a proper subset of the arguments for skf2(...) so we may label these as skf1(V_1,...,V_n) and skf2(V_1,...,V_n,V_{n+1},...,V_{n+m}). If we give these functions variable names of S and T, respectively, clearly the quantifier ∃S cannot fall within the scope of ∃T, since ∃T is itself within the scope of the quantifiers $\forall V_{n+1}$ through $\forall V_{n+m}$, and ∃S is not. Thus we either have ∃T within the scope of ∃S, or they have scopes which are disjoint. In the first case we have satisfied the claim, while in the second case, we would have an FOPC expression which contained the quantifiers $\forall V_1,...,\forall V_n$. Supposing for discussion that $\forall V_n$ were the innermost, then the relevant part of our expression would look something like:

$$\forall V_n(\text{ - - - [part containing } \exists S \text{] } * \alpha * \{ \text{ part containing } \forall V_{n+1}...\forall V_{n+m} \text{ and } \exists T \} \text{ - - - })$$

Where the use of "*" indicates unknown connectors (either ∨ or &), while the use of "- - -" indicates some irrelevant "stuff". The above may be a valid quantification for the clause, but it is not the only possibility. Consider the form of the "[part containing ∃S]". It is an FOPC expression, which may contain the ∃S at its top, or that may be embedded in any number of &'s, ∨'s, ¬'s, or other existential quantifiers. There can be no universal quantifiers outside of the ∃S, and if any existential quantifiers are outside of ∃S, we know they also came from the same node as ∃S (if they did not, there would have to have been either some additional (universal) variables in their argument lists or some lacking from their lists - which is not possible.) Further, we know that S does not occur in any of the parts of the expression outside of its quantifier, since all variables are consistently named in the clause, thus we may make use of the identities

---

[5] This may or may not be realizable. Perhaps a canonical form exists for which a criterion might be found which characterizes those forms which have quantifications versus those which do not. This is at least a challenging topic for further theoretical exploration. The problem is perhaps analogous to topics which have already been thoroughly developed in the study of formal languages, circuit minimization, etc. The author freely admits his paucity of experience in these areas.

which hold for FOPC expressions:

$\exists S(\alpha)\ \&\ \beta \models\!\dashv\ \exists S(\alpha\ \&\ \beta)$, if S does not occur free in β,
$\exists S(\alpha)\ \vee\ \beta \models\!\dashv\ \exists S(\alpha\ \vee\ \beta)$, if S " " " " " ", and
$\exists R(\exists S(\alpha)) \models\!\dashv\ \exists S(\exists R(\alpha))$

to move $\exists S$ higher and higher in its subclause until it was the topmost item. Then we would have:

$$\forall V_n(\text{- - -}\exists S[\ \beta\ ]\ *\ \alpha\ *\ \{\text{ part containing } \forall V_{n+1}...\forall V_{n+m} \text{ and } \exists T\ \}\text{- - -})$$

and finally, we may use these same identities to "stretch the "[ ]" portion to include α and the portion in the "{ }". Thereby we arrive at the logically equivalent expression:

$$\forall V_n(\text{- - -}\exists S[\ \beta\ *\ \alpha\ *\ \{\text{ part containing } \forall V_{n+1}...\forall V_{n+m} \text{ and } \exists T\ \}\ ]\text{- - -})$$

This second form provides a valid quantification for the clause (without transformation) and it has $\exists T$ within the scope of $\exists S$. ■

Theorem 3 can be slightly strengthened. Instead of merely covering clauses which can be quantified without *any* transformation, the above proof applies to any clause which may be quantified without any transformation *which would alter the skf-forest.*

Definition. An skf-forest is *properly branched* if and only if none of its nodes has more than one father node. Any skf-forest which fails to meet this criterion is said to be *improperly branched.*

Theorem 4. If the skf-forest for a clause is improperly branched, then the clause cannot be quantified, unless some transformation is possible which eliminates the improper branching in the skf-forest (this could only be done if a part of the clause were redundant, so that some logical transformation might eliminate that portion of the clause altogether, and thereby eliminate some Skolem functions from the clause.)

Proof: We have at least one node in the skf-forest which has at least two father nodes. We may choose a Skolem function from each of the three nodes. Let C be the Skolem function from the child node, and let F and G be the Skolem functions chosen from different father nodes. Now neither of the nodes for F or G may be an ancestor of the other, by construction. This implies that both F and G have nonempty argument lists, neither of which is a subset of the other. Also, C must have an argument list which has both of these as subsets. Thus we may choose a variable X from the argument list for F which is not in the argument list for G. Likewise, we may choose a variable Y which is not in the argument list for F. Both X and Y will appear in the argument list for C.

Now if it were possible to quantify this clause, we would get an FOPC expression containing $\exists C$, $\forall X$, and $\forall Y$. The quantifier $\exists C$ would need to lie within the scope of both quantifiers $\forall X$ and $\forall Y$, so they may not have disjoint scopes. But if their scopes overlap, one must contain the other. It is not possible for $\forall X$ to contain $\forall Y$ within its scope, since $\exists G$ lies in the scope of $\forall Y$, but not in the scope of $\forall X$. But it is also not possible for $\forall Y$ to contain $\forall X$ within its scope (consider $\exists F$.) We have a contradiction which is becoming familiar: one quantifier must encompass the other, or their scopes must be disjoint. ■

This last theorem justifies our use of the term skf-forest. If the clause has an skf-forest which is not really a forest, then we need not even try to quantify it unless we can simplify it first.

Definition. Let **C** be a clause containing a set of Skolem functions, **S**, and a set of variables, **V**. The *complete skf-forest* for the clause, denoted $\mathcal{F}$, is the set of tree structures formed from the skf-forest by the following additions and adjustments:

(1) For each variable X in **V** that occurs in the argument list of some member of **S**, we place that variable in a new node in the forest which is a father node to all Skolem function nodes which have X in the argument lists of the Skolem functions, but which have no ancestors which have X in their argument lists. All edges which previously ran into these nodes are now directed into the node containing X.

For example, if skf1(X) was a father to skf2(X,Y,Z) before variables are added, then {Y, Z} is placed in a node between these two. {X} will become the father for skf1(X). If the clause contained a Skolem constant node, then skf1(X) would have been its child before addition of variables, and afterward {X} will appear between them. Otherwise {X} will become a fatherless node.

(2) The above rule defines where each variable in **V** belongs in $\mathcal{F}$ except for those variables which do not appear in the argument list of any Skolem functions. If no Skolem constants occur in **S**, these variables are placed in a node by themselves, which becomes a one-node tree on its own. If Skolem constants are present in **S**, then the variables are placed in a branch immediately below the Skolem constants in the complete skf-forest, as a separate branch to others that spring from that node.

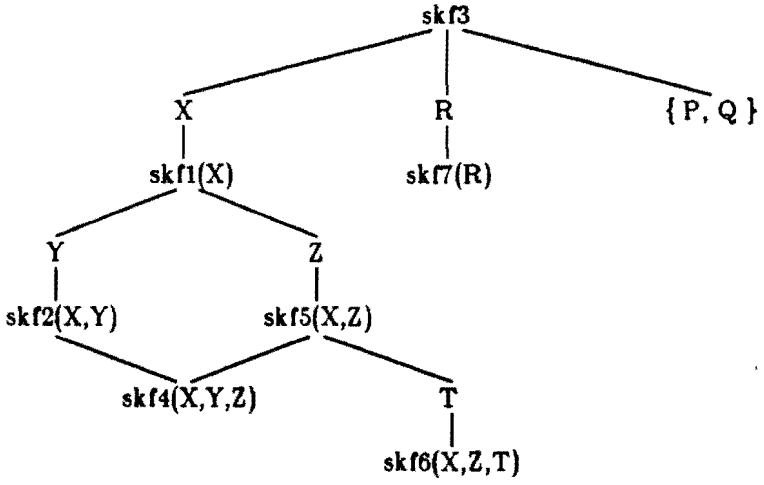
(3) Once this is accomplished, we take one further step: There may be separate variable nodes which both have sets of one or more Skolem-function nodes as their *immediate* children. If, given two variable nodes u and v, one node (say u) has a nonempty set of child nodes which is a proper subset of the child nodes for v, then we move the node u below the node v. All edges from v into children of u are removed. This procedure is done iteratively as long as such situations remain.

(4) A point that needs to be made clear in this definition is that any variables which have the same set of successor nodes in the skf-forest are collected to the *same* node. The complete skf-forest for any clause is unique (up to the ordering of each node's children, of course.)

Just as with the skf-forest, a complete skf-forest reflects only the sets **S** and **V**, and does not depend on the structure in the clause. To continue our last example, we had a hypothetical clause containing the set of Skolem functions:

$$\mathbf{S} = \{\, \text{skf1(X), skf2(X, Y), skf3, skf4(X, Y, Z), skf5(X, Z), skf6(X, Z, T), skf7(R)} \,\}\,.$$

Suppose **V** contained two variables, P and Q, which were not arguments to any of the Skolem functions. Thus $\mathbf{V} = \{\, \text{P, Q, R, T, X, Y, Z} \,\}$. Our complete skf-forest now looks like:



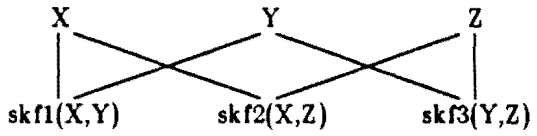
It is always possible to form the skf-forest, although in general it is only a directed graph, not properly a forest. Indeed, the situation can become quite a mess after the variables are added. In the last example, two separate branches of the tree rejoin at the node for skf4(X, Y, Z). Here, the skf-forest had this feature

from the set S, before the variables were placed into the structure. There are also examples of skf-forests which look fine until we add in the variable nodes.

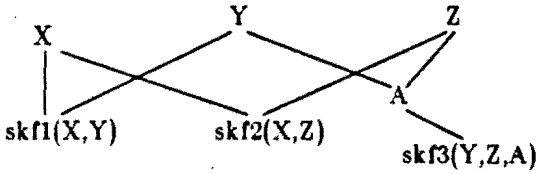
A simple example which becomes a mess only after the variables are added involves a clause with $\mathbf{V} = \{ X, Y, Z \}$ and $\mathbf{S} = \{ skf1(X,Y), skf2(X,Z), skf(Y,Z) \}$. The partial skf-forest contains merely the three unjoined nodes ("tree stumps") :

skf1(X,Y) skf2(X,Z) skf3(Y,Z)

but adding in the variable nodes produces:



To show just a simple example of the use of rule 3 in the definition, let's change the preceding example by adding A to **V** and replacing skf3(Y,Z) with skf3(Y,Z,A). Now the complete skf-forest will be:



Theorem 5. If a clause is quantifiable without any transformation which alters the complete skf-forest, then we may find a quantification for which every quantifier for a variable or Skolem function at any node **n** contains the quantifiers for all of its descendents within its scope.

Proof : Most of our work was done in the proof of theorem 3. It established that our claim holds for any Skolem function node, for it already showed how to include any lower variable or Skolem-function node within its scope. (A variable node that appears as a descendent to a Skolem function node **n** must be a variable in the argument list of a Skolem function in a node **d** that is a descendent to **n**, so it is covered in the argument given there.) Also, it is clear that any variable node must have its variable quantifiers containing all descendent Skolem function nodes in their scope, because the variable appeared in all of their argument lists.

All that remains is to prove our claim for any variable node which is a descendent of another variable node. If there is a Skolem function node between them, the conclusion is clear. So we have only to prove our claim for variable nodes which are father and child. But one variable node **u** is the father of another variable node **v** only if it was the case that **v**'s descendent nodes were a proper subset of **u**'s. Thus there is some node **e** which is a descendent of **u**, which is not a descendent node of **v**, and another node **m** which is a common descendent node to both **u** and **v**. These nodes may themselves be variable nodes, but if they are, we can find Skolem function nodes below them. Thus we may assume that **e** and **m** are Skolem function nodes. Select a Skolem function from each node **e** and **m**, calling them E and M. Suppose we are considering the quantifiers for variables U and V from **u** and **v**, respectively. We can use E to show that the $\forall U$ must not be within the scope of $\forall V$, so it must either contain $\forall V$, or be disjoint from it in scope. But we can also use M to show that the scope of the quantifiers for U and V must overlap. ■

Theorem 6. If the complete skf-forest for a clause is improperly branched, no quantification exists for the clause, unless a logical transformation can be found which eliminates the improper branch from the complete skf-forest.

Proof : We may assume that the structure was properly branched prior to the addition of variables, since otherwise theorem 4 establishes our claim. If the structure becomes improperly branched after addition of variables, then either a variable node or a Skolem function node exists which has more than one father. If it is a variable node, then it must either have more than one variable node father, or more than one Skolem function node father. (By construction, it is not possible for a variable node to have a variable node and also a Skolem function node as fathers, since all other incoming edges are removed when a variable node is placed under another variable node.) If it is a Skolem function node, it must have more than one variable node father. (Any incoming edges it may have had from a Skolem function node are removed when a variable node is placed over it, so it may not have a father of both kinds, and since the structure was properly branched before adding variable nodes, we know it does not have two Skolem function node fathers.) So we have three cases to consider.

Case 1 : Suppose we have a Skolem function node having two variable nodes as fathers.

> Select a representative from the three nodes, a Skolem function S and variables U and V. Since neither of the variable nodes may be fathers of the other, we know that neither variable node has a set of descendent nodes which is a subset of the other's. This implies that somewhere below are Skolem functions E and F, such that U appears in the argument list for E, but V does not, while V appears in the argument list for F, while U does not. We have our usual situation now: U and V are forced to have partially overlapping scopes. □

Case 2 : Suppose we have a variable node with two variable nodes as fathers.

> Here, we must have a variable M in the child node which has a nonempty set of descendent nodes and we must have a representative variable from each of the father nodes, say U and V. We know that U and V must have all of M's descendents as their mutual descendents, but they must both have some descendents which they do not share with each other, since neither is the other's father. Somewhere below, there must be a Skolem function containing U and not V in its argument list, another containing V and not U in its argument list, and another containing U and V both in its argument list (and M). This gives our usual contradiction. □

Case 3 : Suppose we have a variable node with two Skolem function nodes as fathers.

> Choose a representative variable X from the variable node and a Skolem function from each of the father nodes, calling them E and F. Since neither of the Skolem function nodes was the father of the other, we know that E must have some variable U in its argument list that is not in the argument list for F, while F must have some variable V which is in its argument list but which is not in E's argument list. We can see that U and V must both appear in the argument list for any Skolem functions that are in descendent nodes for X. But there must be at least one Skolem function which has X in its argument list, or X would have been in the special separate variable node which never has any father other than the Skolem constant node. We have once again the contradiction that U and V must have improperly nested scopes. □ ■

We are finally ready to give an algorithm for quantification of clausal forms. A four-step algorithm is given on the next page. Step 1 is performed on the entire set of clauses, then steps 2 and 3 on each of the clauses in the set. If step 2 or 3 fails for any clause, we try step 4 on the whole set, and return to steps 2 and 3 to try them again on each clause. If we fail again, we take another solution from step 4. This continues until success, or until step 4 runs out of suggestions.

Step 1. Since function names are meant to be consistent throughout a set of clauses, we must join those clauses which have Skolem functions with matching names before attempting to place quantifiers in them separately. All variable names which happened to match in the separate clauses must be given distinct names, since we have no information implying that they are the same objects, with the exception of those variables which appear as arguments to matching Skolem functions. These must be made to coincide, even though they may have had different names in the different clauses.

Thus, if a clause C_1 had $S_1 = \{ skf1(X,Y,Z) \}$ and $V_1 = \{ X,Y,Z,T \}$, while a subsequent clause C_2 had $S_2 = \{ skf1(P,Q,R) \}$ and $V_2 = \{ P,Q,R,X,T,V \}$, then we must join these before quantification since we know that skf1 in both places is the same entity. We know that X, Y, and Z in the first clause denote the same things as P, Q, and R, respectively denote in the later clause, since they occupy the same place in the arguments.[6] If we harmonized both of them to skf1(X,Y,Z), then all occurrences of P, Q, and R in the later clause must be replaced by X, Y, and Z. We may keep references to the variable T from the first clause as is, but we must deal with X, T, and V in the later clause as follows: occurrences of X must be replaced because we chose to rename occurrences of P to X. Let's rename it to A. T must be renamed because it matches T in the first clause and we have no reason to believe they should refer to the same object. Since step 2 will assume variables of the same name are the same thing, we must rename it. Let's rename it to B. Finally, V has managed to stay clear of any of the variables that were in the first clause or were renamed in the later clause, so we may keep it. We now have a clause that is the "and" of these two clauses, but now $S = \{ skf1(X,Y,Z) \}$ and $V = \{ X,Y,Z,T,A,B,V \}$.

Step 2. Form the complete skf-forest for each clause. If it is improperly branched, jump to step 4. Otherwise, begin at the top and move down the forest as follows: If the forest contains a Skolem constant node, we may place an existential quantifier for each such constant around the entire clause (their order is unimportant). The node for the variables which do not occur in any of the arguments of any Skolem functions is special. We will call these "free variables" and will set their node aside for the moment. Then we may move down as long as no branching occurs in the forest and place either universal or existential quantifiers in the order we find them (see theorem 5). If we meet a branch, we have multiple trees which must be quantified in such a way that their scopes are disjoint. If we can find distinct portions of the clause, each of which contains only variables and Skolem functions of one tree, plus possibly some free variables, and if each of the free variables occur in at most one of these portions, then we may continue by quantifying each portion using its subtree of the complete skf-forest, together with its portion of the free variables. If this succeeds until we are done, proceed to step 3. If it fails at any point, then there is no valid quantification possible for this clause, unless it is done after some transformation which either changes the complete skf-forest in some way, or alters the clause so that the splitting process succeeds when we hit a branch.

Step 3. If there are any free variables in the clause, we must see if we can place quantifiers for them in the portion of the clause to which they were assigned, if the clause was split. Generally they can be placed just inside of the last quantifier placed in that portion. This also goes if the clause never needed to be split, which occurs if the complete skf-forest did not have any branching in it. If this succeeds, then we have a valid quantification for the clause. If any of steps 1, 2, or 3 fail, our only hope is step 4.

Step 4. If the above has failed, the clause is not quantifiable without transformation. If that is the case, we have plenty we can try. Indeed, the problem is that there is too much we might try. One could perhaps consider all logically equivalent sets of CF, to see if any of those are quantifiable via steps 1, 2, and 3. Here the "perfect quantifier" would have at its disposal all tautologies of symbolic logic, or an algorithm to find all of them. It would also have an unlimited amount of time to perform its work. I was naturally unable to provide either of these in the program, but as a gesture toward that end, have the routines called as a group by the routine 'logsimpl'. These actually incorporate a fair number of identities for cleaning up expressions, and are made use of by later routines as well.

---

[6] This explains why we considered it improper to reorder the arguments of a Skolem function. We do allow certain renamings of variables in clauses, so if reorderings of them were also allowed, information would be lost.

4.3. 'simplify'

If quantification was successful, we now have an expression which is legal FOPC. In a sense our goal has been accomplished, for any legal FOPC expression is also a legal APC expression. However, not much can be said for the usual output of that step - it is generally a large, incomprehensible mass. Also, it may have put far too many variable names in the expression: our priority was to find a legal quantification if at all possible. Giving every variable that occurs in different clauses different names was the safest approach toward this end, but may have been a poor choice toward our next goal, making the expression palatable. For example, in quantifying:

alpha(X) :- beta(X).
alpha(Y) :- gamma(Y).

the quantification routine does not know whether the uses of X and Y in the two clauses can be assigned the same variable. Thus the clauses give the FOPC expression:

$$\forall X (\text{alpha}(X) \vee \neg \text{beta}(X)) \quad \&$$

$$\forall Y (\text{alpha}(Y) \vee \neg \text{gamma}(Y))$$

One of the tasks of 'simplify' is to recognize when such a situation can be simplified by joining quantifiers. On this example it will succeed, and produce a slightly better phrase:

$$\forall X [(\text{alpha}(X) \vee \neg \text{beta}(X)) \; \& \; (\text{alpha}(X) \vee \neg \text{gamma}(X))]$$

Then, once it has joined the variables, we can recognize further reductions that are possible; here the 'factor' routine from the previous step might be used to give the form:

$$\forall X [\text{alpha}(X) \vee (\neg \text{beta}(X) \; \& \; \neg \text{gamma}(X))]$$

We do not perform all of the reductions which might come to mind here. For instance, we would not make the last change mentioned, or take the steps that could make it a simple implication. Our aim at this stage is only to produce reductions in the quantifiers, as expressed by the following identities:

(1)	$\alpha \; \& \; \textbf{false} \; \& \; \beta$	$\models\!\dashv$	**false**	
(2)	$C_1 \; \& \; C_2$	$\models\!\dashv$	C_1 ,	if $C_1 \models\!\dashv C_2$
(3)	$\forall X \, f(X) \; \& \; \forall Y \, g(Y)$	$\models\!\dashv$	$\forall X \{ f(X) \; \& \; g(X) \}$	
(4)	$\exists X \, f(X) \vee \exists Y \, g(Y)$	$\models\!\dashv$	$\exists X \{ f(X) \vee g(X) \}$	
(5)	$\alpha \; \& \; (\beta \; \& \; \gamma)$	$\models\!\dashv$	$\alpha \; \& \; \beta \; \& \; \gamma$	
(6)	$\alpha \vee (\beta \vee \gamma)$	$\models\!\dashv$	$\alpha \vee \beta \vee \gamma$	
(7)	$\forall X \forall Y \{ f(X) \; \& \; g(Y) \}$	$\models\!\dashv$	$\forall X \{ f(X) \; \& \; g(X) \}$	
(8)	$\exists X \exists Y \{ f(X) \vee g(Y) \}$	$\models\!\dashv$	$\exists X \{ f(X) \vee g(X) \}$	

The code actually considers a number of generalizations in most of these identities. One reason we hold off on any others is to avoid any interference with what was to have been the other job of 'simplify': recognizing numeric quantifiers. It is expected that the situations in which this CF → APC translator could be applied would likely include situations where our APC → CF translator is also in use. Thus, it is important that the two translators are compatible; as mentioned in the section titled "Recent Background", the APC → CF translator attached special significance to a predicate "number". For example, a numeric existentially quantified expression, such as:

$$\exists(4) X [f(X)]$$

would be translated by that program into an expression with the simpler quantifier $\exists X$:

$$\exists X [\text{number}(X, 4) \; \& \; f(X)]$$

and then would be further translated, to produce clauses.

During the development of the CF → APC translator, it was first determined that number predicates may not be removed until quantification has taken place, then that harmonizing the occurrences of these predicates with the quantifiers from which they derived must wait until the eight identities in this section had attempted to reconstruct a "minimally quantified" expression, but should be attempted *before* further manipulations could be tried, and finally, when writing the code itself, it was discovered that there are situations in which the reassignment is ambiguous!

In view of this, it is not clear what to do with the present program to make it compatible with the APC → CF translator. The overall code design of the present program allows for the "reconstruction" of numeric existential quantifiers in its agenda, just after the **simplify** routine. There are a number of tests written into the **simplify** code and there were also a lengthy set of algorithms that were written to hunt for the appropriate patterns in order to perform such a reconstruction. Indeed, a full 13% of the code is devoted to this goal. It was then discovered that the "number predicate idea" is inadequate as a method for representing these quantifiers. For example, in FOPC the following two statements are equivalent :

$$\exists X [\exists Y (\text{alpha}(X, Y))]$$
$$\exists Y [\exists X (\text{alpha}(X, Y))] .$$

It was natural for this intuition to be carried over to APC, so it was apparently assumed that in APC, the ordering of adjacent existential quantifiers is interchangeable. However, in APC the following two statements are *not* equivalent:

$$\exists (5) X [\exists Y (\text{alpha}(X, Y))]$$
$$\exists Y [\exists (5) X (\text{alpha}(X, Y))] .$$

As a counterexample, consider a pair of variables taken from $U = \{ 1,2,3,4,5 \}$. It is quite true in this setting that:

" There are 5 objects X such that there is at least 1 object Y for which $[Y < X \ \vee \ Y > X]$. "

However, it is not the case that:

" There is at least 1 object X such that there are 5 objects Y for which $[Y < X \ \vee \ Y > X]$. "

Given the last sentence, however, the APC → CF translator first would rewrite it to:

" There is an X such that there is a Y for which $\{ \text{number}(Y, 5) \ \& \ [Y < X \ \vee \ Y > X] \}$. "

This will later become:

$$\{ \text{number}(\text{skf1}, 5) \ \& \ [\text{skf1} < \text{skf2} \ \vee \ \text{skf1} > \text{skf2}] \} .$$

which will result in two simple clauses. We cannot recover the proper order of the quantifiers from this.

Now it is likely that the "number predicate idea" for representing numerical existential quantifiers can be saved, if the predicate is made to carry further information which will keep quantifiers in the correct order. That is the subject of another project, perhaps. If such corrections are not made, however, then in the present program, the code to reconstruct numeric quantifiers from "number" predicates may as well be removed from the translator. (See the routine 'deskoltwo' and those that it calls.) Certainly in the APC → CF translator, the code which creates these predicates should be corrected either by enhancing the number predicates or by faithfully executing the expansions described earlier in the section titled: "Recent Background" . If this latter approach is taken, one can improve the code in the present program, first by removal of the 'deskoltwo' routines, and second by removing the tests which are currently performed in the **simplify** routines (which would become irrelevant).

At present, if one has no number predicates, or if one is given a set of clauses which has the number predicates in unambiguous positions, then this third step of our translator will complete its work, and we will either have a somewhat cleaner FOPC expression than before, or we will have an expression which is otherwise FOPC, but may also have some numeric existential quantifiers.

4.4. 'optimize'

Finally, the last set of routines starts with a legal APC expression and tries to produce a better expression. It considers a list of reductions which is detailed below. If it can make a sequence of simplifications, it tries them in every different order it can, to see if some orderings bypass roadblocks that others run up against. In this phase a massive number of expressions, which are minimal with respect to a certain sequence of transformations, all get produced and compared. This will result in a combinatorial explosion of solutions if a long sequence of reductions is possible. The various suggested improvements are "scored" with a rudimentary complexity measurement function, and the best form that can be found is finally divulged.

The list of identities **optimize** considers fall into seven groups, separated into routines that are called by the 'tryimprove' routine :

(1) The first set considers any ways that two or more numeric existential quantifiers can be combined, to reduce the total number of quantifiers. (See the 'combnum' routine.)

(2) A second set considers simple quantifiers, to see if any of those can be combined into a single numeric existential quantifier. (This is one *correct* way to recover these quantifiers from a set of CF, and does not involve the number predicates mentioned before. See the 'makenum' routine.)

(3) Another set looks for ways to combine simple quantifiers into the fifth form of numeric existential quantifier given in figure 1 (page 3). This form is not as concise as the first four forms, if such forms are possible, but is a fair improvement over the corresponding FOPC, if those reductions are not possible. This code also looks at ways to combine simple FOPC quantifiers with these numeric quantifiers, or to combine more than one of these numeric quantifiers to reduce the number of quantifiers further. (See 'combdist'.)

(4) A fourth set of identities are from the 'logsimpl' routines which occur early in the program code. These are the same identities which the 'quantify' routine tries if it becomes stuck. (See step 4 of the algorithm to quantify a set of CF.) These involve such identities as:

$$(A \,\&\, B) \vee (A \,\&\, C) \quad \models\dashv \quad A \,\&\, (B \vee C)$$
$$(A \vee B) \,\&\, (A \vee C) \quad \models\dashv \quad A \vee (B \,\&\, C)$$

These are basic examples: other identities also are used and in addition, the identities given above are generalized considerably in the actual code. (See 'onelogsimpl'.)

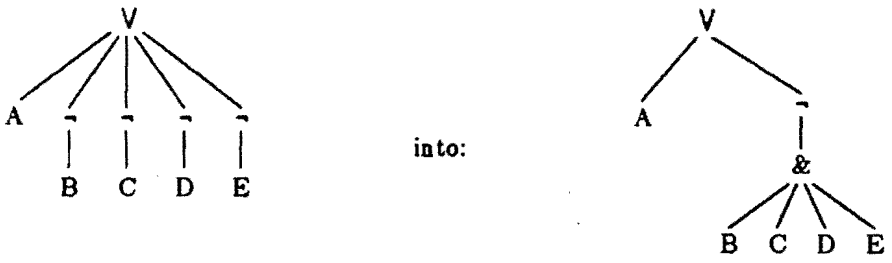
(5) This set involves some identities which are similar to those in the fourth set, but do not appear in the 'logsimpl' routines since they would not have helped the quantification process. These are the various identities for factoring negations. For example:

$$(\neg A) \,\&\, (\neg B) \quad \models\dashv \quad \neg (A \vee B)$$
$$(\neg A) \vee (\neg B) \quad \models\dashv \quad \neg (A \,\&\, B)$$
$$\forall X [\neg f(X)] \quad \models\dashv \quad \neg \exists X [f(X)]$$

The last identity above does not usually improve the readability of the expression, although it may make further simplifications possible.

In the generalizations of the first two there are some judgement calls made. For example, if one has an "or" with five parts, three of which are negated, should one turn it into a "not-and" in which two parts are negated? This would probably not constitute an improvement. On the other hand, what if the count is four to one? Here the result would be shorter, though it is not clear whether or not it has become more readable. Alternatively, if one has an "or" or "and" in which more than one term is negated, but not all of them, one can leave the topmost connector alone, but collect the negated terms below.

For instance, one could transform :



This is the move that is often chosen. (See 'factornegs'.)

(6) Our next set of routines hunts for ways to compound predicates in functions. This requires a careful search process, which needs to scan argument positions from left to right to avoid something akin to deadlocking, which would cause it to fail to recognize some gains that could be made, and it must start the scan over after a reduction is made. An instructive example is the expression:

$$f(A, C, E) \,\&\, f(A, D, E) \,\&\, f(B, C, E) \,\&\, f(B, D, E) \,\&\, f(F, C, E) \,\&\, f(F, D, E)$$

which eventually should become: $f(A \,\&\, B \,\&\, F, C \,\&\, D, E)$, *regardless of the order* of the original six terms. (See 'bothcompred'.)

(7) Finally, a set of identities are concerned with forming basic implications, forming the "higher" implications from the basic ones (forming '$\Leftrightarrow$' or '$\forall$' from '$\Leftarrow$' and '$\Rightarrow$' components), as well as moving implications to more strategic locations. (See 'formimpls'.)

In the next section, there are a number of examples which show these 7 sets of routines in action.

5. PERFORMANCE : RECOMMENDATIONS REGARDING FURTHER WORK

In the process of debugging the translator, I found that there are two points at which its performance causes concern. First, when a significant number of clauses are given in the input, all of which share Skolem functions in common, then the 'quantify' routines will cause a large spike in the stack usage. Secondly, if a significant chain of reductions are possible in the 'optimize' section, then it can spend an intolerable amount of time searching for its best answer.

Consider the following set of four examples, which were given to the translator in the course of testing. The tests were run on a Vax 750 machine[7] which was serving other users. The times given in examples 2 through 5 are "CPU" times, as reported by the unix command: "**ps ux**". After discussing these examples, some suggestions are given for improving performance.

Example 1. To start off with an easy one, consider the following Prolog clause:

$$\text{dist}(f(X), f(Y)) < E \text{ :- } \text{dist}(X,Y) < \text{skf1}(X,Y,E).$$

This is easily read in, quantified, simplified, and optimized. First, 'readinlist' takes the input and forms:

$$\text{dist}(f(X), f(Y)) < E \;\vee\; [\,\neg(\,\text{dist}(X,Y) < \text{skf1}(X,Y,E)\,)\,]$$

Next, 'quantify' determines that the skf-forest for this clause is simple:

---

[7] "Vax" is a trademark of Digital Equipment Corporation, Maynard, Massachussetts.

$$\{ X,Y,E \}$$
$$|$$
$$skf1(X,Y,E)$$

This indicates we can simply place down our quantifiers as :

$$\forall\ X\ \forall\ Y\ \forall\ E\ \exists\ A\ \{\ dist(f(X),f(Y)) < E \vee \neg\ dist(X,Y) < A\ \}$$

where skf1(X,Y,E) has been replaced with A. The 'simplify' routine finds nothing wrong with this quantification, and finally 'optimize' finds only one improvement to the overall expression. It changes the "or not" into "implies", giving:

$$\forall\ X\ \forall\ Y\ \forall\ E\ \exists\ A\ \{\ dist(X, Y) < A \Rightarrow dist(f(X), f(Y)) < E\ \}\ .$$

The reader versed in calculus may recognize this as the predicate logic formula for the *epsilon-delta definition of continuity*, which says that for any points x and y, and any ϵ, there is a δ such that if the distance from x to y is less than δ, then the distance from f(x) to f(y) is less than ϵ.

As a translation problem the above example was quite simple: total CPU time involves only a few seconds. It is representative of the performance of the translator for sets of input clauses that are unrelated. That is, even with a much larger set of input clauses, if no relations and reductions are apparent in the clauses, the translator seems to quantify them easily, and since it does not find any simplifications, the later steps are also rather painless.

Example 2. This next example catches the translator's interest a bit more:

```
:- skf7(X) = skf8(X) .
:- skf7(X) = skf9(X) .
:- skf8(X) = skf9(X) .
f(skf7(X), X) = (g([X], [X, skf7(X)], skf7(X) + 4) - 2 * X) .
f(skf8(X), X) = (g([X], [X, skf8(X)], skf8(X) + 4) - 2 * X) .
f(skf9(X), X) = (g([X], [X, skf9(X)], skf9(X) + 4) - 2 * X) .
```

The readinlist routine turns these 6 clauses into the list:

```
[ ¬ skf7(X) = skf8(X) ,
  ¬ skf7(X) = skf9(X) ,
  ¬ skf8(X) = skf9(X) ,
  f(skf7(X), X) = (g([X], [X, skf7(X)], skf7(X) + 4) - 2 * X) ,
  f(skf8(X), X) = (g([X], [X, skf8(X)], skf8(X) + 4) - 2 * X) ,
  f(skf9(X), X) = (g([X], [X, skf9(X)], skf9(X) + 4) - 2 * X)  ]
```

Next quantify checks for clauses which must be joined. The first and second clauses have the Skolem function skf7(X) in common, so they must be joined. This produces a clause containing all three Skolem functions, skf7(X), skf8(X), and skf9(X). Since one or more of these occurs in all of the other clauses, we need to join them into one big clause. Fortunately, the variable X is used consistently throughout, so that no renaming needs to be done. Once again, our skf-forest is unbranched, and it looks like:

$$\begin{array}{c} X \\ | \\ \{\, skf7(X),\ skf8(X),\ skf9(X)\,\} \end{array}$$

Thus, we can quantify the clause by renaming skf7(X), skf8(X), and skf9(X) as A, B, and C, respectively, and producing:

$$\begin{array}{l} \forall\, X\, \exists\, A\, \exists\, B\, \exists\, C \\ \quad [\ \neg (A = B)\ \&\ \neg (A = C)\ \&\ \neg (B = C)\ \ \& \\ \quad\ \ f(B, X) = g([X], [X, B], B + 4) - 2 * X\ \ \& \\ \quad\ \ f(C, X) = g([X], [X, C], C + 4) - 2 * X\ \ \& \\ \quad\ \ f(A, X) = g([X], [X, A], A + 4) - 2 * X\ \ \ ] \end{array}$$

The simplify routine finds no changes it can make, and zips right through, but the optimize routine chews on this for some time. After about 30 minutes of CPU time, the answer is given:

$$\forall\, X\, \exists (>=3)\, C\ f(C, X) = g([X], [X, C], C + 4) - 2 * X$$

Without too much trouble, the translator determined first that the quantifiers $\exists$ A and $\exists$ B could be joined to give $\exists(>=2)$ B, and then that the quantifiers $\exists(>=2)$ B and $\exists$ C could be joined to give $\exists(>=3)$ C. The reason 30 minutes of CPU time were expended was partly because the translator also determined the other orders in which these could be combined (first combine B and C and then combine that with A, or first combine A and C, then combine that with B). Also it was partly because optimize also tried a couple of other ideas that did not pan out as well, like factoring the negations in the terms of the "and" and then either forming an implication, or considering all of the ways to form compound predicates.

Thirty minutes might be an acceptable wait, however our third and fourth examples really overtax the capacity of the translator ...

Example 3. The following five clauses were input:

$$skf3(X) = f(X,V)\ ;\ skf4(X, Y) = X\ :-\ f(skf3(X), X) = g(V),\ Y < X.$$

$$skf5(X) < X\ :-\ U = f(X, skf6(X, U)).$$

$$skf5(X) < X;\ f(U, X) = g(skf6(X, U)).$$

$$:-\ Z = X,\ U = f(X, skf6(X, U)).$$

$$f(U, X) = g(skf6(X, U))\ :-\ Z = X.$$

and the quantification routine notes that it must combine the last four, so it pauses visibly to do that. The skf-forests for both clauses are simple and we eventually get:

$$\begin{array}{l} \forall\, X\, \exists\, B\, \forall\, Y\, \exists\, C\, \forall\, V \\ \quad \{\, (\, B = f(X, V)\ \vee\ C = X\,) \qquad\qquad \vee \\ \quad\ \ (\, \neg\, f(B, X) = g(V)\ \vee\ \neg\, Y < X\,) \qquad \}, \\ \forall\, X\, \exists\, D\, \forall\, U\, \exists\, E\, \forall\, Z\, \forall\, A \\ \quad \{\, (\, D < X\ \vee\ \neg\, U = f(X, E)\,) \quad \& \\ \quad\ \ (\, D < X\ \vee\ f(U, X) = g(E)\,) \quad \& \\ \quad\ \ (\, \neg\, Z = X\ \vee\ \neg\, U = f(X, E)\,) \quad \& \\ \quad\ \ (\, f(U, X) = g(E)\ \vee\ \neg\, A = X\,) \qquad \} \end{array}$$

The simplify routine performs four transformations on this one. First, we note that the very last two quantifiers, ∀ Z and ∀ A may be combined to give a simpler expression. Thus these become simply ∀ Z and all occurrences of A are replaced with Z. This remedies the over-cautiousness of the quantify routine alluded to earlier. Second, the nesting of the V's in the first of the two clauses is unnecessary, so these are flattened. Third, we take the two clauses and form from them a single "and" expression, harmonizing any variables as necessary. Finally, we note that the "and" of the two expressions which begin with ∀ X can be rephrased with only one quantifier, ∀ F, with the "and" moved inside. Replacing occurrences of X with F gives us the expression:

$$
\begin{aligned}
&\forall\, F \\
&\quad [\,\exists\, B\, \forall\, Y\, \exists\, C\, \forall\, V\, \{\, B = f(F, V) \vee C = F \vee \neg\, (f(B, F) = g(V)) \vee \neg\, (Y < F)\, \}\,] \quad \& \\
&\quad [\,\exists\, D\, \forall\, U\, \exists\, E\, \forall\, Z\, \{\, (\, D < F \vee \neg\, U = f(F, E)\,) \quad \& \\
&\qquad\qquad\qquad\qquad (\, D < F \vee f(U, F) = g(E)\,) \quad \& \\
&\qquad\qquad\qquad\qquad (\, \neg\, (Z = F) \vee \neg\, U = f(F, E)\,) \quad \& \\
&\qquad\qquad\qquad\qquad (\, f(U, F) = g(E) \vee \neg\, Z = F\,) \quad \}\,]
\end{aligned}
$$

This is passed to the optimize routine. Although considerably shorter than before, this expression is far from minimal, so at this point, an explosion of possible improvements presents itself. These incremental improvements can be performed in sequences of up to about 15 in a row, which results in a massive tree search. Just listing the distinct *first* moves gives a sequence of the following routines ('onelogsimpl' appears 16 times because it was able to find 16 *different* ways in which to *start* simplifying the clause!) :

(1) onelogsimpl
(2) onelogsimpl
(3) onelogsimpl
(4) onelogsimpl
(5) onelogsimpl
(6) onelogsimpl
(7) onelogsimpl
(8) onelogsimpl
(9) onelogsimpl
(10) onelogsimpl
(11) onelogsimpl
(12) onelogsimpl
(13) onelogsimpl
(14) onelogsimpl
(15) onelogsimpl
(16) onelogsimpl
(17) factornegs
(18) factornegs
(19) formimpls
(20) formimpls
(21) formimpls

After 3735 minutes of CPU time (that's right, about 62 hours!), the answer is finally given. I was impressed by the answer when it finally came, for it was slightly simpler than the one I had come up with. Whereas I had produced the clauses from the APC expression:

$$\forall\, F\, [\,\forall\, Y\, \exists\, C\, \{\, Y < F \Rightarrow C = F\,\} \Leftrightarrow \forall\, B\, \exists\, V\, \{\, f(B, F) = g(V) \;\&\; \neg\, B = f(F, V)\,\}\,]$$

The translator favored:

$$\forall\, F\, [\,\forall\, Y\, \exists\, C\, \{\, Y < F \Rightarrow C = F\,\} \nabla\, \exists\, B\, \forall\, V\, \{\, f(B, F) = g(V) \Rightarrow B = f(F, V)\,\}\,]$$

I had never realized that this latter, logically equivalent form existed, which actually scored slightly better than my original (according to the complexity measure in use).

As pleased as I was with the result, I took no pleasure in the time it had consumed to arrive at it. Surprisingly, after only about 20 minutes of CPU time, the eventual answer had been found! The translator did not *report* any result, however, until all the world had been searched for an improvement. This can be checked with the use of the 'ptoaverbose' version of the main routine, which prints the upper 3 branches of the tree-search it is engaged in, plus any new "best-answer-so-far" that it finds. Using the translator in this mode provides a way to get the answer more quickly, but should not be considered a real solution to the performance problems.

Example 4. The following contrived set of clauses:

```
g(X, skf1(X, Y)) ; g(X, skf2(X, Y))  :-  f(X, skf1(X, Y)) , f(X, skf2(X, Y)) .
g(X, skf1(X, Y)) ; g(X, skf2(X, Y))  :-  f(Y, skf1(X, Y)) , f(Y, skf2(X, Y)) .
g(X, skf1(X, Y)) ; g(Y, skf2(X, Y))  :-  f(X, skf1(X, Y)) , f(X, skf2(X, Y)) .
g(X, skf1(X, Y)) ; g(Y, skf2(X, Y))  :-  f(Y, skf1(X, Y)) , f(Y, skf2(X, Y)) .
g(Y, skf1(X, Y)) ; g(X, skf2(X, Y))  :-  f(X, skf1(X, Y)) , f(X, skf2(X, Y)) .
g(Y, skf1(X, Y)) ; g(X, skf2(X, Y))  :-  f(Y, skf1(X, Y)) , f(Y, skf2(X, Y)) .
g(Y, skf1(X, Y)) ; g(Y, skf2(X, Y))  :-  f(X, skf1(X, Y)) , f(X, skf2(X, Y)) .
g(Y, skf1(X, Y)) ; g(Y, skf2(X, Y))  :-  f(Y, skf1(X, Y)) , f(Y, skf2(X, Y)) .
f(X, skf1(X, Y)) ; f(Y, skf1(X, Y))  :-  g(X, skf1(X, Y)) , g(Y, skf1(X, Y)) .
f(X, skf1(X, Y)) ; f(Y, skf1(X, Y))  :-  g(X, skf2(X, Y)) , g(Y, skf2(X, Y)) .
f(X, skf1(X, Y)) ; f(Y, skf2(X, Y))  :-  g(X, skf1(X, Y)) , g(Y, skf1(X, Y)) .
f(X, skf1(X, Y)) ; f(Y, skf2(X, Y))  :-  g(X, skf2(X, Y)) , g(Y, skf2(X, Y)) .
f(X, skf2(X, Y)) ; f(Y, skf1(X, Y))  :-  g(X, skf1(X, Y)) , g(Y, skf1(X, Y)) .
f(X, skf2(X, Y)) ; f(Y, skf1(X, Y))  :-  g(X, skf2(X, Y)) , g(Y, skf2(X, Y)) .
f(X, skf2(X, Y)) ; f(Y, skf2(X, Y))  :-  g(X, skf1(X, Y)) , g(Y, skf1(X, Y)) .
f(X, skf2(X, Y)) ; f(Y, skf2(X, Y))  :-  g(X, skf2(X, Y)) , g(Y, skf2(X, Y)) .
```

cause the quantify routine to consume a vast amount of stack space, roughly 175,000 stack variables at its peak. Once the set has been quantified, which requires joining every one of the clauses, then simplify hands it immediately over to the optimize routine as the following:

```
∀ X ∀ Y ∃ A ∃ B [  { g(X, A) ∨ g(X, B) ∨ ¬ f(X, A) ∨ ¬ f(X, B) } &
                   { g(X, A) ∨ g(X, B) ∨ ¬ f(Y, A) ∨ ¬ f(Y, B) } &
                   { g(X, A) ∨ g(Y, B) ∨ ¬ f(X, A) ∨ ¬ f(X, B) } &
                   { g(X, A) ∨ g(Y, B) ∨ ¬ f(Y, A) ∨ ¬ f(Y, B) } &
                   { g(Y, A) ∨ g(X, B) ∨ ¬ f(X, A) ∨ ¬ f(X, B) } &
                   { g(Y, A) ∨ g(X, B) ∨ ¬ f(Y, A) ∨ ¬ f(Y, B) } &
                   { g(Y, A) ∨ g(Y, B) ∨ ¬ f(X, A) ∨ ¬ f(X, B) } &
                   { g(Y, A) ∨ g(Y, B) ∨ ¬ f(Y, A) ∨ ¬ f(Y, B) } &
                   { f(X, A) ∨ f(Y, A) ∨ ¬ g(X, A) ∨ ¬ g(Y, A) } &
                   { f(X, A) ∨ f(Y, A) ∨ ¬ g(X, B) ∨ ¬ g(Y, B) } &
                   { f(X, A) ∨ f(Y, B) ∨ ¬ g(X, A) ∨ ¬ g(Y, A) } &
                   { f(X, A) ∨ f(Y, B) ∨ ¬ g(X, B) ∨ ¬ g(Y, B) } &
                   { f(X, B) ∨ f(Y, A) ∨ ¬ g(X, A) ∨ ¬ g(Y, A) } &
                   { f(X, B) ∨ f(Y, A) ∨ ¬ g(X, B) ∨ ¬ g(Y, B) } &
                   { f(X, B) ∨ f(Y, B) ∨ ¬ g(X, A) ∨ ¬ g(Y, A) } &
                   { f(X, B) ∨ f(Y, B) ∨ ¬ g(X, B) ∨ ¬ g(Y, B) }  ]
```

As the optimize routine searched through the tree of all possible sequences of simplifications, it seemed to be hitting a maximum depth of about 30 simplifications. That is, there were cases where it could simplify the input clause, simplify the simplification, simplify that further, etc., 30 times. After about a half hour of CPU time had elapsed, it had already found what I believe is the best form:

$$\forall X \forall Y \exists A \exists B \{ f(X \vee Y, A \& B) \Leftrightarrow g(X \& Y, B \vee A) \}$$

but I did not have the temerity to let it continue searching until all possibilities had been considered. I have not attempted to estimate how long that would have required, but suspect it would be quite a bit more than the 62 hours that example 3 required.

I have not considered ways to improve the space usage of the 'quantify' routine, since so far it has only proved a minor annoyance in even the last example given. I do see a number of approaches to the improvement of the 'optimize' routine, such that it would take considerably less time to do its work.

First, the routine is tremendously inefficient in the fact that it tries all possible sequences of improvements. Originally, I felt this would be the safe way to ensure that we had the best possible result, given the available moves. However, in the example 3 above, I found the 'logsimpl' routine (which seems to be the most prolific) essentially retrying different orders of moves which in the end always resulted in the same thing. If one can simplify the left branch of an expression, then the right branch, there is certainly nothing new to be found by trying the right branch simplification first, then the left branch.

One way to avoid retracing identical steps is to keep a list of all of the forms so far examined. This method would have its own problems, since it could consume vast amounts of storage, and could bog down with the time it spends searching for matching forms. In the situations where no improvements can be made, this would not affect performance. In the situations where one can make a sequence of up to about 10 improvements (e.g. example 3 above), I am betting that this would present a trade off one would gladly accept. In situations like example 4, where one has a really large set to consider, I am not as certain that one gains anything.

Second, while I have precious little experience with game theory, this search process is quite like the kind of searching one does to examine all possible chess moves from a given position. This is generally not possible, so the experts in game theory have developed methods of pruning tree searches, so that only the most promising paths are examined. This is analogous to the way a person would perform this task - only the "promising" approaches should be considered. I do not have any opinion as to what could be the criteria for deciding if a sequence is promising.

Third, taking the last idea to an extreme, there may be situations in which we are certain we have done the right thing. A number of the simplification procedures which occur in the course of the translation are almost certainly the "right thing to do" whenever they can be done. In this particular context, when a person is attempting to simplify an expression, he or she may note that some move can cut the size of the clause in half (such as combining two simple implications into an "if and only if" implication). In such a situation, the person will likely never need to consider what gains might be possible if this chance is passed up. It may be possible to do this with the seven sets of optimization routines. That is to say, we may be able to say, "If you can form an 'if and only if', then do so, and never look back."

Further, if we are not quite so confident of some moves, say for example we are concerned that a 'factornegs' move might mask a better 'logsimpl' move, then we could still avoid the need for backtracking if we can add a clause to the 'logsimpl' routine which recognizes overly zealous work on the part of 'factornegs'. In a different problem setting, this might merely substitute a combinatorial explosion of code length for a combinatorial explosion of tree search time, but in this setting of APC-expression recognition, since virtually all of the moves considered will shorten the expression, I expect that this could reasonably be done. It may be possible to eliminate nearly all of the branchings in the

search process, using code that is not much longer than the routines are now.

We have now an abundant number of suggestions for improving and correcting the translators, sprinkled throughout this paper. These are summarized in figure 4 below, for the sake of anyone who is considering further work on the translator. It is significant however, that we do not even know what a typical set of inputs might be for such a translator. It is my feeling that improvements or corrections to either an APC → CF or and CF → APC translator should wait until there is an application out there which is begging for them. Once we have such an application running, we can see what its actual needs are. Only then will we be in a position to really see the demands that one might want to place on these programs.

Figure 4 : Some directions for further work :
(In the order mentioned in the paper, with no priorities indicated)

(1) The translator could be augmented to handle APC *annotations* (see reference [4]).

(2) The APC → CF translator should be corrected in its treatment of *numeric existential quantifiers*, and the CF → APC code in the 'simplify' routine modified to operate in corresponding fashion. (Especially the 'deskoltwo' routines.)

(3) One could explore ways to reduce the stack space required by the 'quantify' routine for inputs which require joining a large number of clauses.

(4) A number of suggestions have just been given for improvement of the time required by the 'optimize' routine. One of these could be pursued, or perhaps still another approach could give the best results.

REFERENCES

[1] "Predicate Logic as a Programming Language", by Robert Kowalski,
Proceedings of the IFIP Congress, 1974, pp. 569-575

[2] *Programming in Prolog* by W.F. Clocksin and C.S. Mellish
© 1981 Springer-Verlag (Berlin and New York) Chapter 10, pages 208-217

[3] *From Frege to Godel : A Survey Book in Mathematical Logic, 1879-1931* edited by J. Van Heijenoort
© 1967 Harvard University Press (Cambridge)

[4] *Machine Learning, An Artificial Intelligence Approach* edited by Ryszard S. Michalski,
Jaime G. Carbonell, and Tom M. Mitchell
© 1983 Tioga Publishing Company (Palo Alto California) Chapter 4

[5] "Translating Annotated Predicate Calculus to PROLOG", by Kah-Eng Pua, Report ISG 84-1,
UIUCDCS-F-84-918, Department of Computer Science, June 1984

APPENDIX A

The Internal Representation Used by the Program

The program uses a data structure described below to represent the logical statements being translated. Statements in APC and FOPC are built up inductively from basic terms using various connectives and quantifiers, so we show the representation for each kind of building block, then give two examples of typical statements constructed from them. A basic term is represented by itself. For compound structures, we have:

Logical Statement	Internal Representation
¬ Term	[not, Term]
Term1 & Term2	[and, Term1, Term2]
Term3 ∨ Term4	[or, Term3, Term4]
Term1 ⇨ Term2	[implies, '==>', Term1, Term2]
Term3 ⇦ Term4	[implies, '==>', Term4, Term3] (in other words, we never internally store a backward implication ...)
Term5 ⇔ Term6	[implies, '<=>', Term5, Term6]
Term7 ∇ Term8	[implies, '\=\', Term5, Term6]
∀ X (alpha)	[forall, X, alpha]
∃ X (beta)	[thereis, X, beta]
∃.[X,Y,Z] (gamma)	[thereis, [X,Y,Z], gamma]
∃ (>=n) X (delta)	[therenum, ['>=', n], X, delta]
∃ (n) X (epsilon)	[therenum, ['=', n], X, epsilon]
∃ (m..n) X (iota)	[therenum, [m, n], X, iota]

As a simple example, consider the expression

$$(X > 7) \vee \{ Y < 2 \ \& \ \neg(Z = 3) \}$$

this is an "or" of two terms, the second of which is an "and". Internally, we would have the list structure:

[or, (X > 7), [and, Y < 2, [not, (Z = 3)]]]

As a more complex example, the following was mentioned in the paper:

$$\exists(4)x \ \forall y \ \forall z \ [\ (\text{fun}(x \ \& \ y, z)) \Rightarrow (\ \exists t \ [\ 0 > \text{temp}([x, y], \text{alpha} \vee z, \text{beta} \ \& \ t) \ ] \ ) \ ]$$

This would have the following internal representation (indented to improve readability) :

```
[therenum, ['=', 4], X,
  [forall, Y,
    [forall, Z,
      [implies, '==>',
        fun([and, X, Y], Z),
        [thereis, T, 0 > temp([X,Y], [or, alpha, Z], [and, beta, T])]
] ] ] ]
```

APPENDIX B

An Overview of the Translator Code

The following shows the code for the CF → APC translator down to the second level. That is, it shows the body of each of the five routines called by the main clause:

```
readinlist(InputFile, ClauseList) :-
   see(InputFile), getclauses(FormalList), seen,
   literalvars(FormalList, ClauseList), !.

quantify([], []).
quantify(ClauseList, Quantified) :-
   clausesmerge(ClauseList, GrandSkfList, CompactedList),
   quantlist(GrandSkfList, CompactedList, Almost),
   skfstovars(Almost, Quantified).

simplify(Quantified, APCClause) :-
   checkallfalse(Quantified, Step1),
   checkredclses(Step1, Step2),
   allforone(Step2, Step3),
   listshortquants(Step3, Step4),
   deskoltwo(Step4, APCClause).

optimize(InClause, _) :-
   score(InClause, ToBeat),
   assert((bestsofar :- [ToBeat, InClause])),
   tryimprove(InClause, NewSolution),
   score(NewSolution, Rank),
   clause(bestsofar, [Standard, _]),
   Rank < Standard,
   retractall(bestsofar),
   assert((bestsofar :- [Rank, NewSolution])),
   fail.
optimize(_, BestForm) :-
   clause(bestsofar, [_, BestForm]),
   retractall(bestsofar).

writeout(Clause) :-
   print('----------'), nl,
   writerecurse(Clause),
   nl, nl, print('----------'), nl.
```

The names of all routines, listed in the modules which contain them

module pmain:

findmod(Name) ... finds the module containing a given routine (Name)
edmodof(Name) ... finds and edits the module containing a given routine (Name)

module p0:

prologtoapc(InFile) ... main clause
prologtoapc(InFile, OutFile) ... main clause, writes out result to OutFile.
ptoaverbose(InFile, OutFile) ... talkative main clause

[Below are utility routines, used in more than one place]

maxof, minof, member, nondisjoint, within, list, subset, equalsets, varchk, locate, locatesubof, getmember, append, permlist, pickone, chooset, splitnonempty, choosetsof, crossprod, listifnot, form1, form2, ifmults, complement, clearandcount, pullcountof, prepend, changelem, findanddrop, intersect, distnots, logsimpl, logsimplmems, maylogsimplmems, partsmatch, maypartsmatch, logshorten, flatten, mayflatten, listflatten, maylistflatten, raisesubswith, raiseanywith, remonedup, matchfactor, mixfactor, oppfactor, findwithcommon, recombine, findvars, insidevars, newvars, genunique, decode, nexttry, rmnonz, successor, dropmatches, vartran, varargtran, findskfs

module p1:

readinlist,
getclauses, getsides, makedis, rephrase, djoin, literalvars

module p2:

quantify,
clausesmerge, oneclausemerge, firstmatch, samefunctin, functin, skfjoin, commonfuncts, correspond, argmerge, parallelmerge, quantlist, quantclause, getfrees, trybelow, makeforest, plantskf, listargs, findplot, findsubplot, allocate, shovel, digup, assign, crossrefvars, joinifany, findandpull, subjoinifany, addifnotin, onelevelin, groupargs, gatherup, listgather, fenceoff, buildfence, parallelplace, quantsublist, addquants, skfstovars, varsfromskfs, oneskftovar

module p3:

simplify,
checkallfalse, checkredclses, clauserepeated, onlyrenamed, freesub, allforone, oneforall, listforall, topforall, pulltogether, nonumfsof, nonumfsinside, varcondense, striponequant, harmonize, listshortquants, shortquants, sepvars, distinctgroups, deskoltwo, findnumfs, insidenumfs, makebexis, numfgroup, numfmatches, resmatches, pullmultfirst, rednumfs, distnumfs, moredistns, pullnotall, pullmaybeall, renamenumfs, newnumfvars, replnumfs, makecorres, proximtran, divvyup, insidedivvy, oneorless, nonumfsfrom, onenumffrom, splitproxim, numfassign, parallelparcel, pullsubcountof, splitquant, logrednumfs, forcesubgp, forcesplitof, renidentnumfs, findonechange, onenumfof, zaponenumfof, listfochg, formnewfunctsof, dropnumfs, listdropnfs, pullportion, pullnumfof, easyorhardbex, candropnumf, easybquant, collectquants, collectquant, splitvars, listsplitvs, keepwhatsin, requant, formattosubs, mayneg

module p4:

optimize,
score, listscore, tryimprove, moreimprove, combnum, listcnum, uneqsym, symvars, logeqmemb, logequal, listlogeq, permlogeq, permtraneq, symterm, antisymterm, symargs, makenum, listmnum, simsnumand, simsand, skipsims, combdist, listcdist, mixexisand, anyexisand, listfdene, remixquant, onelogsimpl, onememlogsimpl, listfnegs, factornegs, betternums, pullexis, bothneg, pullnegquant, joinnum, reformexis, dropallnegs, pullmbpson, pulloneneg, pullrestneg, bothcompred, compred, maycompred, listcompred, maylistcompred, repcompred, onecompred, findfunctsof, morefunctsof, notsublogical, groupfuncts, putinbyname, matchbutfor, otherbutfor, onecpat, morecpat, cpatjoin, scanargs, replacearg, format, listformis, maylistfis, formimpls, symimpls, justoneneged, anynegs, forceimpls, stripfopquants, distquants, noexisinboth, forceortoimpl, negatelist, negate, converse, lognegation, lognegmems, onenegmatch

module p5:

writeout,
writerecurse, connwrite

module p9:

[Below are diagnostic routines, that run various chunks of the program.]

step1, step2, step3, step4, optdebug, tryimptst, moreimptst, diagout

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-F-86-954	2.	3. Recipient's Accession No.

4. Title and Subtitle

A Translation System from Clausal Form into Annotated Predicate Calculus

5. Report Date

May 1986

6.

7. Author(s)

Mark S. Goldfain

8. Performing Organization Rept. No.

9. Performing Organization Name and Address

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No.

N00014-82-K-0186
NSF DCR 84-06801
N00014-K-85-0878

12. Sponsoring Organization Name and Address

Office of Naval Research, Arlington, VA
National Science Foundation, Washington, DC

13. Type of Report & Period Covered

14.

15. Supplementary Notes

16. Abstracts

This paper discussed a Prolog program which performs translation from "Clausal Form (CF)" into "Annotated Predicate Calculus (APC)" expressions. A complementary system which translates from APC into Clausal Form also enters into the discussion. A number of theorems are given in the course of a detailed discussion of the translation algorithms. A final section gives some sample test data, from which a general critique of the system's capabilities and performance are drawn. This leads into a discussion of possible further research work.

17. Key Words and Document Analysis. 17a. Descriptors

Logic Programming
Expressive Power
Quantification
Skolemization
De-Skolemization
Clausal Forms
Horn Clauses
First Order Logic
Annotated Predicate Calculus

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 38
	20. Security Class (This Page) UNCLASSIFIED	22. Price