The DUCK Manual

Drew McDermott

YALEU/CSD/RR #399
June 1985

The DUCK Manual

Drew McDermott

YALEU/CSD/RR #399
June 1985

This work was supported in pary by NSF grant DCR-8407077 and ONR N00014-83-K-0281. Thanks for comments by Liz Allen, Bradley Berg, Ruven Brooks, Chris Moorse, and especially Chris Riesbeck.

Table of Contents

List of Figures

ABSTRACT

Duck is a deductive database designed to work as an adjunct to Artificial-Intelligence application programs written in Lisp. It provides the following features:

- Backward chaining logic interpreter
- Forward chaining
- Data dependencies and reason maintenance
- Integration of deduction with Lisp computation
- Multiple databases
- Syntax checking for deductive rules
- Trace package,
- Interactive examination of successful and failed deductions
- Generation of quasi-English versions of rules

1. INTRODUCTION

Duck is a Lisp-based system for writing predicate-calculus rules. As such, it allows a flexible mixture of several styles of programming:

- Rule-based systems: In which expert knowledge is captured in transparent, modular chunks.
- Reason maintenance: In which withdrawal of assumptions causes conclusions to be revised.
- Logic programming: SeaRches based on unification and backtracking.
- Applicative and imperative symbolic processing: Classic Lisp styles.

A practical AI program may include all of these programming styles. Its overall control structure will be written in Nisp, the portable Lisp dialect Duck is written in. [McDermott 83a] At key decision points, expert knowledge will be brought to bear in the form of Duck rules. These rules can perform any sort of automated reasoning, from simple syllogisms to arbitrary Horn-clause programming. They can very easily slip back into Lisp if efficiency or naturalness dictates.

All of these programming styles share one mechanism in common: the Duck Database. This is the set of all Duck formulas currently "believed" (visible to programs). It includes simple facts as well as rules. The basic channel of communication is through a sort of pattern matching called *unification*: the paradigmatic interaction is a query to find all formulas of a certain form. Out of this building block complex deductions can be built. Duck will keep track of them for you.

Unlike many rule-based systems, the Duck rule system is based on the predicate calculus. Rules are not allowed to do arbitrary computations; instead, what they do must be thought of as a *deduction*: a conclusion of true beliefs from true premisses. This means that the conventions the rules use must be thought out carefully, and it means that the content of each rule is transparent to anyone familiar with the vocabulary. Duck provides tools for creating and maintaining such a vocabulary.

To use the predicate calculus, you must analyze the domain of interest as consisting of *individuals* related by *properties*. This is not really a serious restriction, since the individuals may be quite abstract. A statement like "Every battleship in the Mediterranean last Tuesday belonged to the enemy" may be seen as relating the individuals "Mediterranean," "last Tuesday," and "the enemy," using properties like "X is a battleship," "X is located in Y," and "X happened at time Y."

Because Duck allows a flexible array of programming styles, there are a variety of ways to describe it. In this manual, I have chosen to explore it first as a database adjunct to a Lisp program, then as an autonomous deductive retriever, and finally as a reason maintenance system.

It is essentially impossible to use Duck as a stand-alone programming language. All realistic applications will contain as many Lisp functions as predicate-calculus formulas. The Lisp dialect Duck is written in is called Nisp. Familiarity with Nisp is assumed in what follows. You will need to consult the *Nisp Manual* [McDermott 83a] as well as this one in order to write a Nisp/Duck program. However, I hope most of the code here will be readable without intimate familiarity with Nisp. The major novelty in Nisp is its reliance on *compile-time types* to a greater degree than most Lisp dialects. Variables may declared to be of various types at compile time, to allow type checking and coercion, and, for structured types, to allow slot references to be compiled into efficient code. Type declarations are interleaved with normal variable lists, as in this example:

```
(FUNC FOO fixnum (numstruc S)
   (LET (fixnum (B (+ (: ASLOT S) 1)))
      (* B B)   ))
```

To avoid losing the types in the fog, they are written in the opposite case from normal code. Here `numstruc` and `fixnum` are types. `S` is of type `numstruc`; `B` is of type `fixnum`, as is the result returned by `FOO`. With the types compiled out, this code might look like this:

```
(DEFUN FOO (S)
   (LET ((B (+ (numstruc-ASLOT S) 1)))
      (* B B)   ))
```

Nisp is implemented as a macro package that sits on top of a host Lisp dialect. (Porting Duck to a new machine is at most a matter of implementing Nisp on whatever dialect is available there.) Different Lisp dialects have different lexical conventions. Some use upper-case as the default, some use lower, and some do not distinguish upper from lower case. Some use "\" as an "escape character," and some use "/" or some other character. (The escape character is used to force the next character to be treated as alphabetic.)

A Duck program must respect the conventions of its host dialect. So porting Duck to a new machine will require running it through a program called `DIARECT` (for "dialect rectifier") that transforms upper to lower case, adjusts the escape character, and so on. This program comes with Duck.

In this manual, the lexical conventions that are followed are those of the T dialect of Lisp. [Rees 82, Rees 84] That is, the names of functions and variables are upper case. The names of data types are lower case. The escape character is "\". Common Lisp is just like T except that all colons must be slashified. ZetaLisp is just like Common Lisp except that the escape character is "/" instead of "\". Franz reverses the case conventions; the names of functions and variables will be lower case; of types, upper case. The escape character is \.

A special font will be used for Lisp symbols, such as `CAR` and `CDR`. Italics will be used for

"syntactic variables," as in descriptions of functions: (FOO *symbol number*).

Another lexical issue to be aware of is the way the empty list is printed and referred to in different dialects. Although all will accept () at read time, some will print this as nil or NIL. It is safest to get in the habit of typing ().

Duck and Lisp look similar syntactically, in that programs written in them consist of expressions of the form (*operator -arguments-*). However, this syntax covers a lot of semantic variation. In Lisp, the *operator* is a function or "magic word," as discussed below. In Duck, the *operator* is normally a predicate, which makes an assertion about its arguments. Such expressions may also occur as *goals*, things to be proved. Posting a goal is like calling a function, but the values are returned in a quite different way, to be made clear in Chapter 3. I hope it is clear from context whether a given operator is a Lisp function or Duck predicate; they are definitely not interchangeable.

In this manual, most Lisp functions presented may be assumed to evaluate their arguments. In Lisp, some of the most important "operators" resemble functions syntactically, but do not evaluate their "arguments" in the usual way. For instance, (COND ...) takes several expressions and treats them as "COND clauses," with a completely idiosyncratic syntax. In no sense does it evaluate each clause and pass the results to the "COND function." Symbols like COND are used to implement Lisp's syntax, piece by piece, and are known as "FEXPRs" or "syntax descriptors," or "reserved words," or *magic words*.

I will adopt the following convention: Wherever a magic word is described, it will be prefixed with the characters "*/" . So a description of COND might begin: */(COND *-clauses-*): ... .

There are also "Duck magic words," things that look like predicates syntactically, but behave differently. An example is THCOND, which (not surprisingly) plays a role in Duck like that of COND in Lisp. I will use the same */ notation when introducing such "pseudo-predicates."

By the way, several of these pseudo-predicates have names starting with the letters TH. These functions existed in Micro-Planner, a programming system written in 1971 (by Sussman, Charniak, and Winograd [Sussman 71], inspired by the work of Hewitt [Hewitt 71]), which anticipated many of the features of Duck and other "logic programming" systems.

Duck is the offspring of an offspring of Micro-Planner, the Conniver system of [Sussman 72, McDermott 74]. The first version was implemented at Yale in 1976, when I was looking for relaxation after an arduous thesis. Every part of it has since been rewritten at least once. It has been used off and on since 1977 as a teaching tool for graduate students in Computer Science. Starting in 1981, it was turned into a practical system for research at Yale. It is also a commercial product owned by Yale, and sold through Smart Systems Technology. It is available

without charge to universities and government laboratories.

2. THE DUCK DATABASE

Duck implements a relational database similar to those found in AI languages such as PLANNER and Prolog. [Hewitt 71, Warren 77] This kind of database allows efficient implementation of procedurally controlled deduction. There are two ways to approach the Duck data structure: as a generalization of Lisp property lists, or as an implementation of predicate-calculus deduction. The trick is to make it work both ways.

2.1. Storing and Retrieving Information

Lisp has a very useful mechanism for representing binary relations, namely, the property list that hangs on every atom. In Nisp, the function PROP is used to access this value. If John is believed by a program to be Jane's father, then the value of

```
(PROP 'FATHER 'JANE)
```

should be JOHN. This could be set up using the := function, thus:

```
(:= (PROP 'FATHER 'JANE) 'JOHN)
```

(PROP is called GET in many Lisp dialects, with the opposite order of arguments. The magic word := is a generalized SETQ; (:= *e1 e2*) alters the value of *e1* to be the same as *e2*. In most MacLisp descendants, SETF plays the same role. Note that in Zeta and Common Nisp, the colon must be slashified, so the function has the print name /:= or \:=.)

We can retrieve Jane's father by saying (PROP 'FATHER 'JANE). But what if we wanted to know who John's child was? We would have to have written (:= (PROP 'CHILD 'JOHN) 'JANE) as well, so as to be able to evaluate (PROP 'CHILD 'JOHN).

A database does all this bookkeeping for you. It allows you to focus on the *relationship* "John is the father of Jane." Relationships are represented as list structures, with the relation name, or *predicate*, first: (FATHER JOHN JANE). The user does not have to associate the atoms appearing in the structure with each other. He merely writes (ADD '(FATHER JOHN JANE)), and the relationship is noted in the database. Here FATHER is the predicate, and JOHN and JANE are its *arguments*.

Relationships stored in the database are called *assertions*. In the "blocks" world, there are assertions like (ON A B) and (CLEARTOP B); conceptual dependency [Schank 72] might be represented by something like (DO JOHN (PTRANS JOHN PLACE27 PLACE103)). An assertion is added to the database by saying (ADD *formula*), which is analogous to PUTPROP; and removed by saying (ERASE *formula*), which is analogous to REMPROP. (You can re-initialize the database -- erase everything you ever added -- by executing (DB-CLEAR).)

The analogue to PROP is (ASSERTION-FETCH *pattern*). A pattern is like the formula of an assertion, but the *variables* in it, marked by "?", represent slots to be filled by *matching* the

pattern against assertions in the database. For example, instead of writing (PROP 'FATHER 'JANE), just say (ASSERTION-FETCH '(FATHER ?X JANE)). The result, like that of PROP, is () if there are no assertions that look like the pattern in the database. If there are such, a list of them is returned. However, the elements of this list specify more than the returned items; they also specify, for each assertion, what pieces of it each variable matched. So, after executing (ADD '(FATHER JOHN JANE)), executing

```
(ASSERTION-FETCH '(FATHER ?X JANE))
```

returns

```
({ans ((X JOHN)) FATHER1}).
```

This is a list whose only element is an object called an *ans* (for "answer"). (It will look somewhat different in each dialect of Duck.) This peculiar object can then be used by writing ?(X *ans*) to actually see who is Jane's father. A typical program would go:

```
(COND ((:= A (ASSERTION-FETCH '(FATHER ?X JANE)))
       (TTYMSG "Jane's father is " ?(X (CAR A))))
      (T (TTYMSG "Jane's father is unknown"))   )
```

This is slightly more painful than using PROP, but we are amply repaid by the fact that we can also retrieve John's children with no other data structure, by executing (ASSERTION-FETCH '(FATHER JOHN ?X)). Furthermore, Duck supplies many macros for accessing anses smoothly. To print John's children out, we could write:

```
(FOR-EACH-ANS (ASSERTION-FETCH '(FATHER JOHN ?X))
   (TTYMSG ?X " is John's child")    )
```

The */(FOR-EACH-ANS *list-of-anses -body-*) construct is a handy abbreviation. It successively binds a hidden Lisp variable to each ans in the list. Within the body of a FOR-EACH-ANS, this variable does not need to be mentioned explicitly. "?X" refers to the value in the current ans. The "hidden" Lisp variable holding the current ans is ANS*.

So "?" has two purposes: in a fetch pattern, it specifies slots to be filled in by the retrieval mechanism; in Lisp code, it refers to the values of a variable from the "current ans," ultimately derived from a fetch pattern that mentioned that variable. There is one more purpose for "?," which I will describe later.

Note that nothing prevents us from having two or more question-marked variables in a fetch pattern. To print all known fatherhoods, we could execute:

```
(FOR-EACH-ANS (ASSERTION-FETCH '(FATHER ?MAN ?KID))
   (TTYMSG ?MAN " is the father of " ?KID)   )
```

Anses such as those returned by ASSERTION-FETCH consist of two parts: a list of variable bindings, and the name of the assertion that was found in the database and used to generate those bindings. As I will explain in Section 6.3, every assertion has a name. FATHER1 is the name

of the assertion (FATHER JOHN JANE). The name is actually of the form FATHER*n*, where *n* is chosen by Duck when the assertion is created.

Later, we will generalize the support half of an ans, by allowing, instead of a simple assertion name, a complex *proof record* that stores all of the assertions that were used to arrive at the variables bindings. Such generalized anses are returned by a generalization of ASSERTION-FETCH called FETCH, explained below.

The variable-binding lists in anses are in the form ((*var val*) (*var val*) ...). Another word for this kind of list is an "association list," or "substitution." The latter name comes from the fact that the association list specifies values for the fetch-pattern variables, such that substituting the values for the variables in the pattern would make the pattern equal to the retrieved assertion. In our example, substituting JOHN for ?X would make (FATHER ?X JANE) equal (FATHER JOHN JANE), the retrieved assertion. We will extend this idea later, in our discussion of unification.

In summary, here are the Lisp functions for manipulating the Duck database:

- (ADD *formula*): Creates an assertion with the given formula if one does not exist already. As explained in Section 6.1, it supports this formula with a justification from the current ans. As explained in Section 2.2, it initiates forward chaining from this formula if it is new.
- (ERASE *formula*): Erases the assertion with this formula from the database. As explained in Section 6.1, it also erases formulas deduced from this one. If the current datapool DP* is non-(), the formula will be erased from the local datapool only. (See Section 6.2.)
- (DB-CLEAR): Restores the database to its original state, erasing all assertions made by the user.
- (ASSERTION-FETCH *pattern*): Returns a list of anses, one for each assertion in the database that matches the pattern.

In Section 4 I will explain FETCH, the generalized version of ASSERTION-FETCH that attempts to deduce anses as well as generate them from explicit assertions in the database. From now on, my examples will usually be in terms of FETCH rather than ASSERTION-FETCH, to avoid giving the impression to advanced re-readers that the examples work only for the restricted function. For those reading the manual for the first time, just assume that FETCH does what ASSERTION-FETCH does, and don't worry about the complexities to come.

2.2. Deduction by Forward Chaining

The strength of an assertional database is that it provides a flexible environment for doing deductions. Consider the following code:

```
(FOR-EACH-ANS (ASSERTION-FETCH '(IS ?X HUMAN))
   (ADD '(MORTAL ?X))   )
```

This program ADDs to the database an assertion of the form (MORTAL x) for each expression x found in an assertion of the form (IS x HUMAN). The argument to ADD is quoted, so it might appear that just one assertion, literally, (MORTAL ?X), would be ADDed, but ADD substitutes into its argument the value of ?X from the current ans. ADD is said to "varsubst" its argument. (FETCH, ASSERTION-FETCH, and ERASE do the same thing.)

This is a kind of deduction. It depends for its cogency on the truth of "All men are mortal." (This fact may be called its "logical content.") This code is, however, subject to a kind of "timing error." If a new assertion of the form (IS xx HUMAN) comes into the database after the program is executed, then (MORTAL xx) will never be asserted.

An alternative that avoids this problem is to drop the code above, and simply ADD the *implication*

```
(-> (IS ?X HUMAN) (MORTAL ?X))
```

to the database. This assertion has two kinds of meaning: logically, it simply means that (IS ?X HUMAN) implies (IS ?X MORTAL); procedurally, it means that whenever (IS ?X HUMAN) is added to the database, (MORTAL ?X) is to be asserted immediately. If (IS SOCRATES HUMAN) is present before the implication is added, then (MORTAL SOCRATES) is added then. If later (IS CAESAR HUMAN) comes in, then (MORTAL CAESAR) gets added. All of this is done automatically by ADD. The process is called *forward chaining*.

Besides the fact that it is an implication, (-> (IS ?X HUMAN) (MORTAL ?X)) differs from previous assertions in another respect: It contains a variable, namely ?X. A variable in an assertion is given a *universal* interpretation. The implication means, "For *all* X, if X is human, then X is mortal." In more traditional logical notation, we would prefix the formula with an upside-down "A," instead of marking each occurrence of a variable with a question mark.

So now we have three meanings of a "?" variable, or *match variable*:

1. In a fetch pattern, it represents a slot to be filled. If any anses are found, they will specify a binding for this variable.
2. In Lisp code, it refers to the value of a variable in an ans. ?X means the value of X in the current ans; usually used inside a FOR-EACH-ANS or one of its related macros. ?(X a) means the value of X in ans a.
3. In an assertion, it stands for an arbitrary object. That is, it is a universally quantified variable.

We use the same notation for fetch patterns and assertions because ADD uses the same kind of pattern matching for forward chaining that FETCH does with assertions. When (IS SOCRATES HUMAN) is asserted, it is matched against (IS ?X HUMAN), the *antecedent* of the implication, just

as it would have been had (IS ?X HUMAN) been a FETCH pattern. The variable ?X matches the symbol SOCRATES. This symbol is then substituted for SOCRATES in (MORTAL ?X), the *consequent* of the implication, giving (MORTAL SOCRATES), the desired conclusion. The form of matching used here is called *unification*. We will encounter it again in the next chapter.

Forward-chaining rules resemble production rules such as those of Ops5. [Forgy 82] There is an important difference. The conclusion of a forward-chaining rule must follow from the antecedent. The conclusion is always a new formula to be added to the database, not an arbitrary action to be taken. This new formula is tagged with a *data dependency note* indicating what it was derived from. If the assertion that triggered it (in the example, (IS SOCRATES HUMAN)) is erased, then the conclusion will be erased, too. Something similar happens in the other approach to the same deduction:

```
(FOR-EACH-ANS (FETCH '(IS ?X HUMAN))
   (ADD '(MORTAL ?X))   )
```

When ADD is called, the assertion it adds to the database is marked as supported by the assertions in the proof record of the current ans. As before, if those supporting assertions are erased, so will the newly added one.

It is possible to override all or part of these conventions, so that the user can control exactly what supports what, or turn off the machinery altogether. He can even simulate a production system if he likes. However, my recommendation when one finds oneself tempted in this way is to rethink the problem. A solution that makes use of real deduction will have several benefits, including semi-automatic database update, and the use of Duck utilities to explain deductions, as discussed in Chapter 7.

3. DUCK AS A PROGRAMMING LANGUAGE

3.1. Goals and Nondeterminism

We are going to shift gears at this point and separate Duck from Lisp. When Duck is started, you are talking to a Lisp. If you type (DUCK), then the system enters a special sort of "listen loop" that might look like this:

```
(DUCK)
?>(FATHER ?X JANE)
(FATHER JOHN JANE)
X=JOHN
?>
```

You can think of this loop as FETCHing what you type, and printing the result in a pleasing way (first the instance found, then the variable bindings, one per line). However, for now let us forget FETCH altogether, and think of what is typed as "Duck programs." The elementary step of such a program is a *goal*, in this case (FATHER ?X JANE). One way a goal is *satisfied* is to encounter a matching assertion in the database, as in this simple example.

To make this seem more interesting, we can generalize the idea of Duck program to include *conjunctions* of the form

(AND $goal_1$ $goal_2$ ... $goal_n$)

For instance, if you type

```
(AND (BIG ?X) (GREEN ?X) (RUTABAGA ?X))
```

then Duck will try to find an assertion (BIG *thing*) in the database. If it finds one, then it will try the resulting value of of ?X in the other two goals. If there are three assertions,

```
(BIG THING9)      (GREEN THING9)      (RUTABAGA THING9)
```

then the program would set ?X to THING9, and this value would be printed. Duck has found a single object that qualifies as a big green rutabaga.

One obstacle to this tidy scenario is that the database might also contain these assertions:

```
(BIG THING80)      (GREEN THING80)
```

Duck might happen to guess that the correct answer to the first goal was ?X=THING80, and this answer would work on the second goal. But the third goal, which would have become (RUTABAGA THING80), would find no assertion in the database. Such a goal is said to *fail*. In general, whenever Duck has a choice of alternative answers to one goal, it has no way of knowing which choices will lead to the failure of a goal later. Hence it must pick one of the choices, but be prepared to undo that choice and make another one later.

What this means is that Duck is a *nondeterministic programming language*. A conjunction

(AND $goal_1$ $goal_2$ ... $goal_n$)

can be thought of as a series of steps, of the traditional sort, but with the difference that a goal failure will cause previous choices to be retried, until either success is achieved or all choices lead to failure (unless an infinite number of choices is available, in which case Duck will try to run forever).

3.2. Backward Chaining and Unification

So far Duck is a pretty boring programming language, with no loops or subroutine calls. We will rectify this with the following extension. Once the assertions that unify with a goal have been exhausted (or if none work to begin with), Duck looks for *backward rules* of the form

(<- *goal antecedent*)

For instance, suppose we had a goal (SUPPLIER J-C-NICKLES ?X), and no assertions that matched it. We might have a rule

```
(<- (SUPPLIER ?R ?S)
    (AND (RETAILS ?R ?P) (MANUFACTURES ?S ?P)))
```

Such a rule can be interpreted in two ways. Its *declarative interpretation* is as an implication: "For all retailers ?R, products ?P, and manufacturers ?S, if ?S manufactures something (?P) that ?R sells, then ?S is one of ?R's suppliers."

The *procedural interpretation* is as a subroutine that Duck can use on the goal (SUPPLIER J-C-NICKLES ?X), or any other goal whose predicate is SUPPLIER. This is the interpretation we shall focus on. To make the call happen, the *goal*, or *consequent*, part of the rule must be unified with the actual goal. The unification required is more general than what we have seen before, because it involves matching two patterns each of which contains variables. The result is two sets of variable bindings:

```
{?X = ?S*}          ; the variables from the goal
{?R = J-C-NICKLES}  ; the variables from the rule
```

The asterisk in the first set is to remind you that ?S is a variable from the other set.

These sets of bindings are used twice, when the procedure is called and when it returns. To call it, the rule variables are substituted into the rule antecedent, giving this:

```
(AND (RETAILS J-C-NICKLES ?P) (MANUFACTURES ?S ?P))
```

This is a Duck program, in the sense described above. As usual, it is nondeterministic, since a hunt may be required for a value of ?P that satisfied both goals. It may even require calling further procedures, but let's suppose that it does not in this case, and that these two assertions are found in the database:

```
(RETAILS J-C-NICKLES AIDAS)
(MANUFACTURES SOPRANO-SNEAKERS AIDAS)
```

The program will succeed with the new values

```
{?S = SOPRANO-SNEAKERS,  ?P = AIDAS}
```

Now the other list of variable bindings comes into play. Recall that the goal-variable list was:

```
{?X = ?S*}
```

Duck plugs the new value for ?S, SOPRANO-SNEAKERS, into this list, generating the answer ?X=SOPRANO-SNEAKERS to the original goal, "What ?X is a supplier of J-C-NICKLES?"

If the original goal had occurred as part of a larger Duck program, then execution would have proceeded with the next goal just as before. If a later goal failed, then Duck would have gone back for another answer to the SUPPLIER goal, which would have required reactivating the execution of the body of the procedure. This is called "failing back into" the procedure. If this procedure called other procedures, goal failure would continue to propagate down to those, until a redoable choice was reached. The key is that the *most recent* choice is undone, no matter which procedure had control at the time. This style of control structure is called *backtracking*.

It takes a bit of study to understand how backtracking works. The process for generating answers to a goal can be suspended in any state. In particular, the suspension must record which rules have been exhausted for that goal, and which are still in reserve. For example, take the goal (P ?X), which matched (say) two non-rules and three rules:

```
1 (P A)
2 (P B)
3 (<- (P ?X) (Q ?X))
4 (<- (P ?X) (AND (R1 ?X) (R2 ?X)))
5 (<- (P ?X) (AND (S1 ?X ?Y) (S2 ?Y)))
```

A frozen state of the P-generating process might record that assertions 1 through 3 have been used up, that assertion 5 has not been touched yet, and that assertion 4 has generated two answers, and might be able to generate another, by reviving the process for generating answers to (R1 ?X). The frozen state of this subprocess will of course record very similar information about the subgoal. In fact, a frozen goal-generation process can be arbitrarily large. Usually, however, the storage needed to store it is not as important an issue as the time required to run it.

The choices recorded in frozen goal-generation process descriptions are of which assertion to use on a goal or subgoal. Unlike some systems, Duck rules are unordered, except that assertions that solve the goal immediately are used before assertions that lead to backward chaining. That is, in the previous example, the two assertions (P A) and (P B) will be used before the three backward rules, but other than this the order of answer generation is unpredictable (and may well change from one execution to another). I will use the term *implication* or *rule* for an assertion starting with <- or ->. Assertions of the form (-> ...) are called *forward rules*; of the form (<- ...), *backward rules*. Non-rules are called *literal assertions*. Since goals don't

normally start with <- or ->, the ordering policy may be simply stated as, "Literals are used first, then backward rules, in some random order." (Nothing prevents a goal from being of the form (<- ...), in which case, after matching assertions are tried, an attempt will be made to find backward rules of the form (<- (<- ...) ...), probably without success.)

This rule-ordering policy encourages modular rules; each rule must be correct in isolation. If you do need to enforce ordering, in order to do "logic programming," use THCOND; see Section 3.6.

Duck's subroutine-calling mechanism is somewhat different from that of traditional languages (but identical to that of Prolog [Warren 77]). No value is returned; instead, the subroutine transmits all its information back through the variables that occur in its consequent. This transformation does not occur as a side effect in the usual sense. A variable can be given a value only if it does not already have one. This is all achieved through unification, which I will discuss more in the next section.

There are no constructs for looping or transferring control in Duck, except for this subroutine calling mechanism and backtracking. Backtracking automatically implements loops of the sort "Go through a set of objects until one is found with a certain property." Another kind of loop is of the form "Go through a set of objects applying some function to each of them, and collect the results." I will discuss these in Section 3.4.

3.3. More on Unification

It is impossible to appreciate deduction without understanding unification. So far we have looked at a variable matching a constant (when the variable gets bound to that constant), and a variable matching another variable (when the goal variable eventually gets the rule variable's value, when it is found).

To see some of the other possibilities, consider what happens when the database contains the following assertion:

```
(LOVES  ?X (SPOUSE  ?X))
```

which means, "Everyone loves his spouse." LOVES is a predicate, but SPOUSE is a *function*. In Duck, a function is not something that gets called and returns a value. Instead, it simply constructs terms that denote things; (SPOUSE *person*) denotes the spouse of *person*. These terms can then be used as arguments to other functions and predicates.

A term like (SPOUSE POPE-JOHN-PAUL) or (SPOUSE BROOKLYN-BRIDGE) is meaningless. So the rule should really be

```
(<- (LOVES  ?X (SPOUSE  ?X)) ; Everyone loves his spouse
    (MARRIED ?X))            ; if he is married
```

For simplicity, we will neglect this complication.

You may be surprised to find variables in a literal assertion, since so far we have had them only in goals and rules. But recall that a variable in the database is universally quantified. This makes sense even for a literal assertion (i.e., a non-rule). In fact, such an assertion can be thought of as a rule without a body.

Not only does our assertion have variables, but it has a repeated variable. When this occurs, unification will succeed only if the variable matches the same constant each time. Hence, the assertion will satisfy the goal (LOVES JANE (SPOUSE JANE)), but not the goal (LOVES JANE (SPOUSE SALLY)), because the variable ?X must match the same thing in both occurrences.

A trickier case is the goal (LOVES ?Y (SPOUSE FRED)); the substitution returned should be {?Y = FRED}. When ?Y and ?X are matched, one gets the other as value. It doesn't matter which -- suppose ?Y is bound to ?X. Then when FRED and ?X are matched, ?X is bound to FRED. So we have this situation:

?Y has as value ?X which has as value FRED

This kind of situation arises all the time. With unification, the notion of value is "transparent"; if variable V_2 occurs as part or all of the value of V_1, then when V_2 gets a value, that value automatically replaces V_2 in the expression that is the value of V_1. Hence the situation described above is just the same as ?Y and ?X both having the value FRED.

Alternatively, when ?Y and ?X are matched, ?X could be bound to ?Y. Then when ?X is matched to FRED, the unifier will see that ?X is already bound, and hence that its previous value must unify with the new one. So it will unify ?Y and FRED. As before, ?X and ?Y get the same value.

Finally, if you prefer, a third way to think of the process is in terms of outcome. Unification means finding variable bindings that make the two patterns equal. The only one that works is ?X = ?Y = FRED.

With this understood, let's look at a slightly trickier case, involving the goal (LOVES ?Y (SPOUSE ?Z)). By the same sort of reasoning as before, we will have ?Y, ?X, and ?Z constrained to be the same, but this time not to be any particular value. Unification handles this by binding ?Y to ?Z, or ?Z to ?Y, or both to some new variable. (When Duck makes up a new variable, it gives it a name of the form ?_*number*.)

Here is why this might be necessary. Suppose we execute the Duck program

```
(AND (LOVES ?Y (SPOUSE ?Z))
     (OCCUPATION ?Y MILKMAN)
     (OCCUPATION ?Z TRAVELING-SALESMAN))
```

If the first goal succeeds using the "Everyone loves his own spouse" rule, then ?Y and ?Z will be constrained to be the same. Hence if the second goal succeeds, the third will fail (unless some

milkman is moonlighting as a traveling salesman).

One thing to emphasize about unification is that it keeps variables of the same name from different sides of the match separate. In the previous example, I should have carefully separated ?Y and ?X, saying, e.g., that ?Y was bound to ?X*, where, as before, the asterisk signifies that ?X came from the other pattern. This was unnecessary in this case, but bear in mind that the goal could have been (LOVES ?X (SPOUSE FRED)) without changing the outcome. In this case the variable bindings produced would have been either

```
{X = ?X*}          or      {X = FRED}
{X = FRED}                 {X=?X*}
```

As one last example, unifying

```
          (LOVES (SPOUSE ?Y) (SPOUSE ?Z))
  and     (LOVES ?X (SPOUSE ?X))
```

gives the result

```
{?Z = ?X* or, fully expanded, (SPOUSE ?Y)}
{?X = (SPOUSE ?Y)*}.
```

Applying these substitutions makes both patterns equal to (LOVES (SPOUSE ?Y) (SPOUSE (SPOUSE ?Y))). This makes sense, because the query (LOVES (SPOUSE ?Y) (SPOUSE ?Z)) may be thought of as asking, "Which two spouses love each other?" The answer is, "Anyone's spouse loves his spouse's spouse." ((!) It helps in visualizing this to know that you are your spouse's spouse, but the program doesn't know that.)

3.4. List Processing in Duck

Although it was convenient, the SPOUSE function introduced in the last section is not plausible as a predicate-calculus function. That's because a term like (SPOUSE SALLY) may denote someone with another name, like FRED. In general, multiple names for the same individual should be avoided, because Duck has no way of realizing that, for instance, (OCCUPATION FRED MILKMAN) and (OCCUPATION (SPOUSE SALLY) MILKMAN) say the same thing, and hence if one is the database when the other occurs as a goal, that goal should succeed.

Functions are more useful when used to implement data structures. The most useful data structure is the list, or *tuple*, implemented using the function TUP. TUP is unusual in that it takes any number of arguments, as in (TUP FRED SALLY EIFFEL-TOWER). This term denotes the list of three elements whose first element is FRED, second is SALLY, and third is the EIFFEL-TOWER. Unlike the case of SPOUSE, there is no other name for this list.

Because tuples are so useful, there is a special reader syntax for them. (TUP *-elements-*) may

be typed in as !< *-elements-* >. The empty tuple is !<>, or (TUP).

Tuples are decomposed using unification instead of CAR and CDR, as in Lisp. An extension of unification is useful here. A *segment variable* is a pattern of the form (SEG *var*), usually abbreviated !?*pat*. A segment variable unifies with any *sequence* of elements of a tuple. E.g.,

```
(TABLE !<?X !?S>)
```

unfies with

```
(TABLE !<FRED SALLY EIFFEL-TOWER>)
```

with bindings {X=FRED, S=!<SALLY EIFFEL-TOWER>}.[2]

Segments may occur anywhere in a tuple, not just at the end. This complicates the unifier somewhat, because two patterns may unify more than one way. For instance, if (P !<A B C>) is in the database, (ASSERTION-FETCH '(P !<!?S1 ?X !?S2>)) returns three substitutions:

```
{?S1 = !<>, ?X = A, ?S2 = !<A B C>}
{?S1 = !<A>, ?X = B, ?S2 = !<C>}
{?S1 = !<A B>, ?X = C, ?S2 = !<>}
```

This feature means that rules for appending tuples or testing membership in tuples are usually unnecessary. For example, to find two tuples ?X and ?Y such that ?X contains B and concatenating ?X and ?Y gives !<A B C>, the following Duck program suffices:

```
(AND (:= !<!?X !?Y> !<A B C>)
     (:= ?X !<!?X1 B !?X2>))
```

(:= *x y*) is a special predicate that succeeds if *x* can be unified with *y*, and passes on the result. The first goal unifies four ways. The first two fail on the second goal; the second two do not, setting ?X to !<A B> and !<A B C>, respectively.

I should point out that the presence of arbitrary segments makes it almost impossible for a unifier to be complete. For instance, the unification of

```
    (P ?X !<!?X A>)
and (P ?Y !<A !?Y>)
```

yields an infinite number of unifiers, with ?X = !<>, ?X = !<A>, ?X = !<A A>, and so on. Duck does not attempt to deal with this problem, but instead does the best it can. In this example, it would stop after finding the first two answers.

List processing algorithms in Duck often look similar to their Lisp counterparts. Suppose in the SUPPLIERS example before, we wanted to take a tuple of retailers and return a tuple of all

[2]In older editions of Duck, one wrote !&?*var* instead of !?*var*, and this notation is still permitted. In general, !&*term* is allowed, and expands into (SEG *term*). But this generality is almost completely useless.

their suppliers (not worrying about duplication). Here are rules to do the job:

```
(ALL-SUPPLIERS !<> !<>)
   ; If there are no retailers, there are no suppliers

(<- (ALL-SUPPLIERS !<?R !?REST>
                   !<!?R-SUPPLIERS !?REST-SUPPLIERS>)
   ; Concatenating the suppliers of ?R to the
   ; other suppliers gives the complete list
    (AND (THFIND ?R-SUPPLIERS ?S (SUPPLIER ?R ?S))
         ; ?R-SUPPLIERS is the tuple of all suppliers of ?R
         (ALL-SUPPLIERS ?REST ?REST-SUPPLIERS)))
         ; and ?REST-SUPPLIERS contains all the others
```

These rules use THFIND, described in Section 3.6, to collect a tuple ?R-SUPPLIERS of all ?S such that (SUPPLIER ?R ?S). Note that the program is recursive, the basis case being the empty tuple. If it is not empty, then the suppliers ?R-SUPPLIERS of the first retailer are combined with the suppliers REST-SUPPLIERS of all the other retailers to give the answer. The main difference between Lisp and Duck is that Duck uses unification both to decompose the input (the list of all retailers) into first element and remainder, and to combine the outputs into an overall output list of suppliers.

For more on list processing in Duck, see Section 5.1.

3.5. Defining Predicates and Rules

Duck is used by initializing the database with some rules (implications) for the user's domain, and then doing forward and backward chaining, usually interfaced to some Lisp code. One could just use ADD to get all the implications into the database; they are in fact stored like other assertions. But then the system will make up ugly names for them, like <-103. In addition, when bugs are discovered in a rule, the buggy version will have to be carefully ERASEd; just ADDing the new version will not eliminate the old.

A better idea is to use the special form RULE:

*/(RULE *name formula*)

creates an assertion with the given *name*. If the same *name* is defined twice, the old definition is removed. The formula will often be an implication, but it doesn't have to be (and so perhaps the name RULE is a trifle misleading).

IMPORTANT: RULE is a *Lisp* special form, not a Duck predicate. When Duck is started, you are actually talking to a Lisp read-eval-print loop. Even after executing (DUCK), Lisp forms will still be handled by Lisp; when files are loaded, they are supposed to contain Lisp forms, not Duck. Given the ubiquity of Lisp in Duck, it seems simplest to let all the constructs for *defining* Duck predicates and rules be Lisp special forms. It is only when these predicates and rules are

being *used* that you should think of control as being "in Duck."

Here is how RULE would be used for the example in the last section:

```
(RULE NO-SUPPLIERS (ALL-SUPPLIERS !<> !<>))
   ; If there are no retailers, there are no suppliers

(RULE SOME-SUPPLIERS
   (<- (ALL-SUPPLIERS !<?R !?REST>
                      !<!?R-SUPPLIERS !?REST-SUPPLIERS>)
      ; Concatenating the suppliers of ?R to the
      ; other suppliers gives the complete list
       (AND (THFIND ?R-SUPPLIERS ?S (SUPPLIER ?R ?S))
            ; ?R-SUPPLIERS is the tuple of all suppliers of ?R
            (ALL-SUPPLIERS ?REST ?REST-SUPPLIERS))))
            ; and ?REST-SUPPLIERS contains all the others
```

RULE will Skolemize its argument for you. That is, if there are any quantifiers present, they will be taken out in the correct way. So

```
(RULE HAPPINESS (FORALL (P)
                  (<- (HAPPY P)
                      (EXISTS (S)
                         (AND (MARRIED S P)
                              (GOOD S))))))
```

("Having a good spouse means you are happy") is actually entered into the database in the form

```
(<- (HAPPY ?P)
    (AND (MARRIED ?S ?P) (GOOD ?S)))
```

Another important service Duck will perform is checking the syntax of your assertions and fetch patterns. If the Lisp variable SYNCHK* is T, then it will check for the right number of arguments, and the right types of arguments. Any predicate it doesn't know about will cause a complaint.

To define types and declare objects to be of those types, use DEFDUCKTYPE and DUCLARE:

1. */(DEFDUCKTYPE *typename super*): makes typename a type. (DEFDUCKTYPE block obj) would make block be a type (with supertype obj, the general class of "objects"). I use the Nisp convention that types are in lower case, but this is not necessary. (In Franz and other Lisps in which lower case is the default, types are usually upper case.)
2. */(DUCLARE *-types-and-symbols-*): declares each symbol to be of the type given by the nearest preceding type.

```
(DUCLARE block FOO BAZ
         (FUN block (obj (LST block))) BL)
```

declares FOO and BAZ to be of type block, and BL to be a function that takes an object and a tuple (LST) of blocks, and denotes a block. That is, (BL *e1 e2*) will be well-formed only if *e1* is of type obj and *e2* is of type (LST block). If the arguments to DUCLARE end with a type, then each symbol is declared to be of the nearest type *following* it, as in

```
(DUCLARE FOO BAZ block
         BL (FUN block (obj (LST block))))
```

A legal type is one of these things:

1. A built-in type, one of boolean, obj, number, fixnum, flonum, symbol, or prop. If you don't care about the type of something, make it an obj. A prop is a proposition. Normally, you don't declare something to be a prop, but use DEFPRED, discussed below.
2. A symbol defined with DEFDUCKTYPE.
3. An expression of the form (LST *type*). Tuples are automatically of this type.
4. An expression of the form (FUN *resulttype* (*-argtypes-*)). The spouse function could have been declared with (DUCLARE SPOUSE (FUN person (person))), assuming person had been defined, with DEFDUCKTYPE.
5. An expression of the form (*fun*), where *fun* has been defined with an "untyped DEFFUN" (see below).

The DEFFUN and DEFPRED constructs obviate most uses of DUCLARE.

(DEFFUN *denotype* (*fun -types-and-args-*)
[(=: *-English-skeleton-*)])

declares the *fun* to be a predicate-calculus function that takes arguments of the specified types, constructing terms of type *denotype*. (The optional *English-skeleton* is described in Section 7.2.) For example, SPOUSE could more tidily be declared thus:

```
(DEFFUN person (SPOUSE person ?P))
```

The *types-and-args* are interspersed type designators and question-marked variables. The variables serve mainly as placeholders. The function has as many arguments as variables, and each is of the nearest preceding type. For instance,

```
(DEFFUN (LST person) (CHILDREN person ?HUSBAND ?WIFE))
```

defines CHILDREN to be of type

```
(FUN (LST person) (person person))
```

(The variables' actual names become important if an English skeleton is supplied; see Section 7.2.)

It is often a nuisance to have to define a type denoted by just one function. Functions in the predicate calculus are usually used to define "record structures." (TUP is just the most common example.) For instance, one might want to have a function (EMPREC *name salary* ...) for describing employees. We could have a data type employee-record, and describe the function

and the type this way:

```
(DEFDUCKTYPE employee-record obj)
(DEFFUN employee-record (EMPREC person ?NAME number ?SALARY ...))
```

and use them like this:

```
(DEFFUN ... (FOO ... employee-record ?E ...))
```

but it is a nuisance to have to make up this type, which has no other purpose except to be the sort of thing that EMPREC denotes. To avoid this glitch, one can omit the type in a DEFFUN, and a special type will be defined automatically, whose designator is the function name in parens. The previous example could have been written like this:

```
(DEFFUN (EMPREC person ?NAME number ?SALARY ...))
```

and use them like this:

```
(DEFFUN ... (FOO ... (EMPREC) ?E ...))
```

Here (EMPREC) is a type, to wit, "things denoted by terms with function EMPREC."

DEFPRED can be used to define predicates (which are just functions of type (FUN prop (...)) as far as the type system is concerned).

```
*/(DEFPRED (predicate -types-and-args-)
    [(=: -English-skeleton)]
    -rules-)
```

declares the *predicate* to be of type (FUN prop (*-types-*)), where the *types* are derived as described under DEFFUN; and defines all the *rules*, which should involve that predicate. (See Section 7.2 for a description of the optional *English-skeleton*.) For example, the predicate (APPEND *x y z*), which is true of three tuples *x*, *y*, and *z* if *x* concatenated to *y* yields *z*, might be defined this way:

```
(DEFPRED (APPEND (LST obj) ?X ?Y ?Z)
   (APPEND !<> ?X ?X)

   (<- (APPEND !<?A !?R> ?Y !<?A !?Z>)
       (APPEND ?R ?Y ?Z))   )
```

This form declares APPEND to be of type (FUN prop ((LST obj) (LST obj) (LST obj))), that is, a predicate on three tuples of arbitrary objects. It then gives two rules defining APPEND. They will have names APPEND-DEF-1 and APPEND-DEF-2. You can define other rules involving APPEND; just avoid names of the form APPEND-DEF-*n*. As usual, all the rules will be unordered, except that literal assertions are used before implications.

Inside a DEFPRED, a rule of the form (<- *c a*), where *c* is *exactly the same* as the header (without the types), and *a* does not start with the predicate being defined, may be abbreviated *a*. An example would be

```
(DEFPRED (RES node ?N1 ?N2)
   (OR (RESISTOR ?R ?N1 ?N2)
       (RESISTOR ?R ?N2 ?N1))   )
```

which means the same as

```
(DEFPRED (RES node ?N1 ?N2)
   (<- (RES ?N1 ?N2)
       (OR (RESISTOR ?R ?N1 ?N2)
           (RESISTOR ?R ?N2 ?N1)))   )
```

DEFPRED may also be used to define lisprules, which I will talk about in Chapter 4.

There are two different ways to create Duck systems. Which you prefer depends on your operating system and on your personal taste. In the "textfile" approach, you build up a text file containing DEFDUCKTYPEs, DUCLAREs, DEFPREDs, RULEs, and so forth, defining all the concepts in your domain. As you encounter bugs, you go back to the text file, make changes, and re-evaluate relevant sections of it. (Exactly how this is done depends on the operating system.) In the "in-core" approach, the user edits objects as Lisp data structures in core, periodically saving them in text files. Some operating systems do not allow easy visits to text files, and some users just prefer the in-core approach.

The in-core approach is based on the in-core editor and workfile manager. These are documented in more detail, but a section on the workfile manager appears in Section 7.4 clarify those of its features specific to Duck.

If you are using the textfile approach, you must make sure that in the file you build all symbols are declared before being used. This applies to predicates as well as other symbols. The simplest way to avoid undeclared predicates is to put *two* DEFPREDs in the file for each predicate, one at the front with no rules, the other repeating the declaration and defining some rules.

Along the route to a bug-free file, you are likely to encounter complaints from Duck about undeclared symbols. It will stop and ask you to define the symbol and type OK, or, if it was mistyped, type the symbol you really meant. If you do the latter, you must of course make the correction in the text file as well. If you want to abort the whole syntax check, type \\. Exactly how much abortion is caused depends on context. If the syntax error was found in a rule definition, then typing \\ will cause the rule to be defined with illegal syntax, in the expectation that you will soon correct it. In a (DUCK) loop, it will cause the goal to be discarded. In either context, typing GO will cause just this one syntax error to be overlooked. Typing CXT will cause the context of the erroneous item to be displayed. Typing NEW will put you in the *type editor*, which will allow you to define or redefine any symbol or type. This is described in Section 7.5. If you simply want to declare the symbol, type its type; in the special case where it is a predicate, type a list of its argument types to declare it and avoid future questions about its type. (As before, you must remember to put an equivalent declaration in the file.)

An advertisement: Type checking may seem like a nuisance. But I have found that it is almost indispensable in a logic-based system, because seemingly minor typographical errors can cause formulas to be missed completely. For instance, if an argument is omitted in the consequent of a backward implication, the consequent will simply fail to match any goal pattern, and any goal depending on it will quietly fail. One can use the trace package (see below) to find such unexpected failures, but the syntax checker will find them for you automatically.

There are four grades of syntactical involvement, depending on the setting of global Lisp variable SYNCHK*:

1. (): Don't do any.
2. RULES (the default): Check RULEs and DEFPREDs as they are defined.
3. T: Check rules when they are defined, plus every argument to FETCH, ADD, or ERASE.
4. ALL: Check everything as for T. Furthermore, insist that every symbol be defined. In other cases, symbols used as constants do not have to be declared. That is, the system will complain if ON is undeclared in (ON A B), but will take your word for it for A and B. If SYNCHK* is ALL, however, A and B must be declared as well, a bit of a nuisance during debugging.

3.6. Built-In Predicates and Pseudo-Predicates

There are several built-in predicates that you may find convenient. Many of them are not predicates at all, but extend DUCK both syntactically and semantically beyond the orthodox predicate calculus. These "pseudo-predicates" are prefixed with the string "*/" when first described.

Most of these constructs are to be used in ***backward chaining only***. They make sense as goals, but would be meaningless if asserted. (If you do assert them, they will go into the database.) Where a construct does make sense when asserted, I say so.

(> *x y*), (< *x y*), (>= *x y*), (=< *x y*): The usual inequalities. A goal of this form succeeds if *x* and *y* are numbers, and the inequality is satisfied.

(EVAL *arithmetic-expression val*): If the *arithmetic expression* has no variables, succeeds if *val* unifies with its value. Typically *val* is a variable; (EVAL (+ 2 2) ?X) binds ?X to 4.

(DENOT *term tuple*): Succeeds if *term* is the "denotation" of *tuple*, that is, if *tuple* has as elements the function and arguments of *term*. For instance, (DENOT (FOO A) !<FOO A>). (This is the same as =.. in Prolog.) DENOT can be used to convert data structures to "programs" and back; this ability means that Duck is almost as convenient in this regard as Lisp (in which programs *are* data structures).

*/(CONSISTENT *p*): Succeeds if *p* cannot be proven false. In CONSISTENT, variables in *p* are

treated "existentially." That is, (CONSISTENT (NOT (= A ?X))) asks whether there is any ?X that might be unequal to A (and, since there is, it binds ?X to one of them, using a term like (SK X 67)). Any conclusion from a CONSISTENT will depend "negatively" on (NOT *p*), as explained in the section on data dependencies. That is, if something is asserted as a result of a CONSISTENT, as in

```
(FOR-EACH-ANS (FETCH '(CONSISTENT p))
   (ADD q))
```

then the conclusion *q* will be removed from the database if (NOT *p*) is ever asserted.

*/(THNOT *p*): Succeeds if *p* cannot be proven true. (THNOT *p*) is not quite the same as (CONSISTENT (NOT *p*)). It is implemented to be a lot cheaper and not as philosophically pure. In particular, it does *not* do anything with the variables of *p*. And it does *not* set up the data dependencies mentioned. For example, suppose you execute

```
(FOR-EACH-ANS (FETCH '(THNOT (FOO A ?X)))
    (ADD '(BAR A ?X))
```

vs.

```
(FOR-EACH-ANS (FETCH '(CONSISTENT (NOT (FOO A ?X))))
    (ADD '(BAR A ?X))   )
```

and suppose ?X has no value at the outset. The first example will succeed if (FOO A ?X) fails, i.e., no ?X can be found such that (FOO A ?X). The second succeeds if (FOO A SK67) fails, i.e., if there is some value of ?X for which the goal fails. The first would add (BAR A ?X); the second, (BAR A SK67). The second would note that if (NOT (FOO A SK67)) is ever added, (BAR A SK67) should be erased. The first does not do this. In general, the extra safeness of CONSISTENT is wasted, so use THNOT.

(= *exp1 exp2*): True if *exp1* and *exp2* denote the same object. A goal of this form succeeds if *exp1* and *exp2* can be unified, or if they can be proven equal by rules and literal assertions supplied by the user. An equality assertion may be asserted, since putting it in the database is entirely appropriate. (NOT (= *exp1 exp2*)) succeeds unless *exp1* and *exp2* can be proven equal. NOT-= is like THNOT in that it does not create a data dependency to catch a later assertion that *exp1* and *exp2* are equal after all.

(:= *exp1 exp2*): succeeds only if the two expressions are unifiable, and binds variables so as to make the two sides identical. If one is a variable, for instance, it gets the other as value. (NOT (:= *exp1 exp2*)) succeeds only if the two are not unifiable. If (= A B) is in the database, the goal (= A B) will succeed, while (:= A B) will fail. := is cheaper than =, and often what you mean anyway.

*/(ASSERTION *pat*): Like *pat* alone, but backward chaining is suppressed. That is, the answers are just those derived from assertions that are already in the database.

*/(SOME *pat*): behaves just like *pat* as a goal, but succeeds at most once. That is, if there is an answer to *pat*, (SOME *pat*) succeeds and binds variables the way *pat* would have, but if a failure propagates back to this goal, no new answer is generated. This is useful when you know there is at most one answer, and don't want to waste time looking for others, or when you know that a subsequent failure cannot be due to the choice of answer at this goal.

The following pseudo-predicates sweep through the database, collecting or otherwise processing any data matching a pattern. The note below (after the description of THFORALL) describes a problem with the data dependencies set up by these pseudo-predicates.

*/(TOTAL *sum exp pat*): Succeeds if *sum* is the sum of all known instances of *exp* such that *pat*. E.g., if (INCOME-SOURCE JOHN TIPS 5000) and (INCOME-SOURCE JOHN SALARY 10000), then (TOTAL ?SUM ?AMT (INCOME-SOURCE JOHN ?W ?AMT)) sets ?SUM to 15000.

*/(THFIND *tuple exp pat*): Succeeds if *tuple* is the tuple of all known instances of *exp* such that *pat*. E.g., if (LOVES JOHN MARY), (LOVES ZELDA FRANK), and (LOVES HELEN FRANK) are the only LOVES relationships in the database, then (THFIND ?X !<?B ?A> (LOVES ?A ?B)) succeeds with ?X = !<!<MARY JOHN> !<FRANK ZELDA> !<FRANK HELEN>>. THFIND does not bind the variables contained in *pat* and not in *exp*. E.g., in the same example, if the goal had been (THFIND ?X ?A (LOVES ?A ?B)), then ?X would have gotten the value !<JOHN ZELDA HELEN>. This behavior contrasts with that of setof in Prolog, which would have (*mutatis mutandis*) found *two* answers,

```
    ?X = !<JOHN>          ?B = MARY
and
    ?X = !<HELEN ZELDA>   ?B = FRANK
```

*/(MAX *m exp pat*) and */(MIN *m exp pat*): Succeed if *m* is the maximum or minimum *exp* such that *pat*. The variables in *exp* and *pat* are set by the instance found. E.g., in the example above, (MAX ?I ?AMT (INCOME-SOURCE JOHN ?SOURCE ?AMT)) sets ?I to 10000, ?AMT to 10000, and ?SOURCE to SALARY. If there is no deducible instance of *pat*, the goal fails.

(THFORALL *pat conc*): succeeds if every instance of *pat* gives rise to a provable instance of *conc*. E.g., (THFORALL (IS ?X BLOCK) (COLOR ?X RED)) succeeds if the program can't find a non-red block.

In Section 2.2, I mentioned that assertions are tagged with *data dependencies* indicating what supports them. If an assertion is justified by a deduction from a goal involving one of the "collection" pseudo-predicates TOTAL, THFIND, MAX, and MIN. or THFORALL, then there will be a flaw in the data-dependency note. It really should depend upon no new relevant information being asserted; in the descriptions above, that means that no new instance of *pat* has appeared since the assertion was made. To enforce this kind of justification, Duck would have to set up a

"demon" watching for a new instance of *pat*. For efficiency, Duck does not do this, but makes the conclusion depend only on the data that were actually found. That is, if a known instance of *pat* is erased, any collection involving that instance will be invalid, and conclusions drawn from it erased, but if a new instance is asserted, nothing will happen. In Section 6.3 I will talk about how to correct this if you want.

*/(FORALLIN *exp tuple pat*): Succeeds if *pat* is true for all elements *exp* of *tuple*. E.g., given the database above, (FORALLIN !<?A ?B> !<!<JOHN MARY>> (LOVES ?A ?B)) would succeed. *exp* may be an arbitrary term, but it must be unifiable with each element of *tuple*. *pat* is then tested given the variable bindings resulting from each such unification. If each instance of *pat* succeeds, the FORALLIN succeeds.

*/(VARS *pat symb*): Tests whether *pat* contains any variables. If *pat* is a variable, VARS succeeds with *symb* = ALL. If it has no variables, *symb* = NONE. Otherwise, *symb* = PART. Goals of the form (NOT (VARS *pat symb*)) work properly.

The following predicates add flexibility to rule definitions:

*/(ASSERT *formula*): Always succeeds, adding *formula* to the database. When it occurs in a conjunction (such as the antecedent of a backward-chaining rule), the new assertion will be supported by the conjuncts so far. E.g.,

```
(<- (ABOVE ?X ?Y)
    (AND (ON ?X ?Y) (ASSERT (SUPPORTS ?Y ?X))))
```

does the following: when (ABOVE A B) occurs as a goal, it succeeds if (ON A B) is deducible, and asserts (SUPPORTS B A) as a side effect, justified by (ON A B). (Hence, if (ON A B) is ever erased, the assertion will be erased as well.)

*/(THCOND *-clauses-*) is like Lisp's COND, except that "ELSE" is used instead of "T." Each clause is of the form (*test -fmlas-*). Each *test* is a formula. Each test is examined until one is found that succeeds. The remaining clauses are then discarded, and the *fmlas* in the successful clause are tried. Even if they fail, the remaining clauses are never looked at.

A common use of THCOND is exemplified by the following

```
(<- (CONJUNCTS ?P ?Q)
    (THCOND ((:= ?P (AND !?ARGS)) (:= ?Q !<!?ARGS>))
            (ELSE (:= ?Q !<?P>))))
```

The idea is that (CONJUNCTS *exp* ?X) will set ?X to a tuple of the conjuncts of *exp*, a proposition. If *exp* is of the form (AND ...), ?X will be set to !<...>. If *exp* is not of this form, ?X will be set to !<*exp*>.

```
  (DUCK)
g?>(CONJUNCTS (AND A B C) ?X)
X = !<A B C>
...
g?>(CONJUNCTS FOO ?X)
X = !<FOO>
```

THCOND can also be asserted. In this case, after the first *test* succeeds, all the *finlas* after it are asserted.

*/(DELETE *pat*): When used as a goal or asserted, erases the pattern. This is a bizarre thing to do, and is usually wrong.

*/(NEWRULE *name pat*): Defines a new rule. Unlike the other pseudo-predicates, this one is used *only* in forward chaining, not as a goal. (You can call it in backward chaining by saying (ASSERT (NEWRULE ...)). If *name* is "PROMPT," NEWRULE prompts you for the name. Otherwise, it makes up a name, adding some numbers to the name given. The new rule will be asserted globally, even if it is created in a datapool. (See Section 6.2.)

4. INTERFACING LISP AND DUCK

4.1. Calling Duck from Lisp

In Chapter 2, I introduced the function ASSERTION-FETCH which returned a list of anses, one per assertion in the database matching a pattern. In Chapter 3, we talked about how implications in the database might cause deductive goals to become so complex they were best thought of as nondeterministic programs. To put these two views together, we need the idea of a *generated list*. A Lisp program should be able to start a Duck process working on a goal, doing all the things described in Chapter 3. Eventually, let us suppose, that process arrives at an answer to the initial goal. An answer consists of two parts: some variable bindings that represent a deducible instance of the goal, and a proof of that instance. This is just a slight generalization of what we called an *ans* in Chapter 2.

One strategy would be for Duck to return this ans and stop; an alternative might be to keep going and return all the anses it can, or stop after some number is accumulated. But what we really need is for Duck to send anses back to Lisp as they are found, while being "restartable" in case the ones found so far aren't sufficient. This strategy is implemented using *generated lists*. As explained in [McDermott 83a] these are like ordinary lists, except that in addition to elements proper, they may also contain *generators*, processes that can be resumed in order to add more real elements to the list. In many cases, a program using a generated list can pretend it is just a list, but use SAR and SDR instead of CAR and CDR. SAR and SDR check for generators and call them if necessary, so that only proper elements are seen by the program.

FETCH is a generalization of ASSERTION-FETCH which starts such a Duck process, and returns a generated list of anses that can be used to restart it and extract its results. As an example, suppose the database contains:

```
(MORTAL SWALE)          (MORTAL BONZO)

(<- (MORTAL ?X) (IS ?X HUMAN))

(IS SOCRATES HUMAN)     (IS XANTHIPPE HUMAN)
```

(:= GL1 (FETCH '(MORTAL ?X))) would set GL1 to

```
({ans ((X SWALE)) MORTAL16}   ; The names MORTAL16 and
 {ans ((X BONZO)) MORTAL33}   ; MORTAL33 were chosen by Duck
 (*gen ...))                  ; when the assertions were created
```

The first two elements are anses, corresponding to the two literal assertions that matched the goal. FETCH will always return those first. The third element is a generator. It will be called if the program doing the FETCH wants an element of GL1 past the first two. To get the third, for instance, it would evaluate (SAR (SDR (SDR GL1))). The generator would be called, restarting

the deductive process, which would now try backward chaining. A new ans would be found, and GL1 would be destructively altered, so that it looked like:

```
({ans ((X SWALE)) MORTAL16}
 {ans ((X BONZO)) MORTAL33}
 {ans ((X SOCRATES))          ; The name MORTALGUYS was chosen
      (BC MORTALGUYS IS25)}   ; by the user; see Section 3.5
 (*gen ...))
```

The new ans has a more complex proof record part, (BC MORTALGUYS IS25). This expression records that the answer was arrived at by a single backward-chaining ("BC") step, using the implication named MORTALGUYS and the literal assertion IS25.

Normally generated ans-lists are examined using FOR-EACH-ANS and other constructs described below. However, the function (EXTRUDE *n g-list*) can be useful. It keeps calling generators until the *g-list* begins with at least *n* proper elements, or until there are no generators left. In this case, (EXTRUDE 5 GL1) would cause GL1 to look like this:

```
({ans ((X SWALE)) MORTAL16}
 {ans ((X BONZO)) MORTAL33}
 {ans ((X SOCRATES))
      (BC MORTALGUYS IS25)}
 {ans ((X XANTHIPPE))
      (BC MORTALGUYS IS57)})
```

because there are only four answers.

A generated list behaves like a "pipeline" from Duck to Lisp. Duck stuffs anses in, and Lisp pulls them out. Note that from Duck's point of view, the guy at the other end of the pipe is sending it "artificial" failures. Even if its goal succeeded, a request for the next ans causes behavior identical to a failure -- the last choice point is resurrected, and something else is tried. If the number of choices is finite, then eventually they will all be tried, and Duck will give up. But it is all too easy to start infinite deductions which from Lisp's point of view are generated lists that never come to an end. (From Duck's point of view, this is a transaction with a customer who is impossible to please.)

FETCH makes a straight list of the anses derived by simple unification with literal assertions, followed by a generator that will try backward chaining. If nothing can be concluded from backward chaining, this generator will throw itself away (i.e., turn into ()) as soon as it is called. If you know backward chaining will not produce any results for a given pattern, or if you want to ignore them, you can use ASSERTION-FETCH instead of FETCH, and be guaranteed to get back an ordinary list of anses.

FOR-EACH-ANS is the Lisp analog of AND in Duck; a nested set of FOR-EACH-ANSes is equivalent to a single FOR-EACH-ANS of a conjunction. This:

```
(FOR-EACH-ANS (FETCH '(AND (R ?U ?V) (P ?U) (Q ?V)))
   (ADD '(FOO ?U ?V))   )
```

is identical in effect to this:

```
(FOR-EACH-ANS (FETCH '(R ?U ?V))
   (FOR-EACH-ANS (FETCH '(P ?U))
      (FOR-EACH-ANS (FETCH '(Q ?V))
         (ADD '(FOO ?U ?V))   )))
```

In either case, when the ADD finally happens, the values of U and V in the current ans will be substituted into (FOO ?U ?V). The same thing happens to the FETCHes along the way. If (R FRED JANE), (P FRED), (Q JANE) are in the database, then after the first FETCH finds (R FRED JANE), the second FETCH will use the pattern (P FRED), the third FETCH will use the pattern (Q JANE), and (FOO FRED JANE) will finally get asserted.

Now, what if the first FETCH had found (R ?X ?X) in the database? The FOR-EACH-ANS machinery must ensure that ?U and ?V in the next two loops are bound to compatible (i.e., unifiable) things. If (P FRED) is found by the second FETCH, (Q FRED) must be found by the third.

As you might have guessed, this is accomplished by just letting ?U have ?V as its "value," just as in Duck code. This doesn't constrain ?U at all, except that whatever it matches from now on must also match ?V. If the second FETCH find (P FRED), then ?V will be bound to FRED, and hence ?U will be also. So the third FETCH must find (Q FRED). [Alternatively, ?V might be bound to ?U; or ?U and ?V might both have some third variable as their value. All of these mechanisms rely on essentially the same idea.]

If nested FOR-EACH-ANSes are the same as conjunctive goals, why ever use the former, when the latter are more concise? This is to some extent a matter of taste. Typically a good place to use FOR-EACH-ANSes is when Duck code is overwhelmed by Lisp code:

```
(FOR-EACH-ANS (FETCH 'outer-pattern)
   A lot of Lisp code
   ...
   (COND (A Lisp test
          Much more Lisp code
          (FOR-EACH-ANS (FETCH 'inner-pattern)
             Large Lisp body   ))   ))
```

It is always possible to treat this sort of thing as a conjunction or at worst a disjunction of conjunctions (otherwise, Prolog would be impossible). But it is more efficient and clearer to stay within Lisp with an occasional FETCH than to stay within Duck with many CALLs, THCONDs, etc. I will say more about these issues in Section 5.1.

ADD, FETCH and other functions that take pattern arguments substitute into those arguments

the values of Duck variables from the current ans. In many cases, we want the values of Lisp variables as well. For instance, suppose we needed a Lisp program that looks in the database for assertions

(PARTS *device parts-list*)

and asserts for each one:

(NUMBER-OF-PARTS *device number*)

Here is the code:

```
(FOR-EACH-ANS (FETCH '(PARTS ?DEV ?PARTS))
   (LET ((I (LENGTH (TUPELTS ?PARTS))))
      (ADD (LIST 'NUMBER-OF-PARTS '?DEV I))   ))
```

The first thing to explain is the function TUPELTS. Recall that ?PARTS is a tuple, that is, an expression of the form (TUP . . .). TUPELTS extracts the list of its elements. (You can use CDR if you want to confuse the readers of your code.)

Before explaining the further, let me make one revision:

```
(FOR-EACH-ANS (FETCH '(PARTS ?DEV ?PARTS))
   (LET ((I (LENGTH (TUPELTS ?PARTS))))
      (ADD !'(NUMBER-OF-PARTS ?DEV ,I))   ))
```

In the pattern of an ADD, FETCH, etc., it is almost always clearer to use *backquote*, or *quasi-quote*. !'(*-stuff-*) is like '(*-stuff-*), except that pieces marked with a comma ("*,x*") are evaluated. In building up assertion and fetch patterns, the use of LIST and quote obscures what's going on. (Quasi-quote is explained the Nisp manual [McDermott 83a]; most modern Lisps provide some version of this feature.)

In the example, the argument to ADD will be, in effect, traversed twice, first by !' in the process of evaluating it, then by ADD, which substitutes in the values of variables. Because of this, the following variant has the same effect:

```
(ADD !'(NUMBER-OF-PARTS ,?DEV ,I))
```

In this version, the value of ?DEV is substituted in when the argument is evaluated rather than later.

To avoid this double sweep, it is possible to create new Duck-variable bindings in Lisp code, using the Lisp magic word ANS-BIND.

```
*/(ANS-BIND (sign (var val) (var val))
     -body-)
```

locally alters the current ans with bindings of the *vars*. If the *sign* is +, these bindings are added to existing ones; if =, these bindings replace the existing ones. So our part-number example could have been written:

```
(FOR-EACH-ANS (FETCH '(PARTS ?DEV ?PARTS))
   (ANS-BIND (+ (I (LENGTH (TUPELTS ?PARTS))))
      (ADD '(NUMBER-OF-PARTS ?DEV ?I))   ))
```

We can now dispense with the quasi-quote, since the variable-substitution sweep now handles both ?DEV and ?I.

The current ans-variable bindings used by FETCH and ADD are kept in the free variable ANS*, which is rebound by ANS-BIND, FOR-EACH-ANS, etc. Unfortunately this variable is not quite as invisible as I would like; ignoring it can get you into trouble. For instance, consider the following function:

```
(DE FATHER (X)
   (FOR-FIRST-ANS (FETCH !'(FATHER ?F ,X))
      ?F   ))
```

FATHER uses FOR-FIRST-ANS, which here will return the value of ?F from the first ans found by the FETCH; hence it will find X's father. But if (FATHER 'JANE) is called when ANS* contains a binding of ?F, then this function will not do what its writer intended. Instead, it will check to see if ?F is Jane's father, and if so return it, else ().

The Lisp magic word QUACK can be used to fix this problem. */(QUACK -*body*-) evaluates *body* and returns the value of its last expression, but it does this with ANS* bound to an empty ans, and hence "insulates" it from an ongoing deduction. QUACK should be used whenever the existing value of ANS* is irrelevant to the deductions a function does. FATHER should have been written

```
(DE FATHER (X)
   (QUACK
      (FOR-FIRST-ANS (FETCH !'(FATHER ?F ,X))
         ?F   )))
```

It is almost always correct to wrap QUACK around the body of a function doing Duck operations. (The need for QUACK is a flaw in the design of Duck, but one not likely to be fixed anytime soon.) (QUACK ...) is almost the same as (ANS-BIND (=) ...), except that it also clears the proof-record parts of the current ans, as explained in Chapter 6.1.

Lisp and Duck variables coexist in Lisp code. They behave rather differently. A Lisp variable is bound, and may be reassigned several times. A Duck variable is never reassigned, but just gets "more assigned" as time goes by. It starts off completely unconstrained. Once it gets bound, later unifications can only constrain its value further, by binding variables that occur as part of its value. Hence although for completeness you are allowed to execute (:= ?*var val*) in Lisp, binding ?*var* to *val*, this operation cannot arbitrarily change the value of ?*var*, but can only further constrain it. E.g., if ?X == (F ?Y), executing (:= ?X '(F A)) will bind ?Y to A. If the new value doesn't match the old, as in (:= ?X 'A), an error will result. If you want to set a Duck variable in Lisp, but fail gracefully if the new binding is incompatible with the old, write

```
(FOR-EACH-ANS (MATCH '?var val)
   -stuff-dependent-on-new-value-)
```

MATCH returns a list of anses, one for each way its two arguments could unify. (See the end of this section for a fuller description.) This list will normally have zero or one element, but could have more if segment variables are involved.

In Duck, := works better. Executing (:= ?X (F A)) will bind ?X to (F A), or just fail if ?X's current value is incompatible with (F A).

To summarize, here are the functions for producing g-lists of anses. The first four use the database; MATCH operates by unifying its arguments. Remember that these are Lisp functions, and cannot be called from Duck without using the techniques of the next section.

- (FETCH *pattern*): Returns a g-list of anses, one for each proof of an instance of the pattern. There may be more than one proof of a given instance, and all will be returned.
- (NODUPFETCH *pattern*): Like FETCH, but returns only the first instance of any ans. (That is, if two or more anses yield identical pattern instances when substituted, then only one of them will be returned.)
- (ASSERTION-FETCH *pattern*): Returns a list of anses, one for each assertion in the database. Does no backward chaining.
- (BACK-CHAIN *pattern*): Returns a g-list of anses that can be deduced by backward chaining, without ever looking for literal assertions that unify with the pattern.
- */(MATCH [RENAME] *pat1* [RENAME] *pat2*): Returns a g-list of anses, one for each unification of the two patterns. Each ans contains variable bindings for *pat1*; any bindings of variables from *pat2* are discarded. If the RENAME flag is present before a pat, its variables are RENAMEd before the match. So (MATCH '(P A ?X) '(P ?X B)) returns (), while (MATCH '(P A ?X) RENAME '(P ?X B)) returns ({ans ((X B)) (PREMISS)}). (The expression (PREMISS) in an ans means that it doesn't depend on anything in the database.)

Here is a summary of the macros for examining lists of anses. Again, these are Lisp magic words, not Duck pseudo-predicates:

- */(FOR-EACH-ANS *ans-g-list -body-*): evaluates its first argument (usually a FETCH or MATCH) to get a g-list of anses, then evaluates the body for each ans in the g-list. The body has the same format as a Nisp LOOP, [McDermott 83a] so you can use WHILE, UNTIL, and RESULT inside it.
- */(FOR-ANS *ans -body-*): evaluate body once, with current ans ANS* bound to the value of *ans*.
- */(FOR-FIRST-ANS *ans-g-list -body-*): like FOR-EACH-ANS, but stops after first ans, returning the value of the last thing in body.
- */(FOR-SOME-ANS *ans-g-list -body-*): like FOR-FIRST-ANS, but returns the first ans for which the last thing in body evaluates to non-().

4.2. Calling Lisp from Duck

Once control is in Duck, it would be nice if we could just leave it there, piping back "pure" deductions for the use of a Lisp program. Unfortunately, in practical applications deductions are always impure. It is often necessary to interpose nondeductive operations, from arithmetic to I/O. One approach, taken by Prolog [Warren 77], is to add such things to the deductive language. Duck follows this approach up to a point; the predicates and pseudo-predicates of Section 3.6 represent useful extensions of dubious deductive legitimacy. However, the Duck philosophy is that if a large chunk of *ad hoc* computation is necessary in the midst of a deduction, it should be done in Lisp, not in Duck. The result should be summarized in an ans, so that Duck can report its results back in a form as close as possible to a deduction.

Hence there must be mechanisms to allow Duck to call a Lisp function, and get anses back. The pseudo-predicate CALL does the first half. Whenever */(CALL (*fun -args-*)) occurs as a goal, the current variable values are substituted into the *args*, and the Lisp function *fun* is applied to the results. The function decides whether and how the CALL-goal succeeds.

For instance, consider the goal (CHILDHOOD-DISEASE-OF FRED ?WHAT), which might occur in a medical diagnosis program, and succeeds if ?WHAT is a disease Fred had in childhood. This is information that can be obtained only by asking the user, and this must be done in Lisp code. So we provide a rule

```
(<- (CHILDHOOD-DISEASE-OF ?PERSON ?DIS)
    (CALL (ASK-USER-ABOUT-CHILDHOOD-DISEASE ?PERSON)))
```

and a function

```
(DE ASK-USER-ABOUT-CHILDHOOD-DISEASE (PERSON)
   (COND ((NOT (HASMVARS PERSON))
          (TTYMSG "Type a list of the diseases " PERSON
                  " had as a child: ")
          (LOOP FOR ((X IN (READ)))
             (FOR-EACH-ANS (MATCH ?DIS X)
                (NOTE)   )))   ))
```

There are several things to point out here. First, when a function is CALLed this way, ANS* will contain the variable bindings Duck has accumulated at that point. So here variables can be accessed two ways: as argument of the function, or through the current ans. In the Lisp function ASK-USER-ABOUT-CHILDHOOD-DISEASES, both PERSON and ?PERSON will have the same value. The Lisp variable is more efficient to access.

Second, the way such a function sends anses back is by using FOR-EACH-ANS, MATCH, etc. to constrain the current ans, and, when it has a good one, call NOTE to post it as an answer to the CALL-goal. In the example, each symbol typed by the user is checked to make sure that it unifies

with ?DIS, and for each match, a new answer is sent back.

Third, the function HASMVARS tests whether its argument has any logic variables. This function is necessary in CALLed functions to make sure that the current goal is one that the function can handle. To test whether an expression consists entirely of a logic variable, use (IS mvar *expression*), which tests whether the value of *expression* is an mvar, or "match variable."

Observe what happens given the following goals:

- (CHILDHOOD-DISEASE-OF FRED ?WHAT): Since FRED contains no variables, the function types

  Type a list of the diseases FRED had as a child:

  If the person types "(MEASLES MUMPS)", then ?WHAT is unified with MEASLES, and the resulting ans is NOTEd; then with MUMPS, and that ans is noted as well. The goal succeeds with ?WHAT= MEASLES. If a failure propagates back to here, ?WHAT=MUMPS will be tried instead.
- (CHILDHOOD-DISEASE-OF FRED MUMPS): Exactly as before, but now just one of the matches succeeds. There is at most one answer to such a goal.
- (CHILDHOOD-DISEASE-OF ?WHO MUMPS): Now the HASMVARS test fails. If it weren't there, the message

  Type a list of the diseases (\? _549) had as a child:

  would have been printed. As an exercise, try writing a function to handle this goal. (Hint: It should print out "Type a list of all persons who had MUMPS as a child: " in this case.)
- (CHILDHOOD-DISEASE-OF ?WHO ?WHAT): This goal would need a third function.

Actually, there is a bug in this rule. Every time a goal of the form (CHILDHOOD-DISEASE-OF FRED ...) occurs, the user will be asked to repeat the information. Clearly, we should store it in the database, and ask only if we haven't asked before. Here is the way to do that:

```
(DE ASK-USER-ABOUT-CHILDHOOD-DISEASE (PERSON)
   (COND ((AND (NOT (HASMVARS PERSON))
               (NOT (FETCH '(ASKED-ABOUT (CHILDHOOD-DISEASE ?PERSON)))))
          (TTYMSG "Type a list of the diseases " PERSON
                  " had as a child: ")
          (LOOP FOR ((X IN (READ)))
             (FOR-EACH-ANS (MATCH ?DIS X)
                (ADD '(CHILDHOOD-DISEASE-OF ?PERSON ?DIS))
                (NOTE)   ))
          (ADD '(ASKED-ABOUT (CHILDHOOD-DISEASE ?PERSON))))   ))
```

You might try writing a more general version of this as a CALLed function or Lisp macro.

In most cases, we can conceal the use of CALL by using

*/(LISPRULE *name trigger* [<- | -> *-body-*)]

This is an abbreviation for

```
(RULE name trigger [ <- | -> ] (CALL name))
(DE name () -body-)
```

In this case, the function is not allowed to have any arguments; when there are no arguments, (CALL (*fun*)) can be abbreviated (CALL *fun*).

So our example could have been written as follows (changing all occurrences of PERSON to ?PERSON):

```
(LISPRULE ASK-USER-ABOUT-CHILDHOOD-DISEASE
          (CHILDHOOD-DISEASE-OF ?PERSON ?DIS)
          <-
   (COND ((AND (NOT (HASMVARS ?PERSON))
               (NOT (FETCH '(ASKED-ABOUT (CHILDHOOD-DISEASE ?PERSON)))))
          (TTYMSG "Type a list of the diseases " ?PERSON
                  " had as a child: ")
          (LOOP FOR ((X IN (READ)))
             (FOR-EACH-ANS (MATCH ?DIS X)
                (ADD '(CHILDHOOD-DISEASE-OF ?PERSON ?DIS))
                (NOTE)   ))
          (ADD '(ASKED-ABOUT (CHILDHOOD-DISEASE ?PERSON))))   ))
```

DEFPRED may also be used to define lisprules. If an expression of the form (LISP *-body-*) occurs in a DEFPRED, it means the same as (LISPRULE *pred*-DEF-*n* <- *header -body-*).

Note that we allow either arrow (<- or ->) in a lisprule. That's because CALL can be used during forward chaining as well as backward. An implication of the form

(-> *pattern* (CALL (*fun -args-*)))

will cause the fun to be applied to the args (with variables substituted) whenever an instance of the pattern is ADDed. As before, if there are no args, you may write (CALL *fun*). Timing note: the function will not be called until all other forward chaining is done. If more than one such rule is triggered, they are queued up and run in random order, after all other forward chaining.

When a rule of the form (-> *p q*) is used, the *q* will be asserted in addition to the *p*. Sometimes what is wanted instead is for *q* to *replace p* in the database. To accomplish this, write

*/(REVISION-RULE *p* [INVIS] *-body-*)

The body is a sequence of Lisp expressions which are evaluated (just as for a LISPRULE). The value of the last one replaces *p* in the database. If the flag INVIS occurs, then the user will still see the assertion in the form *p* (if he edits it in core). Forward chaining occurs from *p* before the revision as well as after.

A function CALLed during backward chaining finds a bunch of answers, NOTEs each of them, and returns. (Its value is ignored.) This is normally an adequate interface, but occasionally one wants to take advantage of the generated list between the function and the ultimate user of the answers; that is, to generate a few answers, and then suspend, ready to generate more. This is accomplished with

/(AU-REVOIR (-vars-*) *-body-*)

which sends a *generator* instead of an ans. The generator, when restarted, will evaluate *-body-*, which may do more NOTE-ing. The *-vars-* are all the free variables that occur in the *-body-*, if any. For example, suppose a planner needs to borrow $100 from someone. It needs a lisprule to ask a human to go find a lender. If the first one doesn't work out, it will try others, but there is no point in asking them if the first person is willing, and meets all the other criteria. Here is the rule we want:

```
(LISPRULE FIND-LENDER (WILLING-TO-LEND ?PERSON ?AMOUNT) <-
   (COND ((AND (HASMVARS ?PERSON) (NOT (HASMVARS ?AMOUNT)))
          (ANOTHER-LENDER ?AMOUNT))   ))

(DE ANOTHER-LENDER (AMT)
   (TTYMSG "Go find a person who would be willing to lend me $"
           AMT "."
           T "Type his or her name (or () if unknown): ")
   (LET ((NAME (READ)))
      (COND (NAME
             (FOR-EACH-ANS (MATCH ?PERSON (READ))
                (NOTE)   )
             (AU-REVOIR (AMT) (ANOTHER-LENDER AMT)))   )))
```

If the user types a non-() NAME, then it is NOTEd, and an AU-REVOIR is executed to tell the caller that if that lender is no good, to run ANOTHER-LENDER again. (The expression (AMT) following the AU-REVOIR is to tell AU-REVOIR what part of the current Lisp variable-binding environment needs to be restored when the process is restarted.)

I should point out that the way in which Duck variables become known inside a CALLed function is through that not-so-invisible variable ANS*. Hence, this is one case where it is definitely wrong to use QUACK. If you do, the bindings coming into the function will be lost.

NOTE: LISPRULE and REVISION-RULE are Lisp magic words; they are used to define rules. CALL is a Duck pseudo-predicate; it appears in a goal or asserted pattern, and causes Lisp to be called. NOTE is a Lisp function. It sends the current ans back to the goal that did the CALL that got to this piece of Lisp code. AU-REVOIR is a Lisp magic word that sends a generator of more anses. It takes a while to sort out which is which, but attempting to use a Lisp construct in Duck or vice versa will not work.

4.3. Interfacing Forward and Backward Chaining

Forward and backward chaining are quite different processes. The former just runs, adding conclusions to the database; the latter is goal-directed, attempting to prove particular conclusions. It is fairly common, however, to need to call one from the other.

For example, consider the fact "If (P ?X) and (Q ?X), then (R ?X)." To use this fact "backward," one would write

```
(<- (R ?X) (AND (P ?X) (Q ?X)))
```

We have a choice of whether to put P first or Q first, but let's ignore that.

To use this fact as a forward rule, we would write

```
(-> (P ?X) (-> (Q ?X) (R ?X)))
```

That is, if (P α) is ever asserted, assert (-> (Q α) (R α)). This will interact with (Q α) to infer (R α). Again, there is an issue of whether to put P or Q first, but we will ignore it.

What we will not ignore is that there is another possibility: Wait until (P α) is asserted, but then do not wait passively for (Q α) to arrive, but attempt to deduce it. That is, when (P ...) is asserted, start a backward-chaining process for the corresponding Q pattern, and, if it succeeds, assert the resultant R pattern. This is accomplished by writing

```
(-> (P ?X) (-< (Q ?X) (R ?X)))
```

(-< p q) is another way of saying "If p then q," but its procedural meaning is that an attempt should be made to prove p, and q should be asserted for every answer generated in that attempt.

This can be confusing because there are several constructs (like AND and THCOND) which have slightly different meanings in forward vs. backward chaining. E.g., asserting this:

(-< (AND $g_1 \ldots g_M$)
    (AND $a_1 \ldots a_N$))

causes the g_i to be used as goals in the usual nondeterministic way. Each instance of the conjunction that can be proved causes all of the as to be asserted; there is no backtracking on the right-hand side.

Here is a list of all the ways to represent implications. All of them mean "If p then q." But some are used during backward chaining, some during forward. For completeness, -> and <- are included in the following list:

- (-> p q): Assert q whenever p is asserted (forward chaining).
- (<- q p): Make p a subgoal of q whenever q occurs as a goal (backward chaining).
- (-< p q): If asserted, finds all instances of p (by *backward* chaining), and asserts q for each of them. The form (-< p >-) does backward chaining without doing any forward chaining for each answer. This is only worth doing if p has side effects.
- (-<> p q): Combination of -< and ->. Asserts (-> p q), but also does backchaining

on p. Every instance of p found in this process, or asserted later, will cause an assertion of the corresponding instance of q. (But instances of p that merely become deducible later without being explicitly asserted will not cause assertion of q.)

Occasionally one wants to call forward chaining from backward chaining. This is accomplished using ASSERT, described in Section 3.6.

5. EXAMPLES

5.1. Duck Programming Styles

At this point, the basic features of Duck have been presented. The rest of this manual is concerned with advanced topics in Duck, and with tools that make Duck program development easier. In this section, I would like to pause and make some observations about how Duck systems are put together.

We started by viewing Duck as a "passive" database, but have built it, via backward chaining, into a general-purpose programming language. This means that Lisp functions and Duck rules are in competition: any algorithm may be encoded in either language.

Of course, Duck is a rather different language from Lisp. For one thing, it is *nondeterministic*. That is, where in Lisp you might write:

```
(FOR-SOME-ANS (FETCH '(NODE-OF ?MODULE ?N))
   (FOR-SOME-ANS (FETCH '(VOLTAGE ?N HIGH))
      ...))
```

in Duck you simply write:

```
(AND (NODE-OF ?MODULE ?N)
     (VOLTAGE ?N HIGH)
     ...)
```

The meaning of AND is such that answers to the first goal will be generated one by one, and tried on the second, until one succeeds, when "..." will be executed. The interpreter automatically undoes a faulty guess and tries another, so you don't have to program such a loop explicitly. Another name for such nondeterminism is *backtracking*.

This conciseness is purchased at the price of uncontrollability. In Duck, a goal failure at any later time can cause these goals to be retried. Suppose the code above had occurred in a rule with consequent (HOTNODE ?MODULE ?N). Suppose that it had been invoked in a goal like:

```
(AND (HOTNODE MODULE39 ?N) (CONNECTED ?N PREAMP))
```

Then a failure of the CONNECTED goal could cause the VOLTAGE or NODE-OF sub-goals to be retried, in an attempt to find a different solution to the HOTNODE goal.

There is thus a perpetual trade-off between efficiency and clarity. Duck rules often have the pleasant feature that their meanings are transparent; they can be understood in isolation. The inefficiency that results is often unimportant, but you should constantly be on the lookout for it.

Where does "logic programming" fit into all of this? Although this phrase has meant many things, I will take it to mean the use of Horn-clause theorem proving as an underlying computational model, instead of, e.g., the lambda calculus. (By "Horn-clause theorem proving," I

just mean unification and backtracking.) Many programs written to manipulate symbolic structures can just as easily be written in logic as in Lisp.

One example is APPEND, shown in Section 3.5. For another, take the table-lookup function LOOKUP[3], which takes a key and a table consisting of a list of pairs ((*key value*) (*key value*) ...), and returns the entry for the given key, or () if there isn't one. This function can be written in a predicate-calculus-based system like Duck, if we make a few adjustments. First, we represent the table as a tuple of tuples: !<!<*key value*> !<*key value*> ...>. Second, rather than return () in case it doesn't find an entry, the function will just fail. Third, since programs in logic are in terms of predicates, or *relations*, we must give LOOKUP a third argument, so that (LOOKUP *key table val*) is to mean "*val* is the result of looking up *key* in *table*."

Here is the resulting program:

```
(DEFPRED (LOOKUP obj ?K (LST (LST obj)) ?TAB obj ?V)
   (LOOKUP ?K !<!<?K ?V> !?R> ?V)

   (<- (LOOKUP ?K !<?X !?R> ?V)
       (LOOKUP ?K ?R ?V))   )
```

This is quite pretty in some ways. Note the use of unification in the first clause to extract the pieces of the first table entry, compare the key to ?K, and set ?V to the value.

However, it has some drawbacks. First, in Duck, it will be two orders of magnitude less efficient than the equivalent hard-wired Lisp function. (This criticism would not apply to a system like Prolog.)

Second, backtracking causes unusual behavior on the part of a program using LOOKUP. Suppose we had the conjunctive goals:

```
(AND ...
     (LOOKUP ?K ?TAB ?V)
     (FOO ?V))
```

and (FOO ?V) failed after LOOKUP had found ?V. Then the interpreter would go back into LOOKUP to look for another entry. On rare occasions this would be the correct course, but usually it is wasteful or even wrong.

To correct this problem, LOOKUP would have to be rewritten as

```
(DEFPRED (LOOKUP obj ?K (LST (LST obj)) ?TAB obj ?V)
   (THCOND ((:= ?TAB !<!<?K ?V> !?R>))
           ((:= ?TAB !<?X !?R>)
            (LOOKUP ?K ?R ?V))   ))
```

[3]Known as ASSOC in many Lisps.

THCOND is used to squelch backtracking. If (:= ?TAB !<!<?K ?V> !?R>) succeeds, that is, if ?TAB can be unified with !<!<?K ?V> !?R>, then the unification is done, and the alternative branch, in which the first element of the table is passed over, is forgotten. (For Prolog hackers, it is as if there is a "cut" after the tests in THCOND clauses.)

This is a recurrent issue that pops up in "logic programming." So long as the rules can be confined to simple truths about domain entities (electronic circuits, diseases, or the like), then they are substantially clearer than Lisp code. As soon as they are used to talk about data structures (lists and tables), then in practice they are about the same as Lisp code, but harder to control.

To some extent the choice of logic or Lisp is a matter of taste. Therefore, in Duck I have made sure that logic programming can be indulged in when required. There are no artificial limits on what can be expressed. Full depth-first Horn-clause theorem proving is supported. But the clear intent is that Duck be used at a higher level: The user is supposed to be able to make sense of the proof supporting a given result. For this reason, Duck always keeps track of all such proofs, and explicitly attaches them to assertions. This is described in detail in Chapter 6.1, below.

Another important issue is the role deduction plays in a real program. The mechanisms are elegant, but once you have gotten used to them you may find yourself asking, How relevant are they? How many intellectual tasks consist of deducing something? Where does "plausible reasoning" fit in? Are Duck and its ilk mainly for building mathematical theorem provers?

To answer all these questions, you must understand that backward chaining in search of a deductive answer is only a fraction of the work of a reasoning program. A practical program will consist of an algorithmic shell that breaks a problem down into phases, each of which will require one or more deductions. For example, a planning problem solver might first deduce several candidate plans, then deduce patches to bugs in them, then deduce which of the surviving plans is best. Each of these phases has its own distinctive flavor.

In phase 1, the deduction starts with the goal (PLAN-FOR *problem* ?PLAN). Solutions to this query are potential plans to attack the problem. They are *not* guaranteed, optimal plans. It would be difficult to ensure this while retaining the modular character of the rules. If all you're looking for is candidate plans, then there is no contradiction if two independent rules each propose one. If you're looking for the best plan, one rule would be in error, and would have be revised so it specifically fails when the other succeeds. This kind of delicate tuning is just what we want to avoid.

In the second phase, we are deducing patches to the generated plans. Here the deductive queries are of the form "Find a ?PATCH such that it would fix a given bug in a given plan." Each

patch gives rise to a new variant of the plan.

In phase 3, we attempt to deduce which plan is best. Here we can make use of modular statements of the interaction information we suppressed in phase 1. A typical rule might say, "A plan that uses less money is better." Formalizing this rule is an interesting exercise. If two such rules disagree, there must be a way of deciding among them. The simplest way is to have each rule give a numerical weight saying how much better one plan is than another. For instance, the rule might say

```
(<- (WEIGHT ?PLAN1 ?W)
    (AND (COST ?PLAN1 ?C1)
         (COST ?PLAN2 ?C2)
         (< ?C1 ?C2)
         (EVAL (* 10 (- ?C2 ?C1)) ?W)))
```

In this way, time, money, and other resources can be compared.

Thus the use to which deduction is put is often ultimately "non-deductive." Adding up all the weights from rules of this kind and picking the highest is not itself a deductive step. A program will be clean, modular, and extensible to the degree that it consists mainly of deductions glued together by decision-making steps like these.

So the answers to the questions I posed a few paragraphs back are these: Duck-style deductive retrievers are not tailored for mathematical theorem proving, except in so far as this activity can be broken into short, shallow deductions. In fact, Duck programs almost always find themselves doing "plausible reasoning," because what they actually deduce are *candidate* answers to queries, which are then debugged and culled by further processes, some deductive and some not.

5.2. Symmetry and Transitivity

A symmetric relation R is one that satisfies the axiom:

```
(IF (R ?X ?Y) (R ?Y ?X))
```

The IF cannot be represented as a backward rule

```
(<- (R ?Y ?X) (R ?X ?Y))
```

because Duck's simple backtracker would simply go into an infinite loop. Given a goal like (R A B), it would try the subgoal (R B A), which would generate the subgoal (R A B) again, and so on *ad infinitum*.

Often there are many literal assertions involving R, but no rules except the one declaring symmetry. For instance, R might be CONNECTED, where (CONNECTED x y) means that x is connected to y in some graph (like a street map; see below). In that case, it suffices to make the rule into a forward rule

```
(-> (R ?X ?Y) (R ?Y ?X))
```

Now when (R B A) is asserted, (R A B) will be asserted, too, and there will be no need for backward chaining should (R A B) ever occur as a goal.

When there are other rules involving R, one can sometimes simply omit symmetry. Suppose R is COUSIN. This is certainly symmetric, but we do not have to say so, because the rules used to infer (COUSIN *x y*) can be used to infer (COUSIN *y x*) as well.

When these solutions are inadequate, we must bite the bullet and figure out how to include a backward-chaining rule. Rather than do this many times, once per predicate, we write a "meta-predicate" SYMMETRIC, such that asserting (SYMMETRIC *pred*) causes the appropriate rule to be created:

```
; SYMMETRIC takes a binary predicate as argument
(DEFPRED (SYMMETRIC (FUN prop (obj obj)) ?PRED))

(RULE SYMM-DEF
   ; If a predicate is symmetric
   (-> (SYMMETRIC ?PRED)
       ; then if you need to prove it true of two arguments
       (<- (?PRED ?X ?Y)
           ; call SYMM-CHECK
           (CALL (SYMM-CHECK ?PRED ?X ?Y))))   )

; A stack of goals for which symmetry should not be used
; (Because it's already being tried.)
(SPECDECL (LST sexp) (SYMMGOALS* NIL))

(PROC SYMM-CHECK void (symbol PRED sexp X Y)
   (LET ((E !'(,PRED ,X ,Y)))
      (COND ((NOT (MEMBER E SYMMGOALS*))
             (SYMM-TRY PRED X Y (CONS E SYMMGOALS*)))  )))

(PROC SYMM-TRY void (symbol PRED sexp X Y (LST sexp) SYMMGOALS)
   (BIND ((LST sexp) (SYMMGOALS* SYMMGOALS))
      (LET ((E !'(,PRED ,Y ,X)))
         (COND ((NOT (MEMBER E SYMMGOALS*))
                (SYMM-EXTRUDE (FETCH !'(,PRED ,Y ,X))
                              SYMMGOALS))   ))))
```

```
; This function does the work, once the bindings are set up.
; It generates the answers to the reversed goal !'(,PRED ,Y ,X),
; on demand, feeding them back to the original caller.
(PROC SYMM-EXTRUDE void ((GLST ans) AGL (LST sexp) SYMMGOALS)
   (LOOP
    WHILE AGL
    RESULT NIL
    UNTIL (IS *GEN (CAR AGL))
      (NOTE (CAR AGL))
      (:= AGL (CDR AGL))
    RESULT (AU-REVOIR (AGL SYMMGOALS)
             (BIND ((LST sexp) (SYMMGOALS* SYMMGOALS))
                (SYMM-EXTRUDE (NORMALIZE AGL) SYMMGOALS)   ))   ))
```

The idea is to keep track of a list SYMMGOALS* of goals that have been proposed because of the symmetry of ?PRED. If a goal is proposed, and it is already on this list, then it is discarded, thus cutting the recursion off after one level.

The mechanism would be simpler except that we want to generate the answers on demand, rather than find all the answers to the reversed goal. For this we use AU-REVOIR, repeatedly calling SYMM-EXTRUDE to chew another answer off the list.

Note that SYMM-CHECK was passed ?PRED, ?X, and ?Y as arguments; it would not have worked to refer to these variables using question marks inside SYMM-CHECK. This can be confusing, since normally in a rule of the form

(<- *goal* (CALL (*f* ...)))

the variables of the *goal* are available inside the definition of *f*. In this case, however, the backward rule was asserted by forward chaining from a bigger rule, and hence its variables will have been eliminated or renamed by the time the function is called.

A *transitive* relation R is one that satisfies the axiom:

```
(IF (AND (R ?X ?Y) (R ?Y ?Z))
    (R ?X ?Z))
```

If we represent this as a backward rule, then infinite recursions are almost certain. A goal of the form (R A ?V) will generate a subgoal of exactly the same form. If there are any answers, then they will be found, but the rule will not know when to stop; a failure back into this rule after the last answer is found will start a futile infinite recursion.

Usually an R of this kind occurs with a "base relation" to get things started. For instance, the predicate (ANCESTOR *x y*) ("*x* is an ancestor of *y*") is defined in terms of PARENT thus:

```
(DEFPRED (ANCESTOR person ?A ?P)
   ; Base case:
   (PARENT ?A ?P)
   ; plus

   ; either 1)
   (AND (ANCESTOR ?A ?Q) (ANCESTOR ?Q ?P))

   ; or 2)
   (AND (PARENT ?A ?Q) (ANCESTOR ?Q ?P))

   ; or 3)
   (AND (PARENT ?Q ?P) (ANCESTOR ?A ?Q))   )
```

where we have a choice of how to do the recursion. I already argued against the first choice. The second is to handle a goal of the form (ANCESTOR A ?V) by finding a child of A and then finding a descendant of that child. The third choice is to handle a goal of the form (ANCESTOR ?V A) by finding a parent of A and then finding an ancestor of that parent.

Here is a problem: the "ancestor-down" strategy works well only when you start with the ancestor and need to find descendants; the "descendant-up" strategy works well only when you start with the descendant and want to find ancestors. Consider what happens when you use the former on a goal of the form (ANCESTOR ?V A). The first subgoal is (PARENT ?V ?Q), "Find all parent-child relationships." For each such relationship, Duck will try to show that ?Q is an ancestor of A. This is a dreadful strategy.

The solution is to employ both strategies, but use VARS to decide which to use on what problem:

```
(DEFPRED (TRANSITIVE (FUN prop (obj obj)) ?PRED ?BASIS))

(RULE TRANS-DEF
      (-> (TRANSITIVE ?PRED ?BASIS)
          (AND (<- (?PRED ?X ?Z) (?BASIS ?X ?Z))

               (<- (?PRED ?X ?Z)
                   (AND (NOT (VARS ?Z NONE))
                        (?BASIS ?X ?Y)
                        (?PRED ?Y ?Z)))

               (<- (?PRED ?X ?Z)
                   (AND (VARS ?Z NONE)
                        (?BASIS ?Y ?Z)
                        (?PRED ?X ?Y)))   )))
```

If the second argument ?Z has variables or is a variable ((NOT (VARS ?Z NONE))), then we use the first strategy, else the second. We tell Duck about ANCESTOR by just asserting (TRANSITIVE ANCESTOR PARENT). Note that on a goal of the form (ANCESTOR ?X ?Y) this version uses the

"ancestor-down" strategy, finding all the parent-child relationships, and then the descendants of each child. I leave it as an exercise to write the version that goes "descendant-up" in that case. Obviously, it doesn't matter much.

In cases where there is no single basis case, we can define a new predicate to be the disjunction of all the bases, and use that instead.

5.3. Path Finding

Now for a slightly more complex example. Suppose we have a set of cities linked by highways:

```
(DEFDUCKTYPE city obj)

(DEFPRED (LINK city ?C1 ?C2))
```

We want LINK to be symmetric, and the easiest way to do that is this rule:

```
(RULE LINKSYM -> (LINK ?C1 ?C2) (LINK ?C2 ?C1))
```

The object is to define the predicate PATH

```
(DEFPRED (PATH city ?C1 ?CN (LST city) ?P))
```

which is intended to mean that ?P is a nonlooping path from ?C1 to ?CN (represented as a tuple of cities, including ?C1 and ?CN).

In case you are tempted to say something like

```
    (PATH ?C ?C !<?C>)
and
    (<- (PATH ?C1 ?CN !<?C1 !?P>)
        (AND (LINK ?C1 ?C2)
             (PATH ?C2 ?CN ?P)))
```

forget it. The graph of links may have loops, in which case this approach would generate infinite recursions, and find looping paths as well. Instead, we must define an auxiliary predicate PATH1 which explicitly keeps track of cities which must be avoided on pain of looping:

```
(DEFPRED (PATH1 city ?C1 ?CN (LST city) ?P ?AVOID)
  ; True if ?P is a path from ?C1 to ?CN that doesn't go through
  ; any city in ?AVOID

   ; The basis case:
   (PATH1 ?C ?C !<?C> ?A)
```

```
; !<?C1 !?P> is a path if ...
(<- (PATH1 ?C1 ?CN !<?C1 !?P> ?AVOID)
    (AND ; ?C1 and ?CN are distinct
         (NOT (SAMECONST ?C1 ?CN))
         ; the first step exists
         (LINK ?C1 ?C2)
         ; it doesn't take you to an avoided city
         (NOT (ELEMENT ?C2 ?AVOID))
         ; ?P is a path from ?C2 to ?CN that avoids
         ; ?C1 in addition to the previous list
         (PATH1 ?C2 ?CN ?P !<?C1 !?AVOID>)))   )
```

Now PATH can be defined in terms of PATH1 easily:

```
(DEFPRED (PATH city ?C1 ?CN (LST city) ?P)
   (PATH1 ?C1 ?CN ?P !<>))
```

It remains to define the other predicates used. To test if ?C2 is not an element of the avoid list, we must have

```
(DEFPRED (ELEMENT obj ?E (LST obj) ?L)   )

(RULE ELEM-NOT (<- (NOT (ELEMENT ?E ?TUP))
                   (THNOT (:= ?TUP !< !?T1 ?E !?T2>))))
```

I leave it as an exercise to figure out why this works.

SAMECONST is more mysterious. Why can't we just say (NOT (= ?C1 ?CN))? The reason is because we want to allow for the case where one or both are variables. In that case, the goal would always fail. Suppose ?C1 is CLEVELAND, and ?CN is a variable. Then the subgoal at this point would be (NOT (= CLEVELAND ?CN)). Technically, this goal should have an infinite number of answers, one for every object except CLEVELAND. But Duck actually tries the equality as a goal; it succeeds, since there is an object = to CLEVELAND, and hence the NOT goal fails. Neither one of these possibilities is what we want. We actually want the goal to succeed except when both ?C1 and ?CN are given, and they are equal. This is done using SAMECONST:

```
(DEFPRED (SAMECONST obj ?X ?Y))

; True if ?X and ?Y are not the same constant
(RULE NOTSAME (<- (NOT (SAMECONST ?X ?Y))
                  (OR (NOT (VARS ?X NONE))
                      (NOT (VARS ?Y NONE))
                      (NOT (:= ?X ?Y)))))
```

5.4. Certainty Factors

As I pointed out at the beginning of this chapter, in a realistic system most deductions end up being "plausible reasoning" by virtue of finding candidate hypotheses or conclusions that some other process then chooses among. Sometimes this sort of reasoning is formalized by attaching numerical weights, called *certainty factors*, to conclusions. In this section, I will discuss one way of doing this.

Without certainty factors, we might have a goal like (DISEASE PATIENT ?WHAT), "Deduce which disease the patient has." Introducing certainty factors might change this to

```
(CERTAINTY (DISEASE PATIENT ?WHAT) ?CF)
```

or, "With what certainty ?CF does the patient have what disease ?WHAT," with answers like,

```
(CERTAINTY (DISEASE PATIENT HANGNAIL) 0.8)
(CERTAINTY (DISEASE PATIENT CIRRHOSIS) 0.5)
```

There are many problems with this idea, such as, What do the numbers mean? Could the patient have *both* hangnail and cirrhosis? I will sidestep all such questions, and focus on the mechanical issues; users can design what they want.

Another side-steppable issue is how to combine weights. Suppose that one rule tells us that the certainty of hangnail is 0.6, and another 0.4. How do we combine them? We could just add them together, but in some cases we require that the numbers stay between 0 and 1, or between -1 and 1, so more elaborate rules have been proposed. I will neglect the details of combination schemes in what follows.

Now we come to an issue we can't sidestep. Suppose that we are given the goal (CERTAINTY ?G ?CF), and we handle it by generating the subgoal

```
(CONTRIBUTION ?G ?CF)
```

In the example above, suppose we get back these answers:

```
(CONTRIBUTION (DISEASE PATIENT HANGNAIL) 0.6)
(CONTRIBUTION (DISEASE PATIENT HANGNAIL) 0.4)
(CONTRIBUTION (DISEASE PATIENT CIRRHOSIS) 0.5)
```

we combine the 0.6 and 0.4 to get 0.8 (by some rule), and take the 0.5 as is. Unfortunately, suppose we had gotten this result instead:

```
(CONTRIBUTION (DISEASE PATIENT HANGNAIL) 0.5)
(CONTRIBUTION (DISEASE PATIENT HANGNAIL) 0.5)
(CONTRIBUTION (DISEASE PATIENT CIRRHOSIS) 0.5)
```

The first two lines may or may not be combinable. Suppose that there was a rule: "If one of the patient's gloves has holes, he is suffering from hangnail with certainty 0.5." If both gloves have holes, then we will deduce the 0.5 twice, and yet (let us suppose) there is not much more reason to believe in this diagnosis given two gloves than given one. (In fact, holes in both gloves may

count *against* hangnail, but let's just assume the two torn gloves prove no more and no less than one.)

This example suggests that we should find all the distinct certainties, and combine those, and count 0.5 only once. Unfortunately, this strategy neglects the possibility that the evidence sources are independent, and just happen to produce the same certainties. One could adopt some tack like requiring that all rules produce different certainties (5.002, 4.998, etc.), but this is ugly. It is more revealing to have each rule mention the "evidence source," and require that these be unique. Hence we revise the CONTRIBUTION predicate to take three arguments: a proposition, an evidence source, and an evidence amount. Then the rules for computing CERTAINTY must add up the contributions from each distinct source.

```
; A term (CONTRIB option evidence-source amount)
; denotes the amount contributed to the option via the evidence source
(DEFFUN (CONTRIB prop ?P obj ?EVIDENCE number ?AMT))

; This predicate appears in user rules, like
; (<- (CONTRIBUTION (DISEASE ?PATIENT HEPATITIS)
;                   JAUNDICE
;                   0.75)
;     (JAUNDICED ?PATIENT))
(DEFPRED (CONTRIBUTION prop ?P obj ?EVIDENCE number ?AMT))

(DEFPRED (CF-SORT (LST (CONTRIB)) ?IN (LST (LST (CONTRIB))) ?OUT))
(DEFPRED (CF-CULL (LST (CONTRIB)) ?IN ?OUT))
(DEFPRED (CF-ADD (LST (CONTRIB)) ?IN number ?TOT))

; Here is the actual program:
(DEFPRED (CERTAINTY prop ?P number ?CF)

   (AND ; Find all options and contributions, and combine them
        ; into a list of CONTRIB terms
        (THFIND ?X (CONTRIB ?P ?EV ?AMT)
                   (CONTRIBUTION ?P ?EV ?AMT))
        ; Sort into one list per different option
        (CF-SORT ?X !<!?A !<(CONTRIB ?P ?EV ?AMT) !?MORE> !?Z>)
        ; For each option, combine all the amounts
        (CF-ADD !<(CONTRIB ?P ?EV ?AMT) !?MORE> ?CF))    )

(DEFPRED (CF-SORT (LST (CONTRIB)) ?IN (LST (LST (CONTRIB))) ?OUT)

   (CF-SORT !<> !<>)
```

```
(<- (CF-SORT !<(CONTRIB ?P ?EV ?AMT) !?R> ?OUT)
    (AND (CF-SORT ?R ?TABLE)
         (THCOND ((:= ?TABLE !<!?A !<(CONTRIB ?P ?EV1 ?AMT1)
                                      !?RC>
                                   !?Z>)
                  ; If there is already a list for option ?P,
                  ; add entry to it:
                   (:= ?OUT !<!?A
                              !<(CONTRIB ?P ?EV ?AMT)
                                (CONTRIB ?P ?EV1 ?AMT1)
                                !?RC>
                              !?Z>))
                  ; else create a new list for that entry:
                  (ELSE
                   (:= ?OUT !<!<(CONTRIB ?P ?EV ?AMT)>
                              !?TABLE>))   ))))

(DEFPRED (CF-ADD (LST (CONTRIB)) ?CL number ?TOT)
; Add up all the contributions for a given option

  (CF-ADD !<> 0)

  (<- (CF-ADD !<(CONTRIB ?P ?EV ?AMT1) !?MORE> ?TOT)
      (AND (CF-ADD ?MORE ?SUBTOT)
           (THCOND ; Ignore redundant or contradictory contributions:
                   ((:= ?MORE !<!?A (CONTRIB ?P ?EV ?AMT2) !?Z>)
                    (:= ?TOT ?SUBTOT))
                   (ELSE
                   ; We assume an associative certainty-factor combiner
                   ; CF-COMBINE:
                    (CF-COMBINE ?SUBTOT ?AMT1 ?TOT))   ))))
```

CF-ADD calls CF-COMBINE to combine two certainty factors. (It also assumes zero is the "null value" for certainty factors, that is, the value assumed for options with no contributions.) CF-COMBINE is filled in by the system designer. If he chooses a nonassociative CF-COMBINE, then the code for CF-ADD will have to be changed. (I leave this as an exercise.)

The most baffling piece of the code above may be this, from the definition of CERTAINTY:

```
; Sort into one list per different option
(CF-SORT ?X !<!?A !<(CONTRIB ?P ?EV ?AMT) !?MORE> !?Z>)
; For each option, combine all the amounts
(CF-ADD !<(CONTRIB ?P ?EV ?AMT) !?MORE> ?CF)
```

This is a bit of classic logic-programming hocus-pocus. CF-SORT takes its first argument as input, and binds its second to the output, a list of lists of CONTRIB terms. CERTAINTY is a failure-driven generator. The first time it succeeds, it binds ?P and ?CF to a hypothesis and its certainty. If a failure propagates back to it (say, if the user repeatedly asks for another answer), then new

bindings of ?P and ?CF are produced each time. Hence what we need to do with the output of CF-SORT is pick one element of its output, a list of CONTRIB terms for one ?P. That ?P becomes the hypothesis to be returned, and all the contributory certainty factors are added to give ?CF. When a later failure occurs, a different element of the output is selected, and so on.

The straightforward way to write this code would be as follows:

```
; Sort into one list per different option
(CF-SORT ?X ?LIST-OF-LISTS)
; Pick one list
(:= ?LIST-OF-LISTS !<!?A ?LIST-FOR-ONE-P !?Z>)
; Figure out which ?P it is for
(:= ?LIST-FOR-ONE-P !<!<CONTRIB ?P ?EV ?AMT> !?MORE>)
; Add up all the contributions
(CF-ADD ?LIST-FOR-ONE-P ?CF)
```

However, we can dispense with the calls to := by simply replacing all occurrences of their left-hand sides with their right-hand sides. So the original call to CF-SORT generates the list of lists, picks one, and extracts ?P from it, all at once. Which version is best I leave to your judgement, but if you are going to read logic programs you had better get used to this sort of thing.

6. REASON MAINTENANCE

6.1. Erasing and Data Dependencies

Any assertion can be erased. Just execute (ERASE *pat*) and it will go away. To erase a rule, either ERASE its pattern, or execute */(FLUSH *rulename*). The name is not evaluated. The rulename is removed from the current workfile. (See Section 7.4.) The function DB-CLEAR erases everything from the database.

ERASE undoes forward chaining. If B was deduced from A, and A is ERASEd, then B is ERASEd automatically. For this to work, each assertion must point to the assertions it was deduced from and those it was used to deduce. The data structures for doing this are called *data dependencies*. [Doyle 79] In particular, B is tagged with a *justification* that mentions A (and all the other facts and rules that were used in conjunction with A to deduce B). Justifications are set up automatically by the forward chainer. Any assertion with sufficient justification is said to be IN, or visible to the Duck database; otherwise an assertion is OUT, or invisible (and eventually garbage collected). (The IN-OUT labels are due to Doyle.)

Actually, erasure propagation is somewhat trickier than that. If B has a supporter independent of A, then it will stay IN after all. For support to be "independent," it must ultimately not depend on B itself. That is, if two assertions support each other, ERASE will not be fooled into keeping them around with no outside support. This situation can arise from rules like this:

```
(RULE LINK-SYM (-> (LINK ?X ?Y) (LINK ?Y ?X)))
```

Asserting (LINK C D) causes (LINK D C) to be asserted and supported. The second assertion causes (LINK C D) to be re-asserted, but the process stops there; Duck does not pursue forward chaining from an assertion that is already in the database. However, the circular data-dependency structure show in Figure 6-1 is set up.

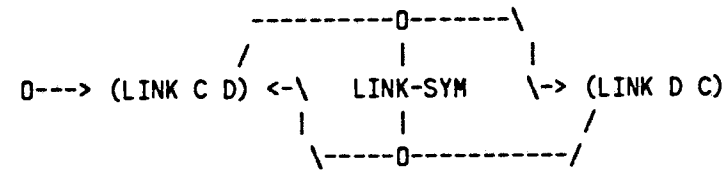


Figure 6-1: Circular Data-Dependency Network

The figure shows a cycle formed by two justifications. The arrow coming into (LINK C D) from the left indicates the independent support for that assertion. The other arrows show how LINK-SYM conjoined with one of the pair of assertions supports the other. If the independent support for (LINK C D) is removed, the whole structure will go OUT.

When (ADD *prop*) is executed, the resulting assertion is supported by a justification derived from the proof record stored in the current ans. For example, in

```
(FOR-EACH-ANS (FETCH '(IS ?X HUMAN))
   (ADD '(MORTAL ?X))   )
```

suppose the FETCH returns the ans {ans ((X SPINOZA)) IS58}, where IS58 is the name of the assertion (IS SPINOZA HUMAN). (See Section 6.3.) (MORTAL SPINOZA) will be supported by IS58 automatically. If (IS SPINOZA HUMAN) is ever ERASEd, so will the new assertion be. The current ans, ANS*, is thus a complete record of the deduction so far, including both variable bindings and the proof that they are correct.

One of the most common (and baffling) bugs in the code of beginning Duck programmers is to forget that ERASE may make the current ans vacuous. Suppose we were using the database to store an assertion of the form (COUNT *k*), where *k* was the current count of some quantity. To increment this count, we might have a function

```
(PROC INCR-COUNT void ()
   (QUACK       ; Don't forget this!
      (FOR-EACH-ANS (FETCH '(COUNT ?K))
         (ERASE '(COUNT ?K))
         (ADD '(COUNT ,(+ ?K 1)))   )))
```

However, instead of counting, this function just erases the current count! Here's why: The FOR-EACH-ANS and the ADD work together to set up the following dependency structure:

```
(COUNT 98) ----O----> (COUNT 99)
```

The ERASE does *not* change the proof record in the current ans, and hence does not alter this justification. By the time the ADD actually occurs, (COUNT 98) is OUT, so (COUNT 99) stays OUT, too.

There is more than one way to fix this problem. One is to use ANS-SUPPORT, discussed below. Another is to move the ADD out of the FOR-EACH-ANS:

```
(PROC INCR-COUNT void ()
   (QUACK
      (LET ((K (GET-COUNT)))
         (ERASE !'(COUNT ,K))
         (ADD !'(COUNT ,(+ K 1)))   )))

(PROC GET-COUNT number ()
   (QUACK
      (FOR-FIRST-ANS (ASSERTION-FETCH '(COUNT ?K))
                     ; FETCH was wrong here anyway
         ?K   )))
```

In fact, we can make it a separate function. Note that I used QUACK twice; the second is

redundant when GET-COUNT is called from INCR-COUNT, but now that we have the function, we will want to use it elsewhere. If in doubt, QUACK. (In case you haven't figure out why ASSERTION-FETCH was required in GET-COUNT, meditate on the impossibility of erasing an assertion that was deduced rather than being explicitly in the database.)

An assertion may be supported by the absence of another assertion. The Lisp special form NEGLECTING is one way to accompish this. */(NEGLECTING *pat -body-*) causes the assertion for pat (which is evaluated and varsubsted) to be added to the proof record of the current ans "negatively." Any ADDs in the body will depend on the assertion for *pat* being OUT.

For instance, since every professor has a Ph.D., with a few exceptions, we can write a rule like this:

```
(LISPRULE PROFPHD -> (OCCUP ?P PROFESSOR)
   (NEGLECTING '(NOT (HAS-PHD ?P))
      (ADD '(HAS-PHD ?P))   ))
```

If we assert (OCCUP POTTER PROFESSOR), then (HAS-PHD POTTER) will also be asserted, supported by (OCCUP POTTER PROFESSOR) being IN, and (NOT (HAS-PHD POTTER)) being OUT. See Figure 6-2.

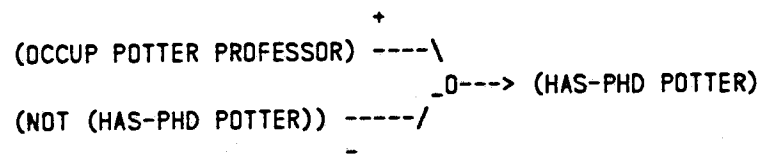


Figure 6-2: A nonmonotonic justification

If the former is ever OUT or the latter IN, then (HAS-PHD POTTER) will be OUT. This support is called a *nonmonotonic* justification.

Another way to accomplish almost the same thing is through the use of the pseudo-predicate CONSISTENT:

```
(RULE PROFPHD (-> (OCCUP ?P PROFESSOR)
                  (-< (CONSISTENT (NOT (HAS-PHD ?P)))
                      (HAS-PHD ?P))))
```

This rule sets up the same data-dependency structure, but only if (NOT (HAS-PHD ...)) is unprovable for the given ?P. Otherwise, the -< fails to generate any answers for its antecedent, and so asserts nothing.

NEGLECTING is a special case of a more general construct: */(ANS-SUPPORT *specs -body-*) evaluates the expressions in *body* with the support of the current ans modified as specified by the *specs*. The modifiers of the ans are other anses or assertion names. In the simplest case, *specs* is

a single ans or name, which is added to the current support. Otherwise, *specs* must be a list beginning with +, =, or -. + means to add stuff to the support. - means to "subtract" it, i.e., make the ans depend on its absence. = means to start from scratch as dictated by the *specs*. Following the +, -, or = sign is a list of items, each of which specifies some assertions to be added positively or negatively to the new ans. Each item is one of the following:

1. An assertion name or pattern: that assertion is taken.
2. An ans: all the assertions in its proof record are taken. Some of them may already be "negative," for instance, if the ans was derived from a deduction involving the CONSISTENT pseudo-predicate. In this case, + will retain the negativity, and - will invert it, making the new ans depend on the *presence* of the assertions in question.
3. A list of the form (< *l*), where *l* evaluates to a list of assertions or anses: each element of *l* is used as above.
4. Another spec beginning (+ ...) or (- ...): the specs are taken recursively. A - inside another - becomes a +.

For instance

```
(ANS-SUPPORT (= (+ (< L)) (- '(NOT (HAS PHD ?X))))
   ...)
```

evaluates "..." supported by the presence of every assertion in the list L and the absence of (NOT (HAS PHD ?X)).

The function SUPPORT can be used to see the complete support for an assertion. It takes as argument a proposition (or an assertion or assertion name), and returns a *support tree* that describes why Duck is justified in believing that the given proposition is IN or OUT. A support tree is a list structure of one of the following forms:

1. (IN: *assertion -support-*)
2. (OUT: *assertion*)
3. (PREMISS: *bead*)

The first form is what SUPPORT returns given an IN assertion. The *-support-* field is a list of support trees enumerating the reasons why this *assertion* is in. The OUT: form is used to mention a justifier whose absence is supporting something else. The PREMISS: form is for nonglobal premisses; the *bead* is from the first data pool containing the premiss. See next section.

For example, given the PROFPHD lisprule described above, (SUPPORT '(HAS-PHD POTTER)) might return

```
(IN: (HAS-PHD99 (HAS-PHD POTTER))
     (IN: (OCCUP85 (OCCUP POTTER PROFESSOR))
          (PREMISS)) ; assuming this is a global premiss
     (OUT: (NOT98 (NOT (HAS-PHD POTTER)))))
```

This is a representation of a tree of justifications going back to premisses (asserted by the user) and OUT assertions.

In Section 7.1, we will look at an interactive "browser" for examining justifications. One difference between that system and this one is that the browser shows you *all* the justifications for an assertion. SUPPORT shows you one *well-founded* justification, and traces it all the way back to its foundations. The former is more likely to be useful in an interactive context; the latter is more likely to be useful for programs to manipulate.

6.2. Data Pools

So far I have talked as if there were one global database, such that a proposition is either asserted in it or not. Actually, the user can build up a set of independent but related "local databases," which are called data pools. They are used to represent hypothetical worlds, time sequences, and packets of special-purpose information.

Consider a problem-solving program. One thing such programs do is plan ahead. They perform deductions based on the state of the world after contemplated actions. For example, say that the database contains the assertions (ON A TABLE) and (ON B TABLE). To model the action of putting A on B, we must ERASE (ON A TABLE) and replace it with (ON A B). Then we can do the deductions we are interested in. Unfortunately, when we are through, we have left the world model in the wrong state. We can get out of it by erasing (ON A B) and adding (ON A TABLE), thus simulating actually undoing the action, but this requires keeping track of every little thing we did, which could be painful and error-prone. Besides, we might want to alternate between processing of two alternative futures; it is out of the question to go through these rituals repeatedly.

The mechanism of data pools saves us the trouble. The idea is to copy the current database and perform the simulated actions on the copy. The original database is undisturbed, and is available for other computations.

Of course, copying an entire database formula by formula is too expensive. But we don't have to do this. Instead, we implement data pools as lists of *beads*, which serve as marks on assertions. If an assertion is marked with a bead that is part of the current pool's list, then that assertion is "visible" in the current pool, or just IN the current pool. Otherwise, it is OUT. Changing the contents of a pool is then a matter of changing the marks on assertions. Every pool has a *characteristic* bead that is used to indicate additions to it. That is, when an assertion is added to a pool, it gets marked with the characteristic bead of the pool.

Consider the example further. Initially, the database consists of one data pool, which has one bead **B1**. POOL-1 = **(B1)**. Associated with this bead are the assertions (ON A TABLE) and (ON B TABLE), plus many more that are irrelevant to the example. To copy this database, just create a brand-new bead **B2**, and add it to the list: POOL-2 = **(B2 B1)**. POOL-2 is a copy of POOL-1,

because **B2** neither adds nor takes away from the data that are already there by virtue of being associated with **B1**. **B2** is the *characteristic bead* of POOL-2. By convention, we put the characteristic bead first in the list of beads for a pool. Now, if we add (ON A B) to POOL-2, it will be tagged with the mark **B2**, not **B1**. As a consequence, POOL-2 contains the assertions (ON A B), (ON A TABLE), and (ON B TABLE). POOL-1 is unchanged.

Of course, we don't want A to be on two things. We also have to be able to erase things from POOL-2 without erasing them from POOL-1. To accomplish this, we allow "negative marks"; more precisely, we allow arbitrary boolean combinations of marks. So (ON A TABLE) must get marked with "**B1 and not B2**." This mark means that the assertion will be visible in any data pool containing bead **B1** and not containing **B2**.

Fortunately, the user can ignore most of this complexity. The current data pool is the value of the global variable DP*. It is initially () when Duck is started. No beads exist, and the only possible marks on assertions are the empty conjunction of beads (always IN) and the empty disjunction (always OUT).

The first step in using data pools is to rebind DP*. Usually one uses DP-PUSH: (DP-PUSH *pool*) is a new pool with a new characteristic bead added to the beads of *pool*.

To make real use of DP*, you must use PREMISS instead of ADD where appropriate. ADD causes its argument to depend on the proof record stored in the current ans. PREMISS in addition marks the assertion it creates with the characteristic bead of the current data pool, DP*. Everything deduced from assertions marked with such a bead will also be so marked. If DP* is (), then ADD and PREMISS have exactly the same effect.

(ERASE *pat*) then does the following. It finds the assertion with the pattern *pat* in the database, and, for each of its justifications, it adds **not B** to it, where **B** is the characteristic bead of DP*.

With these tools, we can write a function that takes a data pool representing a temporal situation and returns a new data pool representing the new situation after applying an action with addlist A and deletelist D:

```
(FUNC NEWSIT datapool (datapool PREVSIT (LST sexp) A D)
   (LET (datapool (NEW (DP-PUSH PREVSIT)))
      (BIND (datapool (DP* NEW))
         (LOOP FOR ((P IN D))
            (ERASE P)   )
         (LOOP FOR ((P IN A))
            (PREMISS P)   )
         NEW   )))
```

PREMISS, ERASE, and FETCH do their deductions in the current data pool DP*. To do a FETCH in

another pool, just rebind DP*. To test what object BLOCK1 is on after performing an action, you do something ike this:

```
(:= DP2 (NEWSIT DP1 ...))
 . . .
(BIND (datapool (DP* DP2))
  . . . (FETCH '(ON BLOCK1 ?X)) . . .)
```

Data pools are just data objects. They can be stored in data structures for later use. One thing to be careful about is that they are *not* automatically garbage collected. If you create a lot of them, you should be sure to get rid of them when they are no longer needed, using (DP-KILL *data-pool*). This function eliminates every reference to the characteristic bead of this data pool, garbage-collecting assertions that are thereby made OUT everywhere. Avoid using a killed data pool; no catastrophic error will occur, but its contents may be unpredictable.

Sometimes you need somewhat finer control over justifications than provided by ERASE. UNPREMISS and UNADD undo PREMISS and ADD respectively. That is, they each delete at most one justification, the one that would be created by PREMISS or ADD. ERASE is more heavy-handed, since it clobbers *every* justification.

Repeated use of DP-PUSH results in a tree of data pools, as in this example:

```
POOL-1 = (B1) contains (ON A TABLE) (ON B TABLE)
   POOL-2 = (B2 B1) contains (ON A B) (ON B TABLE)
      POOL-4 = (B4 B2 B1) ...
   POOL-3 = (B3 B1) contains (ON B A) (ON A TABLE)
  .
  .
```

If an assertion is a local premiss, then one of its justifiers is the characteristic bead of the data pool it is a premiss of. That is, beads act like assertions in justification networks. The SUPPORT function will print such beads out as subtrees of the form (PREMISS: *beadname*). Beads appear in other places too, notably when walking through assertion justifications. (See Section 7.1.) If you don't want them to be seen, set BEADS-SEE* to ().

Modelling temporal sequences is not the only use of data pools. We might model people's belief structures by keeping one bead with facts almost everybody knows, and associating with each person a pool with two beads, this universal bead and an idiosyncratic bead for his own beliefs. For example, to create a new data pool, HENRYS-HEAD, containing everything everyone else believes except for the lamentable opinion that Henry isn't very bright. we could execute:

```
(:= HENRYS-HEAD (DP-PUSH DP*))
(BIND ((DP* HENRYS-HEAD))
   (ERASE '(INTELLIGENCE HENRY LOW))
   (PREMISS '(INTELLIGENCE HENRY HIGH))   )
```

Along the same lines, if there is some set of facts shared by everyone in a certain group, but not everyone in the world, we can make these facts into a bead and include it in the data pools of just those people.

The functions for manipulating data pools are these:

- (DATAPOOL *list-of-beads*): Create a datapool with these beads. The list cannot be empty.
- (BEAD): Create a new bead
- (DP-BEADS *datapool*): Retrieve the beads from the pool, characteristic bead first.
- (DP-PUSH *datapool*): Same as (DATAPOOL (CONS (BEAD) (DP-BEADS *datapool*))), except that the superior is accessible via DP-SUP.
- (DP-SUP *dp*): If *dp* was produced by DP-PUSHing another datapool, returns that one, else ().
- (DP-KILL *datapool*): Cleans up when you are through with a datapool. Flushes all premisses created in that pool. The pool should not be used again.

6.3. Details of Reason Maintenance

Duck's reason maintenance system has more power than it really needs for keeping track of Duck-style deductions. However, applications have been found for that power in several domains, including temporal reasoning. [Dean 83] In this section, for the dedicated, I describe how to get access to a bit more of the full system.

The RMS keeps track of abstract "assertion-like" beliefs, called *ddnodes*, linked by structures called *justifications*. The content of these nodes is unintelligible to the RMS. For it, a *belief state* is a set of ddnodes plus a labeling that specifies which ddnodes are IN and which are OUT. Its job is to manage transitions among belief states, by changing the labeling until it is admissible. (See [McDermott 83b] for the criteria of admissibility.) All Duck has to do is identify assertions with ddnodes in order to get RMS to be its agent in deciding which assertions are currently IN.

For this identification to be possible, every occurrence of a formula must be associated with the *same* ddnode. When a proposition such as (ON A B) is added, the system must create a structure called an *assertion* and mark it IN. The next time (ON A B) is seen, during fetching or adding, it must be identified with the same assertion. When (ON A B) is erased, it is this assertion whose ddnode's justifications are altered to make it go OUT. One way to think of an assertion is as a

kind of "super-atom." When Lisp sees a string of characters in its input, it makes it into a print name and associates it with an atom. The next time it sees the same characters, it winds up retrieving the same atomic symbol by using a symbol table. Similarly, if the user attempts to ADD the same relationship twice, the system will find the already-known assertion, using a discrimination tree. So assertions are "uniquified" lists in the same sense that atoms are "uniquified" strings.

In the actual program, an assertion looks like a two-list of the form (*name proposition*). For instance, (ON A B) might become (ON27 (ON A B)). We have already seen these names; they show up in anses. They are useful because they are more compact than the full assertion. If you want to make an assertion about an assertion, use its name.

The following Lisp functions are provided for going back and forth between assertions, propositions, and names. If you need to call these from Duck, write a lisprule for the predicate (ASSERTION-NAME ?NAME ?PATTERN).

- (ANAME *assertion*): returns the name of the assertion. Note: this symbol is either an ordinary atomic symbol, or what is called an "uninterned symbol." Such a symbol will not be recognized on type-in. Hence if, during debugging, you want to look at a particular assertion name (e.g., to examine its property list), you should set a variable to what ANAME returns, and manipulate that rather than trying to access '*name*. In general, if the assertion was defined using RULE or LISPRULE, its name will be the one you gave; otherwise, it will be made up by the system, and uninterned. (The system uses UNIQUIFY, described below, to make up these names.)
- (APROP *assertion*): returns the proposition of the assertion.
- (NASS *name*): returns the assertion given the name.
- (UNIQUIFY *proposition*): returns the assertion given the proposition. If there is no assertion yet, a new one will be created and named using an "uninterned" (unreadable) atomic symbol of the form *pred-number*, where *pred* is the main predicate of the proposition, and *number* is a different number for each such assertion.

  It is important to realize that creating a new assertion is not the same as asserting it. Once the assertion exists, subsequent calls to UNIQUIFY will find it, but FETCH will not, until it is asserted by someone. If an assertion has been uniquified previously, then ADD will return the old assertion.

  If an assertion is garbage collected (because it has been erased from every datapool), then pointers to it will be invalid. Later calls to UNIQUIFY or ADD will create a new, distinct assertion.

Note: in most places in this manual, no distinction is made between assertions and their propositions, the term "assertion" being used for both. Hopefully the meaning will be clear. (The same confusion between symbols and their print names is very convenient in Lisp manuals – until you get to the chapter on symbols!)

Now back to the reason maintenance system. As I said, it conceives its job as labeling ddnodes

IN or OUT. In addition, it must be able to tell interested parties about transitions involving particular nodes. For instance, the Duck indexer would like to know when an assertion has gone so far OUT it can be garbage-collected. We can avoid having the indexer sweep through the database periodically looking for OUT assertions, by arranging for the RMS to tell the indexer when a node has gone OUT. The arrangement depends on giving every ddnode a *signal function* that will be called when that ddnode changes state, from IN to OUT or vice versa.

In Duck, every assertion has the same signal function, which handles garbage collection (and -> tracing; see Section 7.3).

Sophisticated use of the RMS begins when you see the need for a more specialized signal function. For example, I mentioned in Section 3.6 that THFIND, TOTAL, and other functions that enumerate something in the database do not set up fully correct data dependencies for their conclusions. If you execute

```
(FOR-EACH-ANS (FETCH '(THFIND ?L ?X (US-PRESIDENT ?X)))
   (ADD !'(ALL-THE-PRESIDENTS: ?L))   )
```

in 1978, you will add the assertion (ALL-THE-PRESIDENTS: !<CARTER WASHINGTON LINCOLN ...>) to the database. When (US-PRESIDENT REAGAN) is then added, this assertion should go away and be replaced by a new assertion with the up-to-date list. But, in Duck, it will not. I thought the overhead was too costly for the expected frequency of use.

If you absolutely need this capability, the first step is to create an assertion of the form

```
(RULE PRES-UPDATER
    (MAKES-OUT-OF-DATE (US-PRESIDENT ?X)
                       (ALL-THE-PRESIDENTS: ?L)))
```

where MAKES-OUT-OF-DATE is defined thus:

```
(DEFPRED (MAKES-OUT-OF-DATE prop ?KILLER ?TARGET))

(RULE OUT-OF-DATE-DEMON
      (-> (MAKES-OUT-OF-DATE ?KILLER ?TARGET)
          (-> ?KILLER
              (-< ?TARGET (DELETE ?TARGET))))   )
```

(This code uses -<, described in Section 4.3, to switch to backward chaining, binding the variables in ?TARGET; the pseudo-predicate DELETE, described in Section 3.6, then erases that instance of ?TARGET.) (Note: The same idea would be more costly to apply for the general case of THFIND, TOTAL, etc. Do you see why?)

In some cases, the mere erasure is enough. In others, the RMS still needs to remind Duck to recompute the quantity in question. Such a message is sent using a signal function. It is considered bad form to set the signal function of an assertion directly, because Duck considers that its property. It is better instead to create a new ddnode justified by the assertion, whose

status will always undergo the same transitions as the assertion's, and put the desired signal function there.

The most common way to do this is with

*/(IF-ERASED *pat -body-*)

which associates with the assertion for *pat* a signal function that executes *body* whenever that assertion is erased. In our example, we can change the code to read:

```
(PROC ADD-ALL-THE-PRESIDENTS void ()
   (QUACK
      (FOR-FIRST-ANS (FETCH '(THFIND ?L ?X (US-PRESIDENT ?X)))
         (ADD !'(ALL-THE-PRESIDENTS: ?L))
         (IF-ERASED !'(ALL-THE-PRESIDENTS: ?L)
            (ADD-ALL-THE-PRESIDENTS)   ))))
```

QUACK is essential here; when the body of the IF-ERASED is executed, the ans in effect will be the one in effect when the IF-ERASED itself was executed, and hence will have a binding of ?L and a proof record that depends on the old ALL-THE-PRESIDENTS: assertion. QUACK eliminates both of these. IF-ERASED's first argument is evaluated and varsubsted, just like the arguments to ADD, FETCH, and MATCH.

There is no IF-ADDED (though perhaps there ought to be). However, rules of the form (-> *p* (CALL *f*)) come close. Normally, if (-> *p q*) is in the database, and *p'* is added, where *p'* unifies with *p*, then the corresponding *q'* is created, and a justification is created linking these three formulas. The next time *p'* is added, the rule will not be used again. Instead, the label on the justification will be changed if necessary. But if *q* is of the form (CALL *f*), then a signal function is set up, associated with *p'* in the same way IF-ERASED uses, so that *f* is called whenever *p'* changes from being OUT of the the current data pool to being IN.

This asymmetry reflects the fact that users' CALLed functions normally have side effects that are completely unknown to Duck, whereas all the effects of an ordinary forward chain can be handled the first time it occurs by setting up structures for the RMS. From then on, no unification is required. In some cases, your function will have similar properties. That is, its effect will be to set up data dependencies that take over all the work. Here it would be a nuisance to have this function called every time the assertion is added to a new data pool. To avoid that, wrap FIRSTIME around the call: (-> *pattern* */(FIRSTIME (CALL (*fun* ...)))). Now the function *fun* will be called only once for each instance of the *pattern*, the first time it is added to the database in any data pool.

7. THE DUCK ENVIRONMENT

7.1. Running Duck: Lisp Mode, Duck Mode, Walk Mode

To run Duck, type DUCK or something like that at your operating system. This will leave you in a Lisp with all the Duck subroutines defined. You can run any Lisp function, and use ADD, FETCH, etc. to call Duck. In fact, to define Duck predicates and rules, you will probably use Lisp file-loading utilities (like DSKLAP in Nisp) to load files full of DEFPREDs and RULEs. These predicates and rules can these be used by Lisp programs using ADD and FETCH.

Eventually, however, you will probably want to start a deduction "by hand" and inspect the results, for debugging if nothing else. Here the DUCK function becomes very handy. As explained in Section 3.1, typing (DUCK) puts you into a "read-deduce-print" loop, where what you type is interpreted as a deductive goal (if it cannot be interpreted as a Lisp form or a special command). In this loop, you are said to be in *Duck mode*. The prompt is "?>" or "g?>". The latter form means that there is no "current goal"; after you type a goal, it becomes current, and Duck will look for solutions to it. If there are none, it will print "Nothing." Otherwise, it will print the first one it finds. Typing A will cause it to print the next answer, until there are no more. Here is an example (the A's are typed by the user):

```
g?>(PART ?X CIRCUIT7)
  (PART OPAMP23 CIRCUIT7)
  X=OPAMP23
?>A
  (PART DETEC3 CIRCUIT7)
  X=DETEC3
?>A
No more
...
```

There are other commands besides A. R causes the current or previous goal to be restarted. G <*form*> causes *form* to become the new goal. If *form* is a symbol, its Lisp value becomes the new goal. So if you set G1 to (PART ?X CIRCUIT7), the command G G1 tries the same goal as in the example.

Two commands cause "walk mode" to be entered, to examine the reasons for a conclusion, or the rules that could have been used to arrive at a conclusion. Before getting to these two cases, let me say a few words about walk mode in general.

Many of the entities in Duck, both dynamic and static, may be thought of as graph structures. These include goal hierarchies during deductions, and data type hierarchies. The "walker" or "browser" is a sort of editor that allows you to walk around these graphs, inspecting and in some cases altering them. When the walker has control, Duck is said to be in "walk mode." It will

prompt with "w>." You can type anything Lisp understands, plus commands to move around in the graph and change it.

In "walk" mode, the system displays a current "focus of attention," a goal, hypothesis element, belief, or the like. This will have a short form and a long form, or "tableau." The short form is for recognition when you return to a "focus." The long form is more informative, and will consist of helpfully formatted material, including in particular numbered subparts. This material often points off to other parts of the graph. For instance, a tableau describing why an assertion is believed may mention some supporting beliefs. After staring at such a tableau, you will often want to move the focus of attention to a pointed-at thing, such as a supporting belief. As you move through the graph in this way, it is important to be able to get back to where you've been. The walker keeps track of two different records of where you've been: the *stack* and the *trail*.

The stack records an *ad hoc* view of the graph as a tree. The root is where you began. Every time you move to a new part of the graph, you have *ipso facto* moved "down." If you go back where you've been, you have moved "up." The stack records "which way is up," that is, how to go back to the places you came from.

The trail records where you have been chronologically. When you move up, you may want to jump back to where you just were, even though that is now "down."

The walker's state is global. You are always "in" it. There are Lisp commands for changing the current focus of attention, and they operate on this global state. "Walk mode" is just a convenient interface to these Lisp commands. For instance, the command (DN *-instructions-*) goes down from a node. In walk mode, you can just type the *instructions* on a line, without parentheses.

From Duck mode, walk mode can be entered by typing:

1 E [<*fact*>] *or*
WHY [<*fact*>]

Enter walk mode looking at a "why" tableau for the <*fact*>. If the <*fact*> is omitted, it defaults to the last answer.

2 F [<*goal*>] *or*
WHYNOT [<*goal*>]

Enter walk mode looking at a "whynot" tableau for the <*goal*>. If the <*goal*> is omitted, it defaults to the current goal.

A "why" tableau gives the reason why Duck believes something. Suppose the database contains these rules:

```
(RULE UNCLE1 (<- (UNCLE ?X ?Y)
                 (AND (PARENT ?Z ?Y)
                      (BROTHER ?X ?Z))))

(RULE PARENT1 (PARENT SALLY JOAN))
(RULE BROTHER1 (BROTHER FRED SALLY))
```

and suppose you type the asterisked inputs in this scenario:

```
(DUCK)                          *
g?>(UNCLE ?WHO JOAN)            *
(UNCLE FRED JOAN)
WHO = FRED
?>E                             *
Backward chaining from
   1 UNCLE1 (<- (UNCLE ?X ?Y)
                (AND (PARENT ?Z ?Y) (BROTHER ?X ?Z)))
   the conjunction of
      2 PARENT1
      3 BROTHER1
w>
```

The `w>` signals that you are now in walk mode, looking at a proof record. It shows that the answer was arrived at through backward chaining, as expected. If you type `Q`, you will return to duck mode, where you can type `A` to find another uncle, and so forth. If you type 1, you will be looking at the justification for the rule `UNCLE1`:

```
1 UNCLE1
  (<- (UNCLE ?X ?Y) (AND (PARENT ?Z ?Y) (BROTHER ?X ?Z)))
**IN**
Justification:
Global premiss
w>
```

Here we are not looking at a proof record, but an assertion in the database. The symbol `**IN**` means that it is currently visible. (See Section 6.1.) The reason it is visible is because it is visible everywhere; it is a global premiss, created by `RULE`. If it had been asserted using `ADD` or forward chaining it would have had a longer list of justifiers. (Included in these justifiers may be cryptic entities of the form "`bead<n>`". These indicate which data pools the believed thing is in. If you don't want to see them, set `BEADS-SEE*` to `()`.)

This example is fairly typical of walk mode. Typing a number causes attention to be moved "down" to the entity with that number in the current tableau. Often there are other commands specific to a tableau; if you type `HELP` or `\?` you will be told about them. For instance, while looking at an assertion tableau, if you type `ENGLISH`, a crude pseudo-English version of the assertion will be printed. (See Section 7.2.)

The user can type any of several standard things in "walk mode":

- A positive number: Changes the focus of attention to the numbered part of the thing displayed.
- Zero: This causes the long form of the current tableau to be redisplayed.
- A negative number: Causes the focus to be moved back to a previous point in the stack. After moving down by typing *n*, typing -1 will get you back to where you just were; typing -2 will get you back to the place above that; and so on.
- (*number* ...) : Change the *number*th element. This is not always meaningful. (*number* EDIT) may invoke the appropriate editor to change the element. This is all highly idiosyncratic, and depends on whether in-core editing is operative.
- UNDO: Undo the last change.
- DUCK: Enter duck mode. Type "Q" to get back.
- EDIT [OBJECT | TYPE] *symbol* : Stay in walk mode, and move attention to the given symbol. If it is an object, display its class (CONSTANT, FUNCTION, or PREDICATE). If it is a type, display its supertype and subtypes.
- STACK *n* : Display the walk stack (see above). Prints out *n* short-form nodes, preceded by negative numbers. Typing the appropriate negative number will take you up to the node for that number.
- (WP *var*): Remember the current walk state as the value of the variable. This is called a *named walk point.*
- (JW): Jump back to the chronologically most recent walk point. Repeated executions of (JW) cycle through the 8 previous walk points. These are saved any time you move from one tableau to another.
- (JW *var*): Jump back to the named walk point *var.*
- TRAIL: Display the walk trail, including the 8 most recent walk points, and any named points.

Anything else is either a Lisp form, which is handled as usual, or a tableau-specific command.

A "whynot" tableau displays all the ways Duck *could* have deduced a goal, successful or not. Here is what the tableau might look like for the goal (INPUT-SWITCH ?X MOD5):

```
Goal: (INPUT-SWITCH ?X MOD5)
Rules:
1  RULE36
2  RULE54
```

This layout means that RULE36 and RULE54 were the only two rules whose consequents unified with this goal. If you type the number of one of these rules, you will see a list of all the ways this rule might have succeeded, displayed in a third kind of tableau, a "rule-attempt" tableau. If you type 1, for example, you will be looking at the rule-attempt tableau for RULE36. It records how far RULE36 would get in deducing this pattern. It shows all the unsuccessful paths through the rule (and, if you want, the successful ones). Its format is as in this example:

```
1 Rule: (<- (INPUT-SWITCH ?X ?MODULE)
            (AND (PART ?X ?MODULE)
                 (IS ?X SWITCH)
                 (CONNECTED ?X (INPORT ?MODULE))))
   [Unification: (ENV ((module mod5)))]
Blocked subgoals:
2 (IS AMP34 SWITCH)
3 (CONNECTED SW3 (INPORT MOD5))
No successful paths through this rule.
```

Under "Blocked subgoals" is shown a list of attempts to make it through this rule. In the example, two answers were found to (PART ?X MOD5): ?X=AMP34 and ?X=SW3. AMP34 is not a switch, so that answer died when (IS AMP34 SWITCH) was tried. Duck was able to prove (IS SW3 SWITCH), but it died on (CONNECTED SW3 (INPORT MOD5)).

If you see the problem with your rule, you can type 1 and edit it (if the in-core editor is loaded; if not, go edit it in the file). Otherwise, you can go down to the appropriate blocked subgoal, by typing its number. In this case, it is clear that (IS AMP34 SWITCH) failed as it should, so you type 3 to find out why the second attempt failed.

What you will get by typing the number of a failed subgoal is a "failed rule" tableau, a detailed record of how this attempt failed. On our continuing example, it would look like this:

```
1 Rule: RULE36
Why this rule has succeeded so far:
   2 SW3-PART (PART SW3 MOD5)
   3 SW3-TYPE (IS SW3 SWITCH)

Remaining goals of this rule:
4 ((CONNECTED SW3 (INPORT MOD5)))
```

Sections 2 and 3 of this tableau describe the proof records for the successful subgoals of RULE36. In this instance each goal was satisfied by a single assertion, but in general there may be a full proof tree in each of these positions. The last entry (here, number 4) is a list of all the remaining goals of the rule. If you type that number, you will be given a "whynot" tableau for the first of these goals, that is, a list of all the rules that could have been used to deduce it.

A rule-attempt tableau shows all the ways Duck could find to get part way through a rule. Many of these attempts will be successful. In this case, rather than the line

```
No paths through this rule were successful
```

you will be asked the question

```
-- Want to see the successful paths? --
```

while the tableau is being displayed. Usually you just type N. If you type Y, you will see the

variable bindings obtained for each successful path.

It is easy for Duck to make a mistake in stepping through a rule. One pitfall is that there may be too many ways to try. If a conjunct in the antecedent of a rule has more than ten solutions, the excess ones may be ignored. If you expect many solutions, and you see only a few, you may want to set SHOVENUM* to something greater than 10. Alternatively, you might try looking with a new goal. E.g., if the goal (FOO ?X ?Y) generates many partial solutions, and your ultimate aim is to see why (FOO ?X B) wasn't found, try again with (FOO ?X B) as the goal. To make this easier, you can type (in Duck mode)

```
F (FOO ?X B)
or
WHYNOT (FOO ?X B)
```

F or WHYNOT with no argument means, "Use last typed goal."

One of the most important things you can do from a whynot tableau is define a new rule, usually after you have verified that the existing rules displayed are inadequate. Type NEW RULE, and you will be in the Lisp editor, with the skeleton of a rule to infer the pattern already begun.

A new rule is usually called for when a deduction that should have happened didn't. The dual situation, when a deduction that should not have happened did, is handled from within a why or rule-attempt tableau. If you type 1 (i.e., the number of the rule shown), you will be able to edit an existing rule, typically adding clauses so that the rule no longer succeeds in some situation.

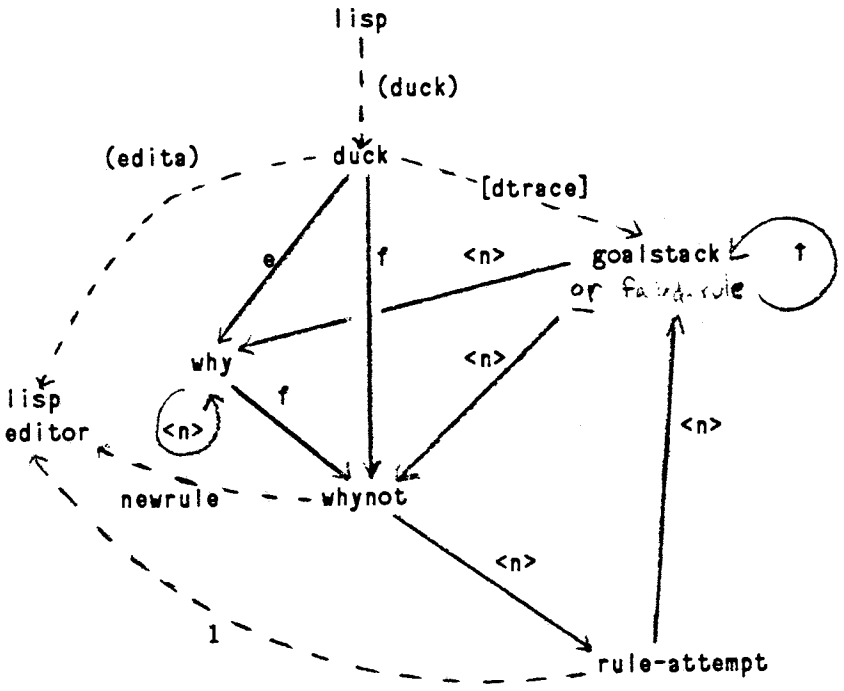
Of course, if the in-core editor is not loaded, you will have to find the relevant files to do these manipulations in.

WHYNOT mode and its relatives are not of much help in tracking down infinite recursions among deductive rules, whose only symptom is that a deduction just goes away and never comes back. If this happens, do not try to attack the problem by using WHYNOT mode. Chances are, in trying to see where some rules breaks down, the system will stumble upon the same infinite recursion and never return. To track down such bugs, one must use more dynamic tools, such as the trace package, which can show a deduction unfolding. See Section 7.3.

The transitions between the various modes of Duck, including the various sorts of walk-mode tableaux, are summarized in Figure 7-1. Each node is a mode or tableau type. The arcs are labeled with a description of the command that gets you from the mode at one end to that at the other. If the arcs is solid, you get back by going "up" (typing a negative number). Otherwise, you get back by typing Q or OK, thereby restoring the previous mode.

MODE TRANSITIONS:

Solid arcs: get back by going "up" (typing a negative number)
Dotted arcs: get back using "q"



In addition:
Type 'duck' in any walk mode to get to duck mode;
'q' to get back
In all interactive Lisp systems, typing an interrupt character (e.g., ↑C) will open a read-eval-print loop, from which execution can be continued (e.g., by typing (return)).

Figure 7-1: Interactions modes and transitions among them

7.2. Pseudo-English Generation

The Duck environment makes it possible to examine the reasons for Duck's conclusions, in the form of rules that contributed or did not contribute to those conclusions. Unfortunately, these rules are written in predicate calculus, which is more perspicuous than many notations, but not transparent to some end-users of the system. If the system builder is willing to work at it, he can tell Duck how to express these rules in something like English.

For example, consider the rule-attempt tableau of Section 7.1, which starts:

```
1 Rule: (<- (INPUT-SWITCH ?X ?MODULE)
            (AND (PART ?X ?MODULE)
                 (IS ?X SWITCH)
                 (CONNECTED ?X (INPORT ?MODULE))))
   [Unification:  ...
Blocked subgoals:
...
```

If the end-user of the system is not conversant with the syntax of predicate calculus or the particular predicates used here, he might prefer to see the following version of the rule, which he gets by typing ENGLISH in walk mode:

```
If a component is part of a module
  and the component is a SWITCH
  and the component is connected to the input port of the module
then the component is the input switch of the module
```

Even someone who is reasonably familiar with the notation might find this kind of comment handy.

To have Duck be able to produce it, the system builder must give a "skeleton" for each predicate and function. This is done in one of two ways: either in the DEFPREDs and DEFFUNs, in a clause beginning =:, or interactively, using the type editor as described in Section 7.5. Here is what the first method might look like:

```
(DEFDUCKTYPE device obj)
(DEFDUCKTYPE component device)
(DEFDUCKTYPE module device)
(DEFDUCKTYPE class obj)
(DEFDUCKTYPE port obj)

(DEFPRED (INPUT-SWITCH component ?C module ?M)
   (=: ?C " is an input switch of " ?M)   )

(DEFPRED (PART device ?C1 ?C2)
   (=: ?C1 " is part of " ?C2)   )
```

```
(DEFPRED (IS obj ?X class ?C)
   (=: ?X " is a " ?C)   )

(DEFPRED (CONNECTED device ?X port ?Y)
   (=: ?X " is connected to " ?Y)   )

(DEFFUN port (INPORT module ?M)
   (=: "the input port of " ?M)   )

(RULE RULE36 (<- (INPUT-SWITCH ?X ?MODULE)
                 (AND (PART ?X ?MODULE)
                      (IS ?X SWITCH)
                      (CONNECTED ?X (INPORT ?MODULE)))))
```

These skeletons are combined with the *control* -- the formula being Englishified -- to produce something resembling English. Each atomic formula in the control gives rise to a single clause in the result (this alone is sufficient to make what is generated rather stilted). The skeleton of the predicate is combined with the control and the *template* -- the form specifying the types of arguments the predicate expects. For instance, in outputting (CONNECTED ?C MOD5), the control is this formula itself, the template is (CONNECTED device ?X port ?Y), and the skeleton is (?X " is connected to " ?Y).

The system produces its output by filling in the skeleton with noun phrases for the variables, which it obtains by matching the template against the control. In the example, ?X matches ?C and ?Y matches MOD5. (This is not unification, but ordinary textual matching; the material in the control is inert data.) If a variable matches a non-variable, then the English generator is called recursively on it. In the usual case, where it is a single symbol, this symbol is just output; here that is what happens to ?Y, which is output as MOD5.

If a template variable matches a predicate-calculus variable, as happens with ?X matching ?C here, then the system produces a noun phrase using type information from the templates. It knows that ?X must be a device, so it outputs a noun phrase like "a device." In the large example above, the first time ?X is seen it occurs at location in the template for PART that gives it type component, and so it becomes "a component" in its first occurrence, and "the component" subsequently.

If there is more than one variable that would have the same noun phrase, then they are identified with ordinal phrases. For example, here is how the UNCLE rule of Section 7.1 could be defined and Englishified:

```
(DEFDUCKTYPE person obj)

(DEFPRED (PARENT person ?P1 ?P2)
   (=: ?P1 " is a parent of " ?P2)   )

(DEFPRED (BROTHER person ?P1 ?P2)
   (=: ?P1 " is a brother of " ?P2)   )

(DEFPRED (UNCLE person ?P1 ?P2)
  (=: ?P1 "is an uncle of " ?P2)   )

(RULE UNCLE1 (<- (UNCLE ?X ?Y)
                 (AND (PARENT ?Z ?Y) (BROTHER ?X ?Z))))
```

Englishification:

```
If a person is a parent of another person,
   and another person is a brother of the first person,
then the third person is an uncle of the second person.
```

Here ?Z becomes "a person" on its first occurrence, "the first person" on later ones, and so on.

This is all you need to get started producing pretty good English. To see the results, just type ENGLISH at any tableau in walk mode where it seems appropriate.[4] To call the English generator from Lisp, execute (PATSAY *proposition*).

If no skeleton is provided for a predicate, then Duck will resort to clumsy locutions such turning (FOO A B) into "the relation FOO holds for A and B." You might as well get in the habit of always providing some skeleton, since it provides a valuable comment. If you find it hard to express a predicate in English, that may be a symptom of a poorly conceived predicate (or of logic programming, which often requires predicates with little declarative meaning).

If you want more polish, you have to know a few more things. First, how English is produced for a proposition or variable depends on the context in which it is found. This context is either affirmative or negative, and is either in the antecedent of an implication or not. The antecedent of an implication is an implicit negative context, but must be treated differently. A free variable in an affirmative context refers to any object. Hence in a a formula like (IF (HIPPIE ?X) (LOVES ?X ?Y)), the variable ?Y must become the noun phrase "every person," as in "If a person is a hippie, then the person loves every person." Most variables do appear in the antecedent of an implication first, and hence get the ordinal treatment described above.

A proposition that occurs in a negative context must be negated. Without guidance, Duck will

[4]If the global variable ENGLISH* is set to DEFAULT, then the English will always be dumped out (in addition to the predicate calculus). If this variable is (), the English-generation feature will be turned off completely.

simply put "it is not the case that" in front of the proposition. So (NOT (LOVES ?X FRED)) would become "It is not the case that the person loves FRED." To produce "The person does not love FRED," the system builder should define LOVES thus:

```
(DEFPRED (LOVES person ?X ?Y)
   (=: * ?X (NOT? " love") ?Y)   )
```

The * signals that negation is handled by the skeleton. The NOT? construct (described below) explains exactly how. In an affirmative context, (NOT? " love") becomes " loves "; in a negative context, " does not love " is produced instead.

The format of a skeleton is (=: [*] *-elements-*), where

- If the "*" is present, it means that the template explains how to negate itself. If absent, the template will be negated by putting "it is not the case that" in front of it.
- Each *element* is one of the following:
  1. A string or symbol -- it is inserted in the output (unless it has a special meaning)
  2. A variable -- its value (the part it matched in the control) is output. If its value is a variable, it is output as a noun phrase such as "a *type*," "the *type*," "another *type*," or "the *n*th *type*," unless it occurs for the first time outside the antecedent of an implication, when it is generated as "every *type*." The noun phrase may be more finely tuned using NP (see below).
  3. The symbol NOT? -- ignored in "affirmative" contexts, and turns into "not " in "negative" contexts. E.g., (GREEN ?X) could have skeleton (=: * ?X " is " NOT? "green").
  4. (NOT? *verb* [*ending* "s "] [*auxiliary* " does ")] -- where *verb*, and, if present, *ending* and *auxiliary* are strings -- in affirmative contexts, turns into *verb-ending*; in negative contexts, turns into "*auxiliary*-not-*verb* ".
  5. (NP *noun* [*-prefix-stuff-*] *?var* [*-suffix-stuff-*]): If *?var* is bound to a variable, the noun is used instead of its actual type to produce noun phrases of the form "a *noun*", "the *noun*", etc. If the noun is ?(), then no noun phrase will be used, but the actual variable will appear. If *?var* matches a non-variable in the control, then its value is output sandwiched between the prefix and suffix.
  6. (*-tests-* => *-skel-*) -- If all of the *tests* are true, includes *skel*, else skips it. A test is either a Lisp form involving argument vars, or is one of the symbols NEG, AFFIRM, IN-ANTE, or IN-CONSE. These symbols test whether the context is negative, affirmative, in the antecedent of an implication, or not in the the antecedent of an implication, respectively. NOT? is equivalent to

     (NEG => "not ").

  7. (SELECT *-clauses- -else-*) -- Each clause must be of the form (*-tests-* => *-skel-*). Each is tested until all its *tests* come out true, and the corresponding *skel* is included. If the final clause is of the form (=> *x -skel-*), where *x* is not a list with => at the top level, then the parens and => can be omitted; in other words, if the SELECT form ends with some stuff that cannot be mistaken for a

list of clauses, that stuff is treated as an "else" clause. (NOT? *verb*) is equivalent to

```
(SELECT (AFFIRM => verb "s ")
        (=> " does not " verb " ")   )
```

or, equivalently,

```
(SELECT (AFFIRM => verb "s ") " does not " verb " ")
```

The tests preceding a => can include Lisp forms. Within those forms, Duck variables will evaluate to the material they matched in the control. Hence to test whether a variable matched another variable (in cases where NP is inadequate), you can use (IS mvar *?variable-from-skeleton*), as in this example:

```
(DEFPRED (FEED module ?X ?Y)
   (=: * (NP "source" "module" ?X)
         (NOT? " feed")
         (SELECT ((EQUAL ?X ?Y) => "itself")
                 (=> ((NOT (IS mvar ?X)) => "the target ")
                     ?Y)   ))   )
```

which causes the following control patterns to produce the following outputs:

```
(FEED AMP1 AMP1) ---> "module AMP1 feeds itself"
(FEED AMP1 SPEAKER2)
    ---> "module AMP1 feeds the target SPEAKER2"
(NOT (FEED AMP1 AMP1)) ---> "module AMP1 does not feed itself"
(-> (NOT (FEED ?X ?X)) (FEED ?X AMP1))
   ---> "If a source does not feed itself,
        then the source feeds AMP1"
(-> (FEED ?X ?Y) (FEED ?X ?X))
  ---> "If a source feeds a module,
       then the source feeds itself"
```

7.3. The Trace Package

Rules and predicates may be traced, for debugging or dynamic explanation of deductions. To trace a rule or predicate, do

```
*/(DTRACE  (symbol -situations-)
           (symbol -situtations)
            ...)
```

Each *symbol* is a predicate or rule name. Each *situation* is one of <-?, <-!, or ->, and says which situations to trace. (In some Lisp dialects, notably Franz, the macro characters ? and ! must be slashified, so the first two of these must be typed <-\? and <-\!.) <-? means, "Trace at start of goal"; <-!, "Trace on goal success"; and ->, "Trace on assertion."

If the situations and parens are omitted (as in (DTRACE *symbol*), it means to trace on <-? and <-!, unless the symbol is the name of a forward chaining rule, when it means (DTRACE (*symbol* ->)). If you DTRACE the symbol *ALL, then every predicate and rule will be traced in the appropriate way. (E.g., (DTRACE (*ALL <-?)) traces all goals, and so forth.)

In trace mode, a message will go by whenever the appropriate situation occurs. It is also possible to get Duck to "break" (stop and open a read-eval-print loop) on goal generation, goal success, or assertion. To cause a break, replace the *situation* in the call to DTRACE with a pair of the form (<-? <-BREAK), (<-! <-BREAK), or (-> ->BREAK). So to break whenever (FOO ...) is asserted or occurs as a goal, do

```
(DTRACE (FOO (<-? <-BREAK) (-> ->BREAK)))
```

When a <-? or <-! break occurs, the system will put you in "walk mode," looking at the stack of active deductive goals called a "goalstack tableau." This tableau form somewhat resembles the "failed rule tableau" described in Section 7.1:

```
↑ Supergoal: pattern
1 Rule: rulename
Why this rule has succeeded so far:
2 assertion
3 assertion
...
>>> GOAL: goal being traced <<<<
Remaining goals of this rule:
4 pattern
5 pattern
...
```

To understand this tableau, visualize the following scenario: The supergoal occurred, and the rule was tried on it. The rule was of the form

```
(<- super
    (AND s2 s3 g p4 p5))
```

Halfway through it, the trace package broke the goal *g*, either because its predicate was DTRACEd, or because some rule that could conclude it was DTRACEd. Duck stopped and displayed a tableau showing why it succeeded on *s2* and *s3*, and the versions of *p4* and *p5* that remain to be proved. (Actually, as with a failed-rule tableau, there will not be a one-to-one correspondence between the previous subgoals and the things listed under "Why this rule has succeeded so far," because the things listed form the *proof record* justifying the deduction so far.)

As elsewhere in walk mode, if you type the number of a piece of the tableau, attention will shift there. If you type the number of a previous subgoal, it will show you why it succeeded; if you type the number of a subgoal to come, it will enter a "whynot" tableau for it. If you type ↑,

then you will walk up to another goalstack tableau for the supergoal. If you decide to proceed with the deduction, just type Q or OK, and the sytem will go on to prove *g*, then *p4* and *p5*.

When a -> break occurs, the system will open a Lisp read-eval-print loop. To exit, resume as you would from any Lisp break point.

The function (DTR *flag*) turns tracing on or off without losing information about what is being traced. If tracing is turned off and then on again, the tracing state is restored. Tracing is turned on if *flag* is T or ON, else off. (DTR) without any arguments toggles trace mode.

(DUNTRACE (*symbol -sits-*) ...) has the obvious meaning. (DUNTRACE ... *symbol* ...) turns off all the tracing for the symbol.

7.4. The Workfile Manager

Nisp supplies utilities for defining programs and data types in main memory, and saving these things in readable form on disk. A group of things to save is called a *workfile*. This concept is described in more detail in the Nisp manual [McDermott 83a]. (Older documentation uses the misleading term "workspace.") As mentioned before, you may choose not to use workfiles, and instead to edit character files yourself.

A Duck workfile consists of types, objects of those types, and rules about them. The functions for defining these things automatically mark them as "to be saved." There are just two things to remember to do:

1. Get into a workfile before defining anything.
2. Save the workfile before leaving Lisp.

To create a workfile, just execute (WORKFILE '*name*). As explained in the Nisp manual, this doesn't allocate storage, but just sets up a group of things to save; subsequent rules and symbols go into this group.

Any function that defines something automatically notes that it should be saved. The user actually performs the save by typing (WSAVE '*file-name*). This must be done before the Duck job is killed (e.g., just before you log out). To reload it the next time you run Duck, just type (WLOAD '*file-name*). There are other utilities for managing workfiles, but the basic idea is as simple as that.

Workfiles and disk files are two different things. A workfile may be saved in a disk file of a different name; that's why you always have to give the name when you do a WSAVE. (You may want different versions of the workfile saved in different locations on disk.)

Once a workfile has been saved on disk, you can inspect the result, and even edit it, so long as

you leave undisturbed the bookkeeping information at the front and back of the file.

Later, to get back into the workfile, you do not redefine the workspace, but just reload it, by executing WLOAD. This will put you back in the workfile contained in that file; when the next WSAVE occurs, the contents will be rewritten, along with things defined after it was reloaded.

The in-core editor is essentially the old ILISP editor, which was closely related to early InterLisp editors. The Franz Lisp manual [Foderaro 82] contains a copy of the documentation for this editor. You can call this editor to edit a Duck rule by using */(EDITA *rulename*). (The rulename is not evaluated.) When you exit the rule editor (by typing OK), the system will ask you if you want to recheck the rule's syntax. You should type Y unless you have left the rule in a messy state to be corrected after doing something outside the editor. If the syntax check finds unexpected errors, you can abort the check (but keep the changes) by typing \\.

There are also facilities for editing predicate and type definitions, which are described in the next section.

7.5. The Type Editor

A benefit of using the in-core approach is that you don't have to plan out all type declarations before any rule definitions. Instead, Duck will let you tell it interactively about new types and objects. For instance, if you walked up to Duck and typed this rule

```
(<- (HAPPY ?P)
    (AND (MARRIED ?S ?P) (GOOD ?S)))
```

(repeated from Section 3.5), it would complain about undeclared predicates HAPPY, MARRIED, and GOOD. When such a complaint occurs, the system will ask you to correct the problem. If you are using the in-core method, you can take advantage of the interactive "browsing" facilities of Duck to make correction on the spot, and have them take effect permanently (so long as you remember to save the workfile before logging out). With the textfile approach, you can still use the interactive debugger, but you must also make the changes in the file yourself.

Here is what the first complaint from the rule above would look like:

```
The symbol HAPPY is undefined or mistyped.
HAPPY misprint or new?
```

If it had been a misprint, typing the correct symbol would fix the problem. If the symbol really is new, and all you want to do is declare it, simply type its type; if it is a predicate, you can type a list of its arguments. If you type NEW, however, you can tell Duck a few other things about the symbol, by entering walk mode and editing the resulting tableau:

```
HAPPY
```

```
1 Class: UNKNOWN
w>
```

The "w>" prompt signals that you are in "walk mode," described in Section 7.1. The current tableau describes everything known about the symbol HAPPY, which isn't much. The sparseness of the tableau for HAPPY indicates that Duck is expecting you to tell it about the symbol.

If you are using the "textfile" method of managing a Duck program (see Chapter 3.5), then the correct thing to do here is to go to the relevant file, add some DEFPREDs, DEFFUNs, or DUCLAREs, and then read those declarations into Duck before proceeding by typing OK.

If you are using the in-core method, you can tell the editor directly about the undefined symbol, and this information will be saved in a file later. (If you remember to do a WSAVE!) The first thing to tell the editor is the *class* of the symbol. The class should be one of CONSTANT, FUNCTION, or PREDICATE. HAPPY should be a predicate, so we type

```
w> (1 PRED)
Template? w> (HAPPY person ?P)
person misprint or new? new
HAPPY
w> 0
HAPPY
1 Class: PREDICATE
2 Template: (HAPPY person ?P)
3 English: ()
```

The "w>" marks things typed by the user. He has defined HAPPY to be a predicate of one argument, a person. Types are, by convention, lower case.[5] The system has never heard of "person" before, so it makes sure it isn't a misprint.

Having built a type definition, we can now change or inspect it. Typing 0 causes the tableau to be redisplayed. Typing (1 ...) will change the class of the object. Typing (2 ...) will change the template. (3 ...) is used to define an English skeleton corresponding to this predicate. E.g., (3 ?p " is happy") would cause the tableau to become

```
HAPPY
1 Class: PREDICATE
2 Template: (HAPPY person ?P)
3 English: (?P " is happy")
```

(This is the skeleton defined with =: in DEFPRED and DEFFUN. See Section 7.2.) Typing (*n* EDIT) will put you in the Lisp editor, altering piece *n* of the tableau. When you leave the editor, the

[5]As noted in the Introduction, in some implementations they are upper case; what is true in general is that types are the opposite from the normal case for the implementation.

new version will be installed.

Any change made by typing (*n* ...) can be undone by typing UNDO.

It is just as easy to edit the type hierarchy. Let's say we want to tell the system more about person. We type

```
w> ARG ?P
person
1 Supertype: UNKNOWN
No examples of objects of this type.
No examples of functions that return this type.
```

We can give it a supertype by typing

```
w> (1 living-thing)
```

To declare FRED to be a person, we type

```
w> EDIT OBJECT FRED
FRED
1 Class: UNKNOWN
w> (1 CONSTANT)
Type? person
w> 0
FRED
1 Class: CONSTANT
2 Type: person
3 English: ()
```

Now typing

```
w> 2
```

gets us

```
person
1 Supertype: living-thing
  Examples:
    2 FRED
  No examples of functions that return this type
```

You can enter the type editor from a syntax check error, or from Lisp using the functions:

1. (DUKEDIT *obj*): Edits constant, function, or predicate *obj*. If *obj* is undefined, the class will be UNKNOWN.
2. (TYPEDIT *type*): Edits the type. If undefined, it will have supertype UNKNOWN.

References

[Dean 83] Dean, Thomas.
Time Map Maintenance.
Technical Report 289, Yale University Computer Science Department, 1983.

[Doyle 79] Doyle, Jon.
A truth maintenance system.
Artificial Intelligence 12(3):231-272, 1979.

[Foderaro 82] Foderaro, John K. and Sklower, Keith L.
The Franz LISP Manual.
1982.
University of California at Berkeley.

[Forgy 82] Forgy, C.L.
Rete: A fast algorithm for the many pattern/many object pattern match problem.
Artificial Intelligence 19(1):17-37, 1982.

[Hewitt 71] Hewitt, Carl.
PLANNER: A Language for Proving Theorems in Robots.
In *Proc. IJCAI 2.* IJCAI, 1971.

[McDermott 74] McDermott, Drew V. and Sussman, G.J.
The Conniver reference manual.
Technical Report Memo 259a, MIT AI Laboratory, 1974.

[McDermott 83a] McDermott, Drew V.
The NISP Manual.
Technical Report 274, Yale University Computer Science Department, 1983.

[McDermott 83b] McDermott, Drew V.
Contexts and data dependencies: a synthesis.
IEEE Trans. on Pattern Analysis and Machine Intelligence 5(3):237-246, 1983.

[Rees 82] Rees, Jonathan and Adams, Norman.
T: A Dialect of Lisp or, LAMBDA: The Ultimate Software Tool.
In *Proc. 1982 ACM Symp. on LISP and Functional Programming*, pages 114-122. Association for Computing Machinery, 1982.

[Rees 84] Rees, Jonathan, Adams, Norman, and Meehan, James.
The T Manual.
1984.
Yale University.

[Schank 72] Schank, Roger.
Conceptual Dependency: A Theory of Natural Language Understanding.
Cognitive Psychology 3(4):552-631, 1972.

[Sussman 71] Sussman, Gerald J., Winograd, Terry, and Charniak, Eugene.
Micro-Planner Reference Manual.
Technical Report Memo 203A, MIT AI Laboratory, 1971.

[Sussman 72] Sussman, G.J. and McDermott, Drew V.
From PLANNER to Conniver, a genetic approach.
In *Proc. FJCC 41.* Fall Joint Computer Conference, 1972.

[Warren 77] Warren, D.H.D., Pereira, L.M., and Pereira, F.
PROLOG -- The Language and Its Implementation Compared with LISP.
SIGPLAN Notices 12(8), 1977.

Index