# Coursenotes
# Introduction to Prolog
# (Prolog-Einführung)

Tom Cornell
cornell@sfs.nphil.uni-tuebingen.de
(29) 72734

March 9, 1998

# Contents

# Chapter 1

# General Introduction

"Prolog" stands for "Programming in Logic." It is the most widely available general-purpose language for doing *logic programming.* Since the earliest days of digital computers, researchers have been interested in *computational logic* or *automated theorem proving.* The goal there is to get the computer to help search for particularly hard proofs, using the computer's practically infinite patience to supplement the logician's insight. *Logic programming* was born when researchers in computational logic discovered that certain approaches to deduction made sense as a general-purpose programming language. For example, consider the following fragment of Prolog code.

```
true_sentence( Sentence ) :-
     syntactic_analysis( Sentence, Structure ),
     meaning( Structure, LogicalForm ),
     value( LogicalForm, true ).
```

This can be read *logically* as stating that for all *Sentence*, if a syntactic analysis of *Sentence* is *Structure* and furthermore *LogicalForm* is a meaning that can be associated with *Structure*, and furthermore the value of *LogicalForm* is *true*, then *Sentence* is a true sentence. As such it could be an axiom in a logical theory designed to recognize all and only the true sentences of German, for instance.

It also has a *procedural* interpretation. In this interpretation, *true_sentence*, is the name of a procedure with one parameter, named *Sentence*. The procedure *true_sentence* is then defined in terms of other procedures—*syntactic_analysis*, *meaning*, and *value*—which it calls. We would then think of the variables *Structure* and *LogicalForm* as local variables of the procedure.

So logic programs have both a logical (usually called *declarative*) interpretation and also a *procedural* one. Even better, those interpretations coincide. This means that it does not matter whether you view a Prolog program as a set of axioms and Prolog itself as a deductive engine for deriving consequence of those axioms, or you view a Prolog program as a set of procedures, and Prolog itself as an interpreter to handle procedure-calling and parameter-passing. You can look at a program from whichever angle best suits your purpose at the time.

This is very useful because, in general, at the beginning of a long and complex programming project you are likely to be more interested in the logic of the problem you have set out to solve, but later on as the project progresses you will become more interested in solving the problem *efficiently*, so details of the procedure by which the problem is solved will become more important.

The organization of the course reflects this progression from concern for correctness to concern for speed. In the first part of the course, we will focus on learning how to write logically correct recursive definitions of data types and relations over these data types. This is tricky enough, without worrying about efficiency at all! Then, in the second part of the course, once we have gotten accustomed to writing recursive programs, we will begin to focus on issues related to the procedural interpretation of logic programs.

This neat division of topics into *Logic* and *Control* cannot be absolute, of course. We are here to learn about a programming language, not about automated theorem proving, and some issues of control will concern us right from the start. In particular, Prolog is a powerful programming language, and, as with any other programming language, it is possible to write programs that do not halt, but instead run on forever (or until we interrupt them). So even at the very start we will be concerned with some procedural details of Prolog, just so we can be sure that we are writing programs that will indeed halt before the end of the world. Also, some knowledge of Prolog's procedural interpretation is necessary to debug programs that don't do what you think you told them to do. When things go wrong, it is often useful to be able to work out with pencil and paper just what steps the Prolog interpreter is going through when it runs your faulty program, and, obviously, you need to know what those steps will be to do this.

(Of course, some programs simply cannot be guaranteed to halt for all inputs. Assuming that the program, whose top level was given at the beginning of this introduction, was correctly implemented, can you think of an input for it for which it might not halt?)

# Part I

# Introduction to Logic Programming

# Chapter 2

# Why Logic Programming is So Wonderful

One always has to start a programming language course with a little lecture one could call “Why bother?” There are a very great many programming languages available nowadays, and learning each one involves developing new habits, memorizing new things and learning new techniques. All of this is can be quite a lot of WORK! So why bother? (WARNING: The rest of this chapter is loaded with PROPAGANDA!)

A programming language is a formal language, like the language of first order predicate logic, or like a grammar formalism like HPSG or Categorial Grammar. As such it is endowed with a syntax which says what the legal expressions are, and in addition a number of ways of *interpreting* syntactically well-formed expressions. On the one hand, we have the interpretation that means something to us, the programmers. This is the *declarative* interpretation of a program, and reflects the way in which the symbols of the program are supposed to reflect the “real world”. (In a sense, Model Theory is about the declarative interpretation of logical languages; aspects of human grammatical competence furnish the declarative interpretation of grammar formalisms.) Computers, of course, know nothing of the real world (many philosophers would say that computers know nothing, Punkt). They just flip bits from one state to another, basically. More abstractly, computers interpret the programs we write in a way which is blind to the meanings we give our symbols, but which depends very much on the structure of the program and the placement of certain keywords. This meaning-free bit-mashing interpretation of a program furnishes its *procedural* interpretation. (In a sense, Proof Theory is about the procedural interpretation of logical languages; grammatical derivations (phrase structural, transformational, type-logical, etc.) provide a procedural interpretation of grammar formalisms.)

Sometimes you will hear people make claims like “Prolog is a declarative language, while C is a procedural language.” This is loose talk and rather misleading. Both languages have both declarative and procedural interpretations.

The thing which makes Prolog special in this regard is that, while for C the procedural interpretation is fairly straightforward and the declarative interpretation is nightmarishly complicated, for ("pure") Prolog both interpretations are extremely simple, and it is relatively straightforward to prove that they are equivalent in a certain sense.

The reason that logic programming languages like Prolog are considered "declarative languages" is that they are designed primarily with their declarative interpretation in mind. They are, I suppose, "problem oriented" where "procedural" languages are more "machine oriented". Instead of treating the statements of the language as commands, the computer treats them as *descriptions*, in particular as descriptions of what things are true in the world described by the program. Because logical descriptions of a problem have implicit *consequences*, we can still think of this static worldview in computational terms: computation in a logic programming setting is a process of making a program's implicit consequences *explicit*. For example, the logical description of human grammatical competence may have among its consequences that a certain linguistic expression (this sentence, for example) is to be assigned a particular semantic interpretation. That consequence is implicit in the statement of the grammar. Turning that grammar into a parser, for example, is essentially the problem of making that implicit consequence explicit. In this sense *proof*, that is, the explicit assertion of the consequence relation, can be thought of as *computation* and, maybe more remarkably, computation can be thought of as proof.

A lot of discussion has taken place in the past several decades over whether declarative or procedural approaches are "better". They certainly are different, and the differences are especially obvious when you are learning your first declarative language. These differences are very easy to over-emphasize! Still, there are some important things that really are easier to do in logic programming. Here are two of them.

## 2.1 From Descriptions of Problems to Solutions of Problems

Computation, whether we approach it from a declarative or procedural standpoint, is mainly about *problem-solving*. We always specify a problem and its solution by providing certain constraints which describe what information we are given to begin with, and certain other constraints which describe what counts as a solution. So, if done correctly, both procedural and declarative programming projects start from the same place: an essentially declarative description of the problem to be solved. Consider the problem of defining the dominance relation that holds between nodes in a phrase structure tree when the first lies strictly closer to the root than the other along the same path. In what follows, we will give definitions in plain English, but phrased as carefully as may be. This is "specificationese", a special (ill-defined) language for giving (hopefully) unambiguous specifications of procedures without requiring the rigid syntax of, e.g., first order logic, or, for that matter, Prolog. Since we don't know anything about Prolog's syntax yet, we will use "specificationese" as an example of a logic programming language. (The corresponding Prolog code is given in an appendix; you can see for yourselves how closely it reflects the structure of these specifications, even without knowing anything about the rules of Prolog syntax.) A rather literal definition of dominance is the following.

```
X dominates Y in T iff
    there is a branch B in T such that
        X is higher than Y in B.
```

We assume for the moment that we can always extract the branches from a tree, and that a branch is a *list*, that is, a data structure with a head and a tail, where the tail is also a list.

```
X is higher than Y in B iff
    the position of X in B is P1 and
    the position of Y in B is P2 and
    P1 < P2.
```

Now we need to define what is meant by the position of an expression in a list of expressions. The obvious recursive definition is the following.

```
the position of X in L is N iff
    either
        X is the head of L, and
        N is 1,
    or  the tail of L is L1, and
        the position of X in L1 is N1, and
        N is N1+1.
```

These definitions, together with a definition of the "branch" relation, give us enough information to actually compute the dominance relationships between nodes in a tree. The translation of this "specificationese" program into Prolog is provided in the appendix to this chapter; again, you should not need to know anything about Prolog syntax to see that the translation from the specification to Prolog is quite direct, indeed almost word for word. (The code in the appendix contains the definitions of trees and branches as well, and is in fact a complete, running Prolog program, though not by any means remarkably efficient.)

Those of you with some programming background probably already know that often *iterative* procedures are more efficient than recursive, non-iterative ones. Here is the corresponding iterative definition, which represents our first optimization of this logic program.

```
the position of X in L is N iff
    I equals 1 and
    a loop with index I, searching L for X, returns N.
```

This sounds very procedural, but we see immediately that a loop such as this is in fact just another kind of recursively defined relation.

a loop with index I, searching L for X, returns N iff
    either
        X is the head of L, and
        N equals I,
    or  I1 is I+1, and
        the tail of L is L1, and
        a loop with index I1, searching L1 for X, returns N.

The only difference between these two definitions is that in the first we do not know the value of N until we have proved the recursive statement and found the value of N1. In the second, we can do our arithmetic before we prove the recursive conjunct. Thought of as ways of defining the "position" relation, there does not seem to be much to choose between the two definitions, except that the first is decidedly shorter and simpler. Thought of as procedures, however, the second one turns out to be more efficient. Using the first definition, the Prolog interpreter must remember, for every time the procedure is invoked, that it must still do some stuff after control returns from the recursive call. So the interpreter must create a *stack frame*, one frame each time the procedure calls itself. Using the second definition, there is nothing that needs to be remembered, so when the end of the recursion is reached, the Prolog interpreter can go straight back to the definition of "higher", where "position" was originally called. No stack frames need to be created (or discarded).

At this point in the course, you certainly do not need to know what a "stack frame" is. All you need to grasp is that there can be two correct definitions of a relation which are logically equivalent (of course they must be equivalent or they cannot both be correct!) but which are not equally efficient. Furthermore, you should appreciate that knowing how to write efficient logic programs requires knowing a fair amount about how the logic program interpreter or compiler works; nonetheless, one can, in most cases, write *correct* logic programs knowing only how *logic* works. (And of course the syntax of your logic programming language, e.g., Prolog.)

There are lots of other ways this "logic program" could be optimized as well. For example, in order to find the position of Y in branch B we start over at the beginning of the branch, this time searching for Y. But clearly, if X does indeed dominate Y, then at some point while we are searching for the position of Y on the branch we will pass X. So why not search for them both in a single pass through the branch? And in that case, why bother actually calculating exact positions on the branch? Why not just work our way through the list looking for X, and, once we have found it, *then* start looking for Y. Only if we find Y under these conditions can we say that it does indeed come after X. This way we can discover whether X is higher than Y in B with a single pass through B, and without doing any arithmetic. A little more thought (left as an exercise to the especially ambitious reader) and we can eliminate the call to "branch" entirely, working instead directly with the tree.

This kind of optimization is a little different than the optimization where we introduced an iterative recursion in place of a non-iterative one. There we needed to know something about the interpreter or compiler that implements our programming language. Our reasoning here is more general, and more abstract. But it is clearly *algorithmic* reasoning, not *logical* reasoning. Once again, we are faced with many different *logically equivalent* definitions of a relation, and the basis of our choice is efficiency, here interpreted as simply the number of steps we must perform to accomplish a certain task.

This is the kind of reasoning that goes into the discovery of better and better algorithms in general, whether for implementation in "declarative" languages or "procedural" languages. And in fact the more reasoning we do, and the more optimization we do, the closer our supposedly declarative programs will resemble supposedly procedural programs. After all, at the heart of all computation, whether procedural or declarative, we will always find algorithms. So it is important not to over-emphsize the difference between declarative programming and procedural programming. Much more important is the distinction between reasoning about *computers* and reasoning about *problems*. And here pretty much everyone is in agreement: reasoning about problems is first and foremost.

If you are in a situation where you have to squeeze the last ounce of performance from your programs, then there will always come a time at which you will say to yourself "I could make this go faster if I rewrote it in C." In some cases that may be true. However, given the current state of logic programming technology, *it almost never is!* The truly significant gains in speed and memory use always come from reasoning about the problem, and reasoning is always better and clearer if it is done *logically*. The true advantage of logic programming lies here, then: *logic programming presents us with the fewest possible barriers between reasoning and programming.*

## 2.2 From Theories to Programs

As linguists, we are generally less interested in the performance of a program than we are in its correctness. In the case of simple mathematical problems like defining and calculating dominance relations in a tree structure, the underlying theory is already well worked out and not subject to much debate. We can safely assume that we know what a tree is, and what operations are important in dealing with one. In computational linguistics and computational cognitive science more generally we are not in this position. While it may appear that we understand what a "sentence" is, it is not at all clear that we understand what a syntactic analysis of a sentence is. We cannot be certain what the primitive objects of the theory are, nor what operations are possible on them, nor what constraints exist on the computation of these operations. Linguistic theories are extremely complex, they exist in many varieties, which are at best only partly compatible, and they are subject to change almost from day to day. So in cognitive science, when one writes a computer program one must always be prepared to demonstrate rigorously just exactly how that program is related to the corpus of Theory.

This is not always easy to do! It is especially difficult if the program is written in a procedural language. The reason is obvious enough: theories are almost always presented to us declaratively, not procedurally. This is especially true in linguistics. A grammar is meant to tell us what the analysis of a particular expression is, not how to arrive at that analysis. This is just one way of stating the well-known distinction between *competence* and *performance*. But the problems don't end there. Even if we are interested in linguistic performance—suppose, for example, that we are collaborating with psychologists rather than linguists—we still have to start from theories. The theory may be about how certain types of sentences require especially large amounts of a person's working memory to process, which on the face of it at least is more "computational". Nonetheless, even here you will find that psychological theories (for example) are not algorithmic. A great deal remains to be filled in by the computationalist. And once again, every decision the computationalist makes must be *rigorously justified* with respect to the original theory. You must be prepared to *prove* that your program does not introduce answers which are not consequences of the theory, nor does it fail to find answers that are consequences of the theory.

As an example, admittedly oversimplifying the problems involved, let us consider the *recognition problem*. The recognition problem for a particular grammar $G$ is the problem of determining for any given string of symols $S$ whether or not $S$ can be derived from $G$, that is, whether or not $S$ is in the language generated by $G$. Consider the following very simple phrase structure grammar. (Here S stands for sentence, NP for noun phrase, VP for verb phrase, and Det for determiner, i.e., articles like "the", quantifiers like "every", possessives like "John's", etc.)

$$S \longrightarrow NP \quad VP \tag{2.1}$$
$$NP \longrightarrow Det \quad Noun \tag{2.2}$$
$$NP \longrightarrow \text{“it”} \tag{2.3}$$
$$Det \longrightarrow \text{“a”} \tag{2.4}$$
$$Noun \longrightarrow \text{“person”} \tag{2.5}$$
$$Noun \longrightarrow \text{“thing”} \tag{2.6}$$
$$VP \longrightarrow \text{“exists”} \tag{2.7}$$

A grammar generates a string if there is a rewriting derivation beginning with the start-symbol $S$, at each step rewriting some non-terminal symbol in the current string until eventually the string consists only of terminal symbols. This grammar generates sentences like *A thing exists.* (For example, apply rule 2.1, then rules 2.2, 2.4, 2.6, 2.7.) Therefore, *A thing exists* is in the set of expressions that it recognizes; *Exists a thing* is not.

The following Prolog program implements a solution to the recognition problem for this grammar.

```
recognize( String ) :-
      s( String ).
```

*Translation*: "For all *String*, *String* is recognized if it is an *s*."

```
s( String ) :-
      np( String1 ),
      vp( String2 ),
      concatenation( String1, String2, String ).
```

*Translation*: "*String* is an *s* if there is an *np String1* and a *vp String2*, and *String* is the result of concatenating *String1* and *String2*."

```
np( String ) :-
      det( String1 ),
      noun( String2 ),
      concatenation( String1, String2, String ).
np( String ) :- string_eq( String, it ).

det( String ) :- string_eq( String, a ).

noun( String ) :- string_eq( String, thing ).
noun( String ) :- string_eq( String, person ).

vp( String ) :- string_eq( String, exists ).
```

For the moment we will say nothing about how `concatenation` or `string_eq` might be implemented, but will assume that they can be implemented correctly, somehow. (They can, don't worry.)

Interestingly, the description of the recognition problem for this grammar looks very much like the grammar itself! For every rule in the grammar there is exactly one sentence in the program, and the form of the sentence in the program is completely determined by the form of the rule. In a sense, the ability to write declarative programs allows us to blur the distinction between competence and performance. Once we have adequately described (some aspect of) linguistic competence, we almost immediately derive a computational device capable of (some limited) linguistic performance. In fact, in a week or so we will see how to construct a recognizer without even mentioning concatenation explicitly; at that point, our recognizer will essentially be just our grammar with the arrows turned around!

Suppose now that the theory changes. Let's say we discover a new verb, *sneezes*, or we discover relative clauses and want to be able to recognize sentences like *A thing which exists exists*. It is as simple to add clauses to our program as it is to add rules to our grammar. Just as it was true that we could optimize our dominance program, there are many ways we can optimize our recognizer, though they are perhaps not as obvious. Still it must be borne in mind that every optimization we make takes us farther and farther away from the original theory. The primary goal in implementing linguistic theories is always *clarity*.

Remember: *Premature optimization is the root of all evil!* (Incidentally, that was said by a computer scientist, not a linguist!)

## 2.3 Appendix to Chapter 2: Prolog code for the "dominates" relation

```
dominates(X,Y,T):-
        branch(B,T),
        higher(X,Y,B).

higher(X,Y,B):-
        position(X,B,P1),
        position(Y,B,P2),
        P1 < P2.
/*
position(X,B,N):-
        head(X,B),
        N is 1.
position(X,B,N):-
        tail(B,B1),
        position(X,B1,N1),
        N is N1+1.
*/
position(X,L,N):-
        I is 1,
        loop(I,X,L,N).

loop(I,X,L,N):-
        head(X,L),
        N is I.
loop(I,X,L,N):-
        I1 is I+1,
        tail(L,L1),
        loop(I1,X,L1,N).

branch(B,T):-
        leaf(T,R), % T is a leaf with label R.
        head(R,B),
        tail(B,[]).
branch(B,T):-
        root(T,R),
        head(R,B),
        tail(B,BTail),
        daughter(T,TDtr),
```

```
        branch(BTail,TDtr).


% List data structure:
head(H,[H|_]).

tail([_|T],T).


% Tree data structure:
leaf(t(R),R).

root(t(R,_),R).

daughter(t(_,DtrList),D):- member(D,DtrList).

member(X,L):- head(X,L).
member(X,L):- tail(L,L1), member(X,L1).


% Test data, for testing branch and dominates:
data(1,t(a,[A1,A2])):-
        A1 = t(a1,[A11,A12]),
        A2 = t(a2,[A21,A22,A23]),
        A11 = t(a11),
        A12 = t(a12),
        A21 = t(a21),
        A22 = t(a22,[t(a221)]),
        A23 = t(a23).
```

# Chapter 3

# Elements of Prolog: Expressions, Unification, and Resolution

Prolog is a particularly simple programming language. It consists of a single data structure (which we will refer to here as *expression trees*) and a single operation on those data structures (called *unification*). The deductive calculus that it implements is equally simple. There is one inference rule (*resolution*) and no axioms. Despite all this it is an extremely powerful programming language. It can be used to implement Turing Machines, or theorem provers for full First-Order Logic, or Prolog itself, for example.

## 3.1 Proplog: The Resolution Rule for Propositional Logic Programs

We will introduce the Resolution Rule first for a strictly *propositional* version of Prolog, called Proplog. Essentially, Proplog is a subset of full propositional (or "sentential") logic in the same way that Prolog is a subset of full first order predicate logic. We cannot do very much with Proplog, but since we do not have to deal with variables, we can introduce the Resolution Rule independently of the Unification Operation. Then, at the end of the chapter, we put them together and arrive at (pure) Prolog.

### 3.1.1 Clausal Form

Proplog (and Prolog) programs are written in a special format called *clausal form*, which we now proceed to define, for the case of propositional logic. Formally, we take a *clause* to be an expression of the form:

$$A_1, \ldots, A_m \leftarrow B_1, \ldots, B_n,$$

where each of the $As$ and $Bs$ stand for propositions. A clause is meant to express a logical implication of the following sort: if all of the $Bs$ are true, then one of the $As$ must be true as well. This rule, which provides a *semantics* for clauses, is also true of the (classical propositional logic) expression

$$B_1 \wedge \ldots \wedge B_n \rightarrow A_1 \vee \ldots \vee A_m$$

The two expressions are *equivalent*, they *mean* the same thing. Here is yet another way of putting the same rule of interpretation: either at least one of the $Bs$ is false or at least one of the $As$ is true.

The following two laws of propositional logic

$$(p \Rightarrow q) \equiv (\neg p \vee q)$$
$$\neg(p \wedge q) \equiv (\neg p \vee \neg q)$$

tell us that clauses are also equivalent to expressions of the form

$$A_1 \vee \ldots \vee A_m \vee \neg B_1 \vee \ldots \vee \neg B_n.$$

This version of the logic of clauses gives rise to yet another way of representing them:

$$\{A_1, \ldots, A_m, \neg B_1, \ldots, \neg B_n\}.$$

The elements of such sets are called *literals*, and the set is interpreted *disjunctively*. In the literature on logic programming and theorem proving you will see clauses represented in all of these (equivalent!) ways.

Given this interpretation (these interpretations?), the clause

$$A_1, \ldots, A_m \leftarrow$$

must mean that the disjunction of the $As$ ($A_1 \vee \ldots \vee A_m$) is true, since it is impossible for one of the $Bs$ to be false (there are none!). Also, the clause

$$\leftarrow B_1, \ldots, B_n$$

means that the conjunction of the $Bs$ ($B_1 \wedge \ldots \wedge B_n$) is false, since it is impossible for one of the $As$ to be true.

The *empty clause* " $\leftarrow$ " (also written $\Box$) therefore ought to mean something like "true implies false," which is of course false. Therefore the empty clause is always interpreted as false.

The importance of clausal form is that we can do away with all the boolean connectives and, in the case of predicate logic, we can do without representing quantifiers as well. This way we don't need special rules to handle them all. Since the axioms of a standard logic are for the most part just there to tell us how to manipulate the connectives and quantifiers, needing no connectives means that we need no axioms, either. Nonetheless we don't lose anything (except succinctness!) by writing in clausal form. *Every sentence of propositional logic* *has a translation into a set of clauses which is logically equivalent to it.* (And the same remains true in the full first order case as well.) As a typical example that we will often encounter, the implication $A_1 \vee A_2 \Rightarrow B$ can be represented by the set of clauses $\{(B \leftarrow A_1), (B \leftarrow A_2)\}$.

A *Horn clause* is a clause with at most one expression on the left of the arrow. The expression on the left of the arrow (if there is one) is called the *head* of the clause. The expression(s) to the right of the arrow (if there are any) make up the *body* of the clause. The four possible types of Horn clause are conventionally named as follows.

**Facts** Clauses of the form $A \leftarrow$ . (Facts have a head but no body.)

**Rules** Clauses of the form $A \leftarrow B_1, \ldots, B_n$. (Rules have both a head and a body.)

**Goals** Clauses of the form $\leftarrow B_1, \ldots, B_n$. (Goals have a body, but no head.)

**Empty Clause** The clause $\Box$, with no head and no body.

A *definite clause* is a clause with *exactly* one expression on the left of the arrow, i.e., either a fact or a rule. A *definite clause program* or *definite clause theory* is a set of definite clauses. Prolog programs (and so, *a fortiori*, Proplog programs) are definite clause programs.

Clauses are represented in Prolog (usually) using “Edinburgh” syntax. Observe

**Table 1** Clausal form, Edinburgh style

**Facts** `A.`

**Rules** `A :- B1, ... ,Bn.`

**Goals** `?- B1, ... ,Bn.`

that all clauses end in a period. Note that the ‘`?-`’ symbol is the Prolog intepreter’s prompt, so you don’t ever type that in yourself. You should have no need to ever type in the empty clause. Note that (at least in SICStus Prolog) the expression “`?-.`” (got by typing a period at the Prolog interpreter’s prompt) will cause an error.

### 3.1.2 The Resolution Rule and the Interpretation of Logic Programs: The Propositional Case

The resolution rule was originally developed for use in full First-Order Logic theorem provers. Prolog uses a very specialized form of resolution (called “SLD Resolution”) specifically adapted to definite clause programs, so I will not present a general introduction to resolution systems here.

The fundamental operation of a resolution system takes a pair of clauses as input and produces a new clause, called their *resolvent* as output.

**Table 2** Proplog's version of the Resolution Rule

1. Given: A goal clause $\leftarrow G_1, \ldots, G_k$
2. Search the program for a program clause $A \leftarrow B_1, \ldots, B_n$ such that $G_1$ matches $A$.

   If no such clause exists, then fail.
3. Replace $G_1$ with $B_1, \ldots, B_n$.
4. If the result is the empty clause, then halt, reporting success.

In a Prolog computation we are given a program which, as we already know, is a set of definite clauses, and we are also given a single goal clause which expresses the particular problem instance we wish to solve. In the kind of resolution implemented in Prolog, one of the two clauses we resolve must always be the goal clause, and the resolvent always becomes the new goal clause. This is called *linear resolution*.

The resolution step then proceeds as follows. First, we take the current goal clause and we select one of its constituent atoms. In principle, we can select any atom we want, but Prolog will always select the leftmost member of the goal clause. This is called resolution with a *selection function*. Prolog's selection function applies to a sequence of expressions and returns the leftmost expression in that sequence. This sort of resolution system is called *linear resolution with selection function*, or SL-resolution. Therefore you will often see Prolog's particular type of resolution called SLD-resolution, with the 'D' standing for 'Definite Clauses'. The variant of resolution we will use for Proplog is just SLD resolution further restricted to clauses where all the atoms are propositions.

Our second step, after selecting an expression from the current goal clause, is to search the program for a clause whose head matches the selected sub-goal. In Prolog (and so also in Proplog), this search follows the order in which clauses appear in the program text. In the propositional case, the resolvent of the goal with the clause we select is the result of (a) removing the selected member of the goal clause and (b) replacing it with the body of the selected program clause.

Note that if we resolve the goal with a fact, then we replace it with nothing (since facts are clauses without bodies). That is, we simply remove it from the goal. Intuitively, when the goal finally becomes empty, then we know we have reduced our original goal to a collection of facts, and so we have proved the original goal. This intuitive presentation of the declarative interpretation of logic programs under the SLD Resolution Rule is in fact not quite accurate, but it will do for the purpose of gaining reliable intuitions about how Prolog programs should be interpreted. For the reader interested in more details of the logic underlying (propositional) resolution, we offer the following subsection, which can be skipped if desired.

### A Brief Introduction to Propositional Resolution

The careful reader may have noticed that in our presentation of the types of Horn clause we referred to clauses like "$\leftarrow A_1, \ldots, A_m$" as *goal clauses.* However, when we were talking about the interpretation of clauses, we observed that such a clause is equivalent to the negation $\neg(A_1 \wedge \cdots \wedge A_m)$. If our intuitive goal is to prove $A_1 \wedge \cdots \wedge A_m$, why do we refer to the *negation* of that expression as the "goal clause"?

The answer is that resolution-based theorem-provers are *refutation* provers. Recall the definition of logical consequence. Given a program (a set of clauses) $P$, a clause $G$ is a consequence of $P$ just in case every interpretation which satisfies $P$ also satisfies $G$. Notice that if this is true, that every satisfying interpretation of $P$ also satisfies $G$, then there can be no satisfying interpretation of $P$ which satisfies $\neg G$. If there were, then there would be a way of interpreting $P$ which satisfied $G \wedge \neg G$, which is a contradiction. Note further that if no satisfying interpretation of $P$ also satisfies $\neg G$, then there exists no satisfying interpretation of $P \cup \{\neg G\}$. So one way to prove that $G$ is a consequence of $P$ is to prove that $P \cup \{\neg G\}$ is unsatisfiable. This is called a *proof by refutation.* To prove something, you refute its contrary.

Sometimes it is easier to prove that sets of logical sentences are satisfiable, sometimes it is easier to prove that they are unsatisfiable. In the latter case, refutation methods are obviously useful, and that is the situation we are in with computational logic. As a result most automated theorem proving methods are based on refutation.

In the particular case where all our sentences are in clausal form, we can implement a refutation theorem prover in the following way. We know that the empty clause, $\Box$, is unsatisfiable. Suppose then that we had an inference rule which was *sound*, that is, which was such that if its inputs were satisfiable, its output would be too. Suppose further that we apply this inference rule to a program $P$ and a negated goal $\neg G$, and eventually derive $\Box$. It is a law of logic that, if $A$ implies $B$, and furthermore $B$ is false, i.e., $\neg B$ is true, then $A$ must be false. By the soundness of our inference rule we have that, if the inputs are satisfiable ($A$), then the output is satisfiable ($B$), and by the fact that the output is $\Box$ we have also $\neg B$. Therefore the claim $A$, that the inputs were satisfiable, must be false: the inputs were in fact unsatisfiable. By an induction on the length of the proof, we work our way back to the realization that our original assumptions, taken together, must be unsatisfiable. That is, $P \cup \{\neg G\}$ is unsatisfiable, that is, there is no interpretation of $P$ which makes $\neg G$ true, that is, every interpretation of $P$ makes $G$ true, that is, $G$ is a consequence of $P$.

The final piece of the puzzle, then, is to show that the Resolution Rule is indeed sound. For general propositional resolution, we can represent the rule as:

$$\frac{\{A_1, \ldots, A_i, X, A_{i+1}, \ldots, A_m\} \qquad \{B_1, \ldots, B_j, \neg X, B_{j+1}, \ldots, B_n\}}{\{A_1, \ldots, A_m, B_1, \ldots B_n\}}$$

To prove soundness we assume that the inputs are satisfiable and show that, under that assumption, the output must be as well. Assume that the boolean valuation $v$ is one of the interpretations which make the inputs true. Then $v$ either makes $X$ true or it makes it false. Assume for the moment that it makes $X$ true. By assumption, $v$ also makes the second input clause (call it $B$) true. But if it makes $X$ true, it must make $\neg X$ false, so in order to make $B$ true, it must make one of the other literals in $B$ true (call it $B_k$). Since $B_k$ is not $\neg X$, it must be a member of the output clause. So we know that, under $v$, the output clause must be satisfiable, because it contains a literal that $v$ makes true. The other half of the proof, under the assumption that $v$ makes $X$ false, is entirely symmetrical, and we leave it as an exercise to the reader.

We can also show that the Resolution Rule is *complete*, which means that if the output is satisfiable, then the inputs must have been as well. That means that if we *can't* derive $\square$ from $P \cup \{\neg G\}$ then $G$ must not be a consequence of $P$. That proof is more complex, however, and less important for building confidence in the declarative interpretation of Prolog programs. So we leave it out, and refer the reader to Melvin Fitting's excellent and very useful book *First-Order Logic and Automated Theorem Proving* (Springer, 1990).

### 3.1.3 An Example of Proplog in Action

We will illustrate the resolution rule with the simple theory in propositional logic given in table 3. We use "Edinburgh syntax" here—refer back to table 1 if necessary.

This theory looks a lot like a phrase structure grammar, but note that it means something a little funny, since it just treats what we think of as grammatical categories as if they were simply facts. So all we can really do is establish whether or not `s` is a true fact. Roughly, the lexicon is a collection of facts and the program can tell us whether or not these facts can be combined in such a way as to make an `s` or a `vp`, for example. So ultimately the only interesting questions we can ask are questions like "Are there any S's," or "Are there any NP's." We cannot, for example, find out what those S's and NP's actually are. Still, it makes a good starting place: the subject matter is familiar, and the form is simple.

Suppose then that our initial goal is the query "`?- s.`" We apply our selection function to select an atom from the goal. The leftmost atom in the goal is, of course, `s`. Then we search the program for clauses whose heads are `s`. There are two of them:

```
s :- np, vp.
s :- np, aux, vp.
```

According to our Prolog-style implementation of the Resolution Rule, we must try the first clause first. So we store some indication that we have not yet tried clause 2 (this is usually called a "choice point"), and we remove the atom `s` from the goal and replace it with the body of the first clause, giving the new goal "`?- np, vp.`"

**Table 3** A Proplog Theory.

```
s :- np, vp.
s :- np, aux, vp.

np :- det, n.   np :- name.

det :- the.     det :- every.

n :- prisoner.

% No clauses for name.

aux :- has.

vp :- v.

v :- escaped.

% Lexicon:
the.   every.
prisoner.
has.
escaped.
```

Now the cycle repeats itself. We apply the selection function to the goal, yielding `np`. We search the program for clauses headed by `np`, finding now

```
np :- det, n.
np :- pron.
np :- name.
```

We start with the first clause, as always. So we push a new choice point onto the stack, recording that clauses 2 and 3 remain untried, and we replace the selected atom in the goal with the body of the selected clause, yielding the new goal "`?- det, n, vp.`" We select `det` next, pick the first clause in its definition, and replace it in the goal with the body of that clause, yielding the new goal "`?- the, n, vp.`" We select `the`, pick the first and only clause in its definition and replace it in the goal with the body of its clause. The body of its clause is empty, so the result of this "replacement" is "`?- n, vp.`" It should be clear from this point how the rest of the proof proceeds. We summarise it as follows.

```
?- s.
?- np, vp.
?- det, n, vp.
?- the, n, vp.
```

```
?- n, vp.
?- entity, vp.
?- vp.
?- v.
?- exists.
?- .
```

In the last step we derive the empty clause, so the original goal succeeds.

Just as the program looks a lot like a context-free phrase-structure grammar, so this proof looks a lot like a derivation from that grammar. The only real difference is that we throw away the terminal categories `the`, `entity`, and `exists`, rather than glueing them together into a string. We will see later how to capitalize on this correspondence between definite clause proofs and phrase structure derivations to develop an extremely powerful grammar formalism, known as Definite Clause Grammars (DCGs). For the moment, however, we are just trying to understand how the Resolution Rule, as implemented in Prolog, works.

### Backtracking

Our previous proof went very smoothly. What happens when things go wrong? Suppose, for example, that we changed the definition of `np` to the following.

```
np :- name.
np :- det, n.
np :- pron.
```

All we have done here is to reorder the clauses so that the one defining noun phrases as names (i.e., proper nouns) comes first. Observe that there are no clauses in the definition of `name`. So the derivation does not proceed very far. We give the first few steps of the derivation below, together with the choice points we create; we represent choice points here as pairs of the form `(<selected-atom>,<next-untried-clause>)`, and write them so that the left-most choice-point is always the most recent.

```
Goal:                Choice Point Stack:
?- s.
?- np, vp.           (s,2)
?- name, n, vp.      (np,2),(s,2)
```

Now, the search for clauses defining `name` fails, so we have to backtrack. Backtracking just means that we pop the most recent choice-point off of the top of the stack, and go back to the point at which we selected that atom. In this example, the last choice we made was which clause to resolve `np` against, and the next untried clause is clause number two. So we choose that clause, and replace `np` with `det, n`. Now the proof looks as follows:

```
Goal:                Choice Point Stack:
?- s.
```

```
?- np, vp.          (s,2)
?- name, n, vp.     (np,2),(s,2)
<fail; backtrack!>
?- np, vp.          (s,2)
?- det, n, vp.      (np,3),(s,2)
```

Now the proof continues as in the first example.

What if the definition of `np` had consisted only of the single clause "`np:- name.`"? There are no more clauses to try, so there is no reason to push a choice-point.

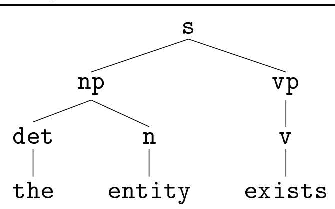
```
Goal:               Choice Point Stack:
?- s.
?- np, vp.          (s,2)
?- name, n, vp.     (s,2)
```

So, on backtracking, we would go all the way back to the `s`-rule, to try its second clause. Of course, in this grammar every sentence requires an NP, so there is ultimately no way past this faulty `np`-rule! In this case we will ultimately run out of choices and, at some point, the proof will fail and the choice point stack will be empty. At that point, Prolog finally admits failure, and prints out the rather uninformative answer "`no`".

**Proof Trees and Search Trees**

A tool which is often useful in the design and analysis of logic programs is the `proof tree`. Intuitively, a proof tree just records the successful proof of a goal in tree-like form. A little more formally, if we think of arbitrary Proplog clauses as grammar rules, then proof trees are just the trees generated by that "grammar". And when a Proplog program (a "propgram"?) really does represent a grammar then, well, the resemblence between proofs and derivations becomes that much more obvious. The proof tree in figure 1 can be read as follows. In order to

**Figure 1** Proof tree for the goal `?-s.`



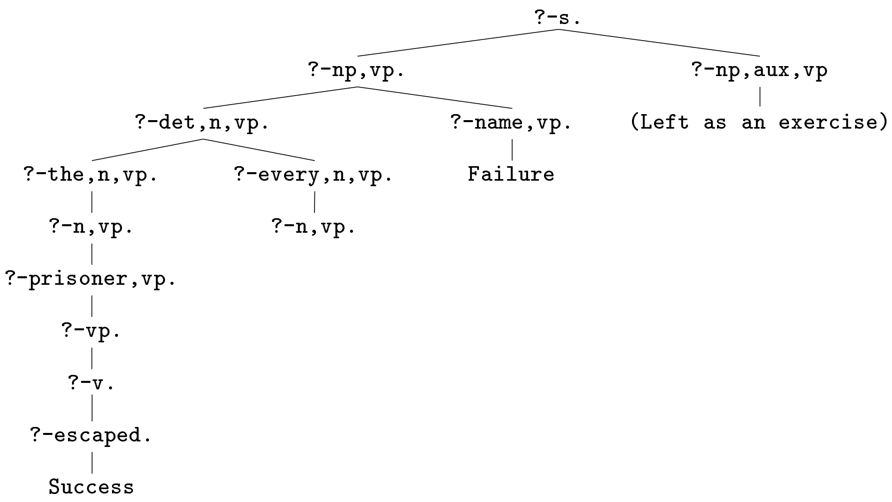

prove `s` we had to prove `np` and we had to prove `vp`. To prove `np` we had to prove `det` and `n`. To prove `det`, we had to prove `the`, and the proof of `the` was immediate from the program.

Proof trees can be very helpful when you are designing a new program. They help one to reason about a problem in terms of its subproblems: in order to prove my overall goal, what things do I need to accomplish first? The tree shape reminds one that, once one knows that `np` must be established, the problem of how to do so can be solved in a similar fashion, and in isolation from other problems, like how to prove `vp`.

A proof tree is read conjunctively: a parent node is established by establishing its first daughter, *and* its second daughter, and so forth. This focuses attention on what logical resources are *required* in order to prove a goal. However, it does not tell us what gets done with other resources in our program. That is, the proof tree focuses on a successful proof, and describes how that proof went, but it abstracts away from all the searching the Prolog engine might have had to do to find that proof. To give us a better idea of how much work the Prolog engine has to do we use *search trees*.

Search trees are read disjunctively. Nodes are labeled with goals, and each daughter node represents one way of attempting to prove the goal labeling its parent node. The shape of a search tree depends very heavily not only on the program but also on the interpreter that runs it. In particular, we can only draw a search tree once we know the selection function our interpreter uses. Since Prolog uses a leftmost-first selection function, we develop a search tree as follows. Given a node labeled with the goal `?-G1, ... ,Gn.` we search the program for clauses whose heads match `G1` and, for each such clause, we give our node a daughter labeled with the resolvent of the parent goal with that clause. That is, we replace `G1` with the body of the clause we have found.

Figure 2 Partial search tree for ?-s.




The tree in figure 2 shows most of the search tree for the goal `?-s.`, given the program in table 3. (For the most part the missing pieces are just copies of parts of the search tree given in the figure.) We have ordered daughter nodes from left to right so as to reflect the order of clauses in the program. Accordingly, Prolog can be seen as executing a depth-first, left-to-right search of the search tree, looking for leaf nodes labeled "Success" (i.e., looking for instances of the empty clause). If we reorder the clauses in the program so as to force backtracking, and reorder the subtrees of the search tree accordingly, we move the branch that ends in "Failure" to the left of the success branch. Prolog's backtracking can now be seen as just climbing back up the search tree to the most recent branching node and trying another daughter of that node until it runs out of daughters to try; then it must climb back up a little higher. In this way, Prolog will eventually search the entire tree if it has to, and if it finds no success-nodes, then it will report failure and halt. That is, it will halt as long as the search tree is finite. If the search tree is infinite, well then, things are a little more complicated, as we see in the next section.

### Left Recursion and Infinite Proof Searches

There are two famous problems that can arise due to compromises that were made in the design of Prolog. One of them is as follows. I asserted of the program with the faulty `np`-rule that at some point Prolog will run out of choices. However, there are programs for which Prolog will never, ever run out of choices. Often this is not a problem: it just means there are an infinite number of distinct solutions, and if you want them all you must have infinite patience! But sometimes this means that an otherwise logically correct program, given a true goal with a finite number of distinct answers will nonetheless fail to halt. Suppose we add the following clause to the `s`-rule.

```
s :- s, and, s.
```

We add "`and.`" to the lexicon as well. Now, given the faulty `np`-rule, we will ultimately have to use this new clause and expand the goal "`?- s.`" to "`?- s, and, s.`" Ultimately, the other two clauses for `s` will fail again, for lack of any proofs of `np`, and we will work our way down to clause 3 of `s` again, expanding the goal to "`?- s, and, s, and, s.` Every time we expand "`s`" to "`s, and, s`" we get a whole new fresh set of choices for how to expand the leftmost literal. Clearly this program will never halt: it will even fail to fail. In terms of search trees, the search tree for this goal now contains an infinite branch.

The problem is that we have a rule in our program which is *left recursive*: it calls itself first, before it does anything else. Left recursive programs will (almost) always eventually fall into an infinite proof-search. This is a severe problem, but it can be remedied. It is always possible to convert a left recursive program into an equivalent, non-left-recursive program, but the result is not always so readable. In our case we could redefine the grammar so that it contains a new non-terminal symbol, called `foo`, and redefine `s` as follows.

```
s :- foo.
s :- foo, and, s.

foo :- np, vp.
foo :- np, aux, vp.
```

What we have done is to package all of the unproblematic `s`-rules from the old grammar under the new symbol `foo`. Now the `s`-rule will only call itself after it has found a `foo`. In our example with the faulty `np`-rule, we are guaranteed to run out of ways to prove `foo`, and, failing to find a proof for `foo`, we will not look for further ways to prove `s`.

The solution works, but at some cost: in our case, we have introduced a symbol which has no part to play in our theory of grammar. It is meaningless; or rather, it only has a procedural meaning. As a result, our proofs are no longer identical to derivations in the corresponding grammar. Even worse is the following. Normally a sentence of the form "S and S and S and S," can have a number of interpretations, corresponding to the bracketings [S and [S and [S and S]]], [[S and S] and [S and S]], and so on. Our modified program will find only one way to prove that such a sentence is an `s`, corresponding to the first of these bracketings. These problems can be gotten around, but the example still serves to show that, even when efficiency is not a concern, we still have to know something about Prolog's procedural interpretation just to get our programs to halt. Efficiency may not be a concern at the moment, but *effectiveness* always is.

## 3.2 The Unification Operation

### 3.2.1 Expressions—A universal data structure

Prolog is designed for *symbolic* computations; in this it is very different from a language like Fortran which is designed specifically for *numeric* computations. We begin by assuming that we have available to us a large collection of unanalyzable *primitive expressions*, or just "primitives", for short, corresponding to a sort of *alphabet* or *lexicon*. Out of these we will construct larger and larger expressions which we can use in various ways: as names for things, relations on the things so named, sentences about these relations, and finally sets of such sentences. These logical objects will then be shown to correspond to familiar programming constructs along the following lines.

| | | |
|---|---|---|
| terms (i.e., names) | $\leftrightarrow$ | data structures |
| relations | $\leftrightarrow$ | procedure and function calls |
| sentences | $\leftrightarrow$ | procedure definitions |
| sets of sentences | $\leftrightarrow$ | programs |

Our primitives come in three basic flavors, or *types*: variables, numeric constants, and symbolic constants. The *type* of a given primitive is determined by the following *naming conventions*.

**Variables** Variables are written as strings composed of

- letters,
- numbers, and

- the underscore character (_)

beginning with either

- an uppercase letter, or
- the underscore character.

For example, *X*, *X1*, *NP_Tree*, *_the_weather* are all legal variables.

**Constants**

**Numerals** Okay, you all know what those look like ...

**Symbols** Symbol names are either

**'standard' symbols** which are composed the same way as variable names, except that they begin *with a lowercase letter,* or

**operator symbols** which are for the most part predefined (for example, <, =, +, −>, :−, \+, etc.)

**quoted symbols** that is, arbitrary strings enclosed in single quotes.

For example, *a*, *a31*, *death_and_taxes* are all legal names for symbolic constants.

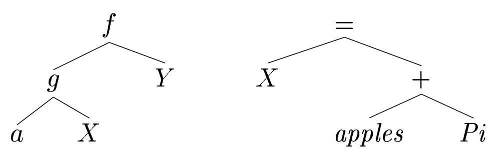
From these primitive symbols we construct *expression trees* according to the following recursive definition.

- A single node labeled with either a constant or a variable is an expression tree.

- A tree whose root is labeled with a symbol and whose subtrees are all expression trees is an expresison tree.

- Nothing else is an expression tree.

**Figure 3** Some expression trees, written "horizontally" as $f(g(a, X), Y)$ and $X = apples + Pi$.




Concretely, we write these expression trees in one of two ways. If the root is labeled with a standard symbol or a quoted symbol, then we write

$$\langle \textit{root-label} \rangle(\langle \textit{sub-tree-1} \rangle, \ldots, \langle \textit{sub-tree-n} \rangle).$$

For example, $f(g(a, X), Y)$ is a way of writing the expression tree whose root is $f$ and whose subtrees are the expression trees which we write as $g(a, X)$ and $Y$ (see figure 3). Operators, on the other hand, are written infix when they take two arguments and are written either prefix or postfix, depending on the operator, when they take only one argument. For example, $X = \mathit{apples} + \mathit{Pi}$ is a way of writing the expression tree whose root is labeled '=' and whose subtrees are the expression trees $X$ and $\mathit{apples} + \mathit{Pi}$. As usual, we can (and sometimes we must!) use parentheses to make it clear what the structure of the expression tree is. For example, the following are all correctly written expression trees.

$$\mathit{ancestor}(X, \mathit{eve})$$
$$\mathit{head}(\mathit{NounPhrase}, \mathit{Noun})$$
$$\mathit{Current\_Product} * 2$$
$$3 * (4 + 2)$$

The reason for focusing on the more abstract notion of *expression tree* rather than the concrete expressions illustrated above is that we can abstract away from the different ways of writing expression trees, and focus on their structure, no matter how it may happen to be written down.

### 3.2.2 Unifying Expression Trees

Now that we have some data structures we can work with, let us investigate what we can do with them. We ask first perhaps the most elementary question we can imagine: When are two data structures equal? We will ask this question under particularly strict conditions: When is it *valid* to claim that two expression trees are equal?

Validity means that a claim holds under all the possible (logical) ways of interpreting the symbols it contains. Consider for example the following claim.

$$2 + 2 = 4$$

Now, numerals like 2 and 4 and operations like + have certain very popular, very traditional interpretations, and under these interpretations, both the expression $2 + 2$ and the expression 4 denote the same number. However, to paraphrase Humpty Dumpty in *Through the Looking Glass*, which is to be the master, you or the symbol?

> "When I use a word," Humpty Dumpty said in a rather a scornful tone, "it means just what I choose it to mean–neither more nor less."
> "The question is," said Alice, "whether you can make words mean different things."
> "The question is," said Humpty Dumpty, "which is to be master—that's all."

For example, there is no mathematical or logical difficulty in interpreting '2' and '4' as themselves, i.e., as one-character strings, and interpreting '+' as the concatenation operator, in which case '$2 + 2$' is another name for the string '22', which is certainly not the same string as '4'.

So let us ask ourselves under what conditions a given equation is valid? The answer is quite simple: when both sides are *syntactically identical*. Clearly, for example,

$$2 + 2 = 2 + 2$$

will be true no matter how we interpret '2' or '+'. (Equality, in first-order logic, has a fixed interpretation: we don't get to play around with that!) This is the kind of equality which Prolog uses: *syntactic equality*. The advantage of this is that from it we can get back any more permissive interpretation we might want. For example, it is a relatively simple matter to define a predicate `arithmetic_value` or a predicate `string_value` in Prolog such that `arithmetic_value(2+2,4)` is true and `string_value(2+2,22)` is true. Note that it would be rather more complicated to implement the string interpretation if our basic interpretation of our symbols was the standard arithmetical one.

Now let us grant ourselves some variables, to represent unknown information. A fundamental question in logic programming is a question like the following (written Edinburgh-style).

```
?- 2+X = Y+3.
```

This particular goal is interpreted as meaning "for what values of `X` and `Y` is this goal valid?" The answer is that it is a valid claim about expressions only as long as `X=3` and `Y=2`. Only then are the expressions on either side of the equality syntactically equal.

Two expressions that can be made syntactically equal are said to *unify*, and so goals like that above are often termed *unification problems*. The set of variable bindings {`X=2,Y=3`} (more often written {`X/2,Y/3`}) is called a *unifier* for the given pair of expressions. We can very fruitfully think of this unification problem as expressing a certain kind of constraint on the values of the unknowns `X` and `Y`. Note that the solution to the original equation is itself a set of equations. It is a set of equations of a very special sort, namely one in which the left hand side is always a variable and furthermore it is a variable which appears nowhere else in the set of equations. (Most especially, it does not appear on the right hand side of its own equation.) Such a set of equations is said to be in *solved form*: it is a constraint satisfaction problem which "wears its solution on its sleeve," so to speak. The unification operation is essentially a process of converting an arbitrary set of equations into a set in solved form (if that is possible).

Consider now a unification problem with a less restricted set of solutions.

```
?- tree(s,NP,VP)=tree(Parent,john,Predicate).
```

Clearly `Parent` must equal `s` and `NP` must equal `john` to solve this equation, but what are we to do with `VP` and `Predicate`? Clearly we can assign any expression to `VP` as long as we assign the same expression to `Predicate`. There are an infinite number of distinct expression trees, so this problem has an infinite number of solutions. However, no matter what those solutions are, they satisfy the simple equation `VP=Predicate`. The solution {`Parent=s, NP=john, VP=Predicate`} is indeed a solution—it is in solved form—and furthermore it is a *most general* solution, or "most general unifier" (*mgu*, for short). Typically, mgu's are not unique (in this example we could have said {`Predicate=VP`} instead, for example). However, all mgu's for a given problem are isomorphic; they differ only in the names of the variables. So we usually speak of "the" mgu for a problem and say that, if it exists, it is unique (up to isomorphism).

These same ideas can be applied to sets of equations, not just single equations. We usually speak of "systems" of equations in this case. Like single equations, systems of equations also have a (unique) most general unifier. Consider, for example, how we might intuitively go about solving the following system of equations.

```
?- T=tree(s,NP,VP), T=tree(Root1,john,Pred),
         T=tree(Root2,Subj,sneezes).
```

Here we have three different constraints on the value of `T`, for which we must find a single solution. One thing we can do, since we know that among other things `T=tree(s,NP,VP)`, we can substitute the expression on the right for `T` in the original system. We transform it to the following, equivalent system.

```
?- T=tree(s,NP,VP), tree(s,NP,VP)=tree(Root1,john,Pred),
        tree(s,NP,VP)=tree(Root2,Subj,sneezes).
```

Clearly, this system has the same solutions as the original: all we have done is to substitute equals for equals. In other words, we have *eliminated the variable* `T` from the system of equations. (Note that we have not literally "eliminated" it: it still appears once on the left hand side of the first equation. Nonetheless, it will play no further role in our calculations.)

Now, if we are to solve the second member of this system,

```
tree(s,NP,VP)=tree(Root1,john,Pred),
```

then it will have to be the case that the following three equations are satisfied.

```
Root1=s, NP=john, VP=Pred.
```

In general, two structures can only be equal, under the theory of syntactic equality, if their functors are the same (`tree`, in this case), they have the same number of arguments (3, in this case), and each of the arguments is equal. So we *decompose* a more complex unification problem—the problem of unifying two structures—into several smaller ones—the problems of unifying the argument-pairs.

We are now looking at the following system.

```
?- T=tree(s,NP,VP), Root1=s, NP=john, VP=Pred,
    tree(s,NP,VP)=tree(Root2,Subj,sneezes).
```

One thing we can do now is to "eliminate" the variables NP and VP.

```
?- T=tree(s,john,Pred), Root1=s, NP=john, VP=Pred,
      tree(s,john,Pred)=tree(Root2,Subj,sneezes).
```

Next, we can decompose the last equation.

```
?- T=tree(s,john,Pred), Root1=s, NP=john, VP=Pred, Root2=s,
                    Subj=john, Pred=sneezes.
```

Finally, we can do variable elimination on `Pred`.

```
?- T=tree(s,john,sneezes), Root1=s, NP=john, VP=sneezes, Root2=s,
                    Subj=john, Pred=sneezes.
```

This system is in solved form; it is the most general unifier for the original system. We find, after all that work, that there is just one value possible for `T` in the most general unifier.

In general, we can characterize the unification operation as the transformation of a system of equations into a system in solved form, using the following three rules.

**Table 4** Rules for solving unification problems.

**Trivial:**

$$\{u = u\} \cup S \Longrightarrow S.$$

**Term Decomposition:**

$$\{f(u_1, \ldots, u_n) = f(v_1, \ldots, v_n)\} \cup S \Longrightarrow \{u_1 = v_1, \ldots, u_n = v_n\} \cup S.$$

**Variable Elimination:**

$$\{x = v\} \cup S \Longrightarrow \{x = v\} \cup \sigma(S),$$

where $x = v$ is not in solved form, $x$ does not occur in $v$, and $\sigma$ is a function replacing every occurrence of $x$ in $S$ with $v$ (the substitution of $v$ for $x$ in $S$).

We will find, once we put unification and resolution together to yield Prolog, that Prolog programs typically generate very large systems of equations to solve. The user generally has a much narrower view of what the problem of interest is, namely the original goal that s/he input. So Prolog does not output the entire solved form of the entire system of equations it has solved. Instead it just prints out the bindings of the variables which appeared in the original goal.

Systems of equations in solved form can be thought of equivalently as defining functions from variables appearing on the left of the equations to the expressions occurring on the right. So we can apply a unifier to an expression (as we do in the definition of variable elimination). We refer to such functions as *substitutions*, and we write them as *postfix* operators. So, if $\theta$ is the substitution function defined by some solved system of equations, then we may write $E\theta$ for the result of applying $\theta$ to an expression $E$.

Here are some examples of pairs of expressions, their mgu's (thought of now as substitution functions and written using the Variable/Value notation), and the result of applying the mgu to the expressions.

1. $f(X)$, $f(f(Y))$, $\sigma = \{X/f(Y)\}$,
   $f(X)\sigma = f(f(Y))\sigma = f(f(Y))$
2. $0 + X$, $N + 1$, $\sigma = \{N/0, X/1\}$,
   $(0 + X)\sigma = (N + 1)\sigma = (0 + 1)$
3. $apply(f, a, Term)$, $apply(f, Arg, f(Arg))$,
   $\sigma = \{Arg/a, Term/f(a)\}$,
   $apply(f, a, Term)\sigma = apply(f, Y, f(a))\sigma = apply(f, a, f(a))$

The following pairs of expressions do *not* unify.

1. $a$, $f(X)$: $a$ and $f$ are not identical symbols.
2. $X$, $f(X)$: "occurs-check" violation—$X$ occurs in its own value.
3. $f(X)$, $f(X, Y)$: nothing to unify $Y$ with.

## 3.3 Prolog: Resolution with Unification

### 3.3.1 The language of First Order Logic

First-order logic uses expressions, roughly as we have defined them here, to play two distinct roles: as names for individuals (*terms*) and as statements of relations holding among individuals (*formulas*). At a certain level, there is no formal difference between terms and formulas. As *data structures* they are identical: they are all instances of expression trees. So, for example, it is possible to unify two expressions regardless of whether they are both terms or both formulas. However there is in first order logic a crucial *semantic* difference between them. For example, the term $2 * 2$ combines two numbers to yield another number. It has a definite *value*; in fact, we can think of $2 * 2$ as just another name for 4. Or, to be excruciatingly precise, both $2 * 2$ and 4 (and "four") are names for some number which, of course, we cannot speak of without naming it one way or another. On the other hand, a formula like $even(4)$ (equivalently $even(2 * 2)$) does not name another number. In this case it names a property, or rather, states a claim that that property (the property of being even, or divisible by 2) holds of (the number denoted by) 4.

So, in defining a first-order language to express a particular problem of interest, solutions to the problem, and proofs of the solutions, we begin by sorting our heap of primitive expressions into two distinct piles: *predicate-names* and *function-names*. Furthermore we associate with each symbol a number, called its *arity*, which indicates how many arguments it combines with to form a well-formed expression of the appropriate sort. A collection of symbols sorted this way is often called a *ranked alphabet* or a *signature*. Concretely, when presenting a signature we will write function and predicate names followed by their arities as, e.g., $f/2$. Keep in mind that $f/1$, the one-place function or predicate symbol, is (logically speaking) a *different symbol* from $f/2$.

Note that we will usually refer to function names with zero arity as "constant terms" or just "constants." Note also that we do not consider variables as part of the signature; we always assume that we have as many of them as we need, and so there is no need to name them in advance, and they never take arguments (in first order logic), so there is no need to assign them arities.

The rules of our syntax are as follows. From the symbols in our signature, together with a collection of variables, we will construct *terms*. Terms are expression trees whose roots are labeled with variable names, constant names or function names, and which have exactly the number of daughters specified by the root's arity. (So variables and constant terms, having zero arity, will have no subexpressions.) Each subterm must also be a term.

> A *term* $T$ is an expression satisfying one of the following conditions:
>
> - T is a variable;
> - T is a constant-name, that is, a function-symbol of arity 0.
> - T is an expression-tree whose root is labeled by a function-symbol $f$, with arity $n$, with exactly $n$ subexpressions, each of which is itself a term.

Given a signature in which $s$ is a function symbol of arity 1 and 0 is a constant, 0, $s(0)$, and $s(s(0))$ are all terms. $s(0)$ has a single subterm, 0, and $s(s(0))$ likewise has a single subterm, $s(0)$. Terms are interpreted as names for things. Suppose, for the moment, that we want to be able to talk about natural numbers and elementary arithmetic, so we want our terms to name numbers. So, for example, we intend that $s(0)$ name the number one (i.e., the successor of the number zero). As noted, first order logic is distinguished from higher order logics in requiring that variables all have zero arity. So $X$ is a term, but $X(0)$ is not any sort of well-formed expression of first order logic. (If you try and type it into the Prolog interpreter, you will get an error message!)

From terms we can go on to define *atomic formulas* as follows.

> An *atomic formula* is an expression tree whose root is a predicate-symbol $p$ of arity $n$, with exactly $n$ subexpressions each of which is a term.

This sounds a lot like the definition of a term! Note, however, that terms are defined recursively (a term is made out of other terms), but atomic formulas are not (an atomic formula is *not* made out of other atomic formulas).

In standard first-order logic, we would go on to define *well-formed formulas* as collections of atomic formulas bound together by the *connectives* $\wedge$, $\Rightarrow$, etc. and attached under the *quantifiers* $\forall$ and $\exists$. For example,

$$\forall X (\exists Y\, eq(X, s(Y)) \Rightarrow \neg eq(X, 0))$$

would be a formula of standard first order logic using our signature (with *eq* a predicate symbol of arity 2). However, as we already know, Prolog uses (part of) the logically equivalent, but formally different writing system of clausal form. In clausal form we do not represent the connectives and quantifiers directly, so I will not go into any further detail about the standard writing system of first order logic.

### 3.3.2 Clausal Form for Predicate Logic

In the case of the predicate calculus, a clause is still an expression of the form:

$$A_1, \ldots, A_m \leftarrow B_1, \ldots, B_n,$$

but now each of the $As$ and $Bs$ are *atomic formulas*. Beyond that, we just have to deal with the fact that the atomic formulas in predicate clauses can contain variables as subexpressions. In this case, the variables are treated as if they were all bound by universal quantifiers ($\forall$) placed at the front of the clause. So, for example,

$$entity(X) \leftarrow$$

means the same thing as the standard first order logic sentence

$$\forall X\, entity(X)$$

namely, "for all $X$, $X$ is an *entity*." Similarly, a clause like

$$mortal(X) \leftarrow human(X)$$

means the same thing as

$$\forall X (human(X) \Rightarrow mortal(X))$$

that is, "for all $X$, if $X$ is human then $X$ is mortal," and the clause

$$\leftarrow different\_from\_self(X)$$

has the same meaning as

$$\forall X \neg different\_from\_self(X).$$

Note that $\forall X \neg \phi$ is equivalent to $\neg \exists X \phi$. (That is, "for all $X$ $\phi$ does not hold," is equivalent to "there is no $X$ such that $\phi$ holds.") For this reason it is often useful to think of variables occurring only to the right of the arrow as being implicitly *existentially* bound. Accordingly, the following clause

$$ancestor(X, Y) \leftarrow parent(X, Z), ancestor(Z, Y)$$

can either be read as "for all $X, Y, Z$, either $X$ is an ancestor of $Y$ or else $Z$ is not a parent of $X$ or else $Z$ is not an ancestor of $Y$," (its literal meaning) or it can be read as "for all $X, Y$, $X$ is an ancestor of $Y$ if there is a $Z$ such that $X$ is a parent of $Z$ and $Z$ is an ancestor of $Y$."

Just as our interpretation of propositional clauses allowed us to do away with connectives, our interpretation of predicate clauses allows us to do away with quantifiers.

The definitions of Horn clause, definite clause, rule, fact, goal and empty clause remain the same for predicate clauses. Proplog programs were sets of propositional definite clauses. Prolog programs are sets of (first-order) predicate definite clauses.

### 3.3.3 The Resolution Rule for Predicate Clauses

Just as with the definition of clauses, we do not change anything in the definition of the Resolution Rule except to add to it mechanisms to deal with variables.

We define first a *renaming substitution*: it is a substitution which simply maps variables to new variables. If we are considering a pair of clauses that we would like to resolve, we require that the set of variables they involve must not overlap; each must have its own separate set of variables. This is not really a very strong requirement, though, since if they both mention a variable named $X$ we can just apply a renaming substitution to one of them. This is called *naming apart*, and after naming apart we call the pair of clauses *separated*. Resolution is only defined for a pair of separated clauses.

In defining resolution for predicate clauses, we now add a *constraint pool*, where we keep track of all of the unification problems we have solved. The constraint pool acts as an *environment*, telling us the current value of all of our variables. As before, the first step in the resolution process is to select an atom from the current goal. Our second step, after selecting an atom from the current goal clause, is to search the program for a clause whose head *unifies* with the selected sub-goal. (Note that this covers the case of "matching" atoms in the propositional subcase.) In the process of finding a clause whose head unifies with the currently selected atom, we will of course find the mgu for the clause-head and the goal-atom. The resolvent of the goal with the clause we select is the result of (a) removing the selected member of the goal clause, (b) replacing it with the body of the selected program clause, and (c) adding the mgu to the constraint pool. Note that high-performance Prologs do not usually bother to keep the constraint pool in solved form; in particular, variable elimination is not usually used. Given $\{X = Y, Y = a\}$, a Prolog interpreter can simply follow the obvious chain of inferences to get from $X$ to its current value, $a$.

Interpreted as an environment or substitution giving the current values of all of our variables, we can think of step (c) alternatively as applying the sub-

**Table 5** Prolog's version of the Resolution Rule

1. Given: A goal clause $\leftarrow G_1, \ldots, G_k$ and a constraint pool $C$.
2. Search the program for a (variant of a) program clause $A \leftarrow B_1, \ldots, B_n$ such that $G_1$ unifies with $A$ with $\sigma$ as their most general unifier (mgu). If no such clause exists, then the resolution step fails.
3. Replace $G_1$ with $B_1, \ldots, B_n$.
4. Replace $C$ with the solved form of the system $\sigma \cup C$. Thinking of $\sigma$ as a substitution function, the new goal equals

$$\leftarrow (B_1, \ldots, B_n, G_2, \ldots, G_k)\sigma$$

stitution we have built up to the new goal. The point is, if we select an atom containing a variable `X`, and unify it against the head of some program clause, with the result that we now have new information about the value of `X`, we must keep in mind that that new information applies to every instance of `X` in the goal, not just in the selected atom.

As before, if we resolve the goal with a fact, then we remove the selected atom from the goal. Of course, unifying the selected atom with the fact may create new variable bindings which need to be added to the constraint pool.

A resolution derivation is just the repeated application of resolution steps. A resolution derivation succeeds just in case we eventually derive the empty clause. In that case, the *solution* of the original goal is the bindings for the variables that appeared in that goal taken from the final state of the constraint pool.

Suppose for example that we want to resolve the two clauses below. (The $s$-function is meant to be interpreted as the arithmetic successor function; that is, $s(N) = N + 1$.)

$$\leftarrow plus(s(0), s(s(0)), X). \tag{3.1}$$

$$plus(s(X), Y, s(Z)) \leftarrow plus(X, Y, Z). \tag{3.2}$$

The clause in (3.1) is a goal clause, and the clause in (3.2) (a rule clause) is meant to come from a program defining addition for natural numbers. Note that clause (3.2) is indeed a true sentence about the natural numbers: if $X + Y = Z$, then certainly it is true that $(X + 1) + Y = Z + 1$.

Keep in mind that we must name the clauses apart before we do anything else. (In particular, notice that the variable name $X$ has gotten used in both clauses.) So in effect we are resolving the following two clauses.

$$\leftarrow plus(s(0), s(s(0)), X_1). \tag{3.3}$$
$$plus(s(X_2), Y_2, s(Z_2)) \leftarrow plus(X_2, Y_2, Z_2). \tag{3.4}$$

There is only one member of the goal clause (3.3), so we select that. It does indeed unify with the head of the program clause (3.4). Their mgu is

$$\{X_1 = s(Z_2), X_2 = 0, Y_2 = s(s(0))\},$$

and this becomes the value of our current environment. Therefore our resolvent will be the new goal clause:

$$\leftarrow \mathit{plus}(X_2, Y_2, Z_2) \tag{3.5}$$

which, in the current environment, equals

$$\leftarrow \mathit{plus}(0, s(s(0)), Z_2) \tag{3.6}$$

Suppose for the moment that our program also contains (some variant of) the clause

$$\mathit{plus}(0, X_3, X_3) \leftarrow \tag{3.7}$$

which is another true statement about the natural numbers and addition: the sum of zero and anything is that thing again. We can resolve the current goal with this clause with the mgu

$$\{X_3 = s(s(0)), Z_2 = s(s(0))\}.$$

Adding this to our constraint base, we get

$$\{X_1 = s(Z_2), X_2 = 0, Y_2 = s(s(0)), X_3 = s(s(0)), Z_2 = s(s(0))\},$$

which, in solved form, would be

$$\{X_1 = s(s(s(0))), X_2 = 0, Y_2 = s(s(0)), X_3 = s(s(0)), Z_2 = s(s(0))\}.$$

We have derived the empty clause, so our derivation succeeds. The answer to our original goal, in which $X$ (renamed to $X_1$) was the only variable, will be reported as "`X=s(s(s(0)))`. According to our intended interpretation of this notation, this is the correct answer. We have proved that $1 + 2 = 3$. (Three cheers!)

Unification is an extremely versatile operation, and is used in Prolog to implement

- the construction of new expressions from old,
- the decomposition of one expression into several component expressions,
- testing, branching and reasoning by cases,
- parameter passing.

In the above example, unification was used to build up the value of the goal variable $X$: first we unified $X$ with $s(Z_2)$, then we unified $Z_2$ with $s(s(0))$. We also used unification to take apart the original first term of the addition: unifying $s(0)$ with $s(X_2)$ in effect stripped off the first (and only) $s$ from $s(0)$, leaving behind the solution for $X_2$. We also used unification for branching. Our first selected atom unified with the first program clause and not with the second ($s(s(0))$ does not unify with 0). The situation was reversed in the next step: our selected atom unified only with the head of the second clause (0 does not unify with $s(X)$, for any $X$). Parameter passing by unification is pervasive in this and every example of SLD resolution; every time we unified a selected atom with the head of a program clause, we passed information from the goal down to the program clause by constraining the values of its parameter variables.

## 3.4 Exercises

1. What is the type of each of the following symbols?

   (a) *Pi*

   (b) '*Variable*'

   (c) *_constant*

2. Which of the following are (well-formed) expression trees?

   - 3
   - 3(2)
   - '3'(2)

3. Assume that we have defined an infix operator that we can write as *'s*. Compute the value of the following expressions, given the accompanying equational systems.

   (a) X — { X=(Y 's car), Y=(Z 's uncle), Z=john }

   (b) (X 's Y) — { X=(john 's Z), Y=car, Z=uncle }

   (c) (X 's Y) — { X=(Z 's W), Y=U, Z=john, W=uncle, U=car}

4. Compute the mgu's for the following pairs of expressions, and apply them to the expressions.

   (a) $less\_than(X, ((0+1)+1))$, $less\_than(N, N+1)$

   (b) $agr(pers(third), num(N))$, $agr(P, num(sing))$

   (c) $term(f(a,b), C, T1, T2)$, $term(f(X,Y), f, X, Y)$

5. Use truth tables to prove that $p \Rightarrow q$ is equivalent to $\neg p \vee q$.

| $p$ | $q$ | $p \Rightarrow q$ | $p \vee q$ | $\neg p$ |
|---|---|---|---|---|
| $t$ | $t$ | $t$ | $t$ | $f$ |
| $t$ | $f$ | $f$ | $t$ | $f$ |
| $f$ | $t$ | $t$ | $t$ | $t$ |
| $f$ | $f$ | $t$ | $f$ | $t$ |

6. Convince yourself that the following statements are logically equivalent.

   (a) $(A_1 \vee \ldots \vee A_m)$ is true.

   (b) For some $i$, $1 \leq i \leq m$, $A_i$ is true.

7. Convince yourself that the following statements are logically equivalent.

   (a) $(B_1 \wedge \ldots \wedge B_n)$ is false.

   (b) For some $j$, $1 \leq j \leq n$, $B_j$ is false.

8. Convince yourself that the following statements are logically equivalent.

   (a) $\neg\phi$

   (b) $\leftarrow \phi$

# Part II

# Logic

# Chapter 4

# Anatomy of a Prolog Program

Now we have all of the pieces we need. Let's put together a Prolog program and see how it works. We will begin with an extremely simple, almost trivial program.

```
eq( X, X ).
```

> *Prolog Syntax Note*: In Prolog we write fact clauses (more usually called *unit clauses*) without the arrow, and terminated by a period. So this piece of (Edinburgh) Prolog syntax corresponds to the (mathematical) clause "$eq(X, X) \leftarrow$".

On the face of it, this program is useless. It just tells us what everybody already knows: everything equals itself. But in fact it gives us a little more. Because of the way Prolog works, this predicate gives us direct access to Prolog's unification operation. Consider for example the following goal clause.

```
?- eq(sum(0,4,X1), sum(0,X2,X2)).
```

> *More Prolog Syntax*: In these notes, goal clauses are written with a `?-` symbol instead of the expected $\leftarrow$. The `?-` symbol is the Prolog interpreter's prompt. *You don't type this!* The interpreter will type it for you. Note that all clauses are terminated with a period.

Following the description of the Resolution Rule outlined in the last chapter, the first thing we need to do is to find a program clause whose head unifies with the goal. Our program has only one clause in it, so we don't have far to look! Now, the selected goal expression and the head of our selected program clause unify if and only if there is a substitution which makes them identical. So we are presented with the following unification problem:

$$eq(sum(0,4,X1), sum(0,X2,X2)) = eq(X,X) \ .$$

Applying the term decomposition rule, we derive the new equations

$$X = sum(0, 4, X1), X = sum(0, X2, X2) \ .$$

We can apply variable elimination using either equation; we will use the first. We add the new equation

$$sum(0, 4, X1) = sum(0, X2, X2)$$

to the system. We can apply term decomposition again, yielding the following new equations.

$$0 = 0, X2 = 4, X1 = X2$$

The first is trivial and can be deleted from the system. Then we eliminate $X2$, to yield the following.

$$X1 = 4$$

Eliminating $X1$ yields a system in solved form. In particular,

$$X1 = 4, \ X2 = 4 \ .$$

There is another way to arrive at a solution to the unification problem. In this case, we think of the problem in the following terms.

$$eq(sum(0, 4, X1), sum(0, X2, X2))\theta = eq(X, X)\theta$$

We are to solve for the unknown substitution $\theta$. Scanning these two expressions from left to right, we find that we first require the substitution

$$\theta_1 = \{X/sum(0, 4, X1)\}.$$

Applying this substitution, we reduce our unification problem to the following simpler problem.

$$eq(sum(0, 4, X1), sum(0, X2, X2))\theta = eq(sum(0, 4, X1), sum(0, 4, X1))\theta$$

We know in addition that $\theta$ must satisfy the additional constraint that it include the substitution $\theta_1$ which we have just discovered. (Formally, we know that $\theta$ must satisfy the equation $\theta = \theta_1\sigma$ for some yet-to-be-discovered $\sigma$.)

Now, considering the second argument position, the roots of the two expressions are identical (they are both $sum$), and the first arguments of the $sum$-expressions also unify (they are both constants and identical). The first point at which our two expressions differ is where one contains $X2$ and the other contains 4. Since $X2$ is a variable, we know we need the variable binding $X2/4$. Applying the new substitution $\theta_2 = \{X2/4\}$, we reduce the unification problem to the following:

$$eq(sum(0, 4, X1), sum(0, 4, 4))\theta = eq(sum(0, 4, X1), sum(0, 4, X1))\theta$$

where, in addition, $\theta$ must satisfy $\theta = \theta_1\theta_2\sigma'$, for some yet-to-be-discovered $\sigma'$. (So you can see how we are building up our solution little by little as we learn more about the bindings of our unknowns.)

Finally, we discover the binding $X1/4$ and we are done. The solution to our orginal goal is found by composing all of the substitutions we have discovered along the way, and applying the resulting substitution to the variables that we originally entered (i.e., $X1$ and $X2$). That is,

$$\begin{aligned}\theta &= \theta_1\theta_2\theta_3 \\ &= \{X/sum(0,4,X1)\}\{X2/4\}\{X1/4\} \\ &= \{X/sum(0,4,4), X2/4, X1/4\}\end{aligned}$$

and so now the answer is explicit: $X1 = 4, X2 = 4$.

Logically speaking we have in effect asked Prolog if there exist values for $X1$ and $X2$ which make true the formula

$$eq(sum(0,4,X1), sum(0,X2,X2)).$$

Prolog has discovered that our program has among its consequences that if $X1 = 4$ and $X2 = 4$ are true, then our original goal is true as well.

So we see that this program is not completely trivial. When we give it two inputs which are not in fact identical, but are unifiable, this one-clause program will actually unify them and produce as an answer their most general unifier:

$$\begin{aligned}X1 &= 4 \\ X2 &= 4\end{aligned}$$

This example is a little subtle, and you should be certain you understand it because it expresses a very important basic feature of Prolog: *Equality* means *unifiability.* In fact, the meaning of the 2-place predicate symbol '=' (or rather '`=/2`') which is built into Prolog is exactly the meaning we have given `eq/2`. That is, `=/2` could have been (and in some Prologs it is) defined by the single fact-clause

```
X = X.
```

For the moment we will continue to use `eq/2`, but the reader should be prepared for us to switch back into standard Prolog without further comment.

Now we can extend this simple program a little. The following clauses implement a bit of Number Theory in Prolog. The intended interpretation of the predicate name `nn/1` is "natural number".

```
eq(X,X).

nn(X) :- eq(X,0).
nn(X) :- eq(X,s(Y)), nn(Y).
```

*More Prolog Syntax*: This predicate introduces rule clauses. In rule clauses, the arrow is represented '`:-`'. Once again, rule clauses like all clauses are terminated with a period.

Suddenly, just by adding this single predicate to our program, we have a lot of new things to talk about.

We intend by these two clauses to provide a *definition* of the predicate `nn/1`. First, note that the definition is in *two parts.* In programming languages like Pascal or C when we define a procedure or function we provide a unique definition of that procedure or function. Languages like Pascal and C are *deterministic.* If we execute a call to a procedure $p$, we do not expect the program to have to decide which $p$ we might mean. Prolog is *non-deterministic.* Logically speaking, we have no right to presume that the Prolog interpreter will select one or the other clause; good programming style demands that we be prepared for either choice. In fact, of course, since Prolog is implemented on a deterministic computing machine, its choice is entirely predictable: it will always try the first clause first.

So far so good, but what if the selected clause (i.e., the first clause) doesn't work? More concretely, suppose we type in the goal clause

```
?- nn(s(0)).
```

The goal unifies with the head of the first clause yielding the new goal clause

```
?- eq(s(0),0).
```

This goal cannot be solved by this program: the two expressions $s(0)$ and 0 cannot be unified, so they cannot be made `eq`. Put another way, the goal is not a logical consequence of the program.

When this happens, Prolog *backtracks.* The Prolog interpreter keeps track of every point of the computation at which it had a choice to make. When the computation fails, as in this example, the Prolog interpreter checks its list of choice-points and returns to the most recent one, and proceeds by making a different choice of program clause. If it runs out of alternatives without finding one that works, then it must backtrack even further to a previous choice-point. Only when it runs out of alternatives and choice-points does it finally give up and report that the problem has no solution.

In the current example, the attempt to solve `eq(s(0),0)` will fail, so the interpreter will look for alternatives. There is only one way to solve `eq/2` goals, so there are no alternatives to try there. The last choice-point was in solving the `nn/1` goal. So the interpreter will backtrack to that point, so that the current goal once again is

```
?- nn(s(0)).
```

Now it will try solving that goal using the other clause in the definition of `nn/1`, so the new goal clause will be

```
?- eq(s(0),s(Y)), nn(Y).
```

This new goal clause introduces another feature of Prolog. The second clause defines `nn/1` *in terms of itself*. This kind of definition is called *recursive*. On the surface it looks like we have introduced a piece of circular reasoning, the deductive equivalent of an infinite loop. But this is not in fact the case. To see that this is so, let us continue with the example, and see what happens. Once again, the new goal is

```
?- eq(s(0),s(Y)), nn(Y).
```

Prolog will select the leftmost sub-expression of the goal, namely

`eq(s(0),s(Y))`.

We can see that this has a simple solution which will require that $Y$ be equal to 0.

> *Exercise*: Work out this subproblem in detail and verify that this is indeed the solution.

The definition of `eq/2` is a *fact*, that is, a clause with no body, so after solving `eq(s(0),s(Y))`, we simply remove it from the goal, and we derive the new goal

```
?- nn(0).
```

This has an immediate solution using the first clause of the definition. We have reduced our problem to facts; we have no more goals to solve, and so we have successfully established that $nn(s(0))$ is indeed a consequence of our program. Note that *only* the first clause will succeed for the goal ?- `nn(0)`. The second clause will fail. (Why?) So in fact there is no infinite loop. The program halts and reports the correct answer. This example is summarised in Table 4.1.

The reason that this apparently circular code actually halts is the key to the successful use of recursion in writing computer programs. In a certain sense that we can define rigorously, every time we called `nn/1`, the problem we wanted to solve got *smaller*. Ultimately, it got small enough so that we could solve it directly, using the (non-recursive) first clause.

*Every example of a (well-written) recursive predicate that you will ever see has the same basic structure.* The set of clauses defining a particular predicate must include some which are *not* recursive. These clauses form what is called the *base* of the recursion. In this example, and indeed in most examples, there is a single base-clause: the first clause in the definition of `nn/1`. The second clause in the definition of `nn/1` is an example of what is called the *recursive step*, the processing step in which the procedure calls itself. In this case, the call to `eq/2` which precedes the recursive call to `nn/1` does two things. First, it assures that the value of $X$ is in fact of the form *s(Y)*, that is, $X$ is the application of the $s$-function to some argument. Second it actually extracts that argument as the value of $Y$.[1] Then `nn/1` calls itself again, but this time on $Y$ instead of $X$.

[1] This is an example of using unification in the first place for reasoning by cases—the first clause will only succeed if $X$ is of the form 0; the second will only succeed if $X$ is of the form

**1.** Current Goal: `?- nn(s(0)).`
Selected Clause: `nn(X1) :- eq(X1,0).`
Solution Substitution: $\{X1/s(0)\}$

**2.** Goal: `?- eq(s(0),0).`
Clause: `eq(X2,X2).`
Solution: (Fails—no substitution can be found.)

**2.** (Backtracking ... )
Goal: `?- eq(s(0),0).`
Clause: (Fails—no more clauses for eq/2.)

**1.** (Backtracking ... )
Goal: `?- nn(s(0)).`
Clause: `nn(X1) :- eq(X1,s(Y1)), nn(Y1).`
Solution: $\{X1/s(0)\}$

**2.** Goal: `?- eq(s(0),s(Y1)), nn(Y1).`
Clause: `eq(X2,X2).`
Solution: $\{X2/s(0), Y1/0\}$

**3.** Goal: `?- nn(0).`
Clause: `nn(X3) :- eq(X3,0).`
Solution: $\{X3/0\}$

**4.** Goal: `?- eq(0,0).`
Clause: `eq(X4,X4).`
Solution: $\{X4/0\}$

**5.** Goal: (The goal is empty; we're done!)

Table 4.1: Summary of the computation of the goal `?- nn(s(0)).`

```
 ?-nn(X).
     |
 ?-eq(X,0)
```

Answer: `X = 0`

Figure 4.1: One way of answering `?- nn( X ).`

Every time we extract the argument of $s$ in $X$ we wind up with a strictly smaller expression—if $X$ is finite (which we can safely assume that it is, since a user is presumed to have typed it in in less than a whole lifetime!), then clearly we can do this only a finite number of times before there is nothing else to extract.

> *Exercise*: Neither (a) *s(s(s(a)))* nor (b) *s(s(s(f(f(f(a))))))* is a natural number, according to our definition. Therefore, at some point processing must fail in each case. How many calls to `nn/1` will be necessary before failure is detected?

We can think of the predicate `nn/1` as defining a new *data type*. However, in many ways our logically-based definition of this data type is more useful than the sorts of definitions we find in other programming languages. We can use it either to *recognize* a particular data item as being of type *natural number*, (as in the example, where we essentially checked that *s(0)* was indeed of the type natural-number)—this is the standard task of data type definitions—or we can use it to *generate* new data items of the that type. Consider now the goal:

```
?- nn( X ).
```

We can think of this goal as expressing the question "Which X's are natural numbers?" Obviously, if we have written our definition correctly, there should be an infinite number of answers, so we will not consider all the possible proofs of this goal here, but here are a few, presented as *proof trees*, in figures 4.1–4.3. Proof trees should be read as follows: nodes are labeled with goals; if a node is labeled $A$ and its daughters are labeled $B_1, \ldots, B_n$, then there must exist a program clause of which `A :- B`$_1$`, ... ,B`$_n$ is an instance; that's the clause that was resolved against at this point in the proof.[2]

Considering figures 4.1–4.3, one can easily see how an infinite number of proofs could be constructed, each yielding a different value for $X$.

*s(Y)* for some $Y$; all natural numbers are of one of these two forms, and no natural number is of both forms, so the call to `eq/2` will cause the program to behave deterministically, even though in general Prolog clauses are non-deterministic. The same application of unification is also used to take the data structure which is the value of $X$ and access one of its components, which becomes the value of $Y$. So we are doing both case-switching and data access all in one step—you may get some idea of the power and versatility of the unification operation!

[2]Linguists: Note that we can think of program clauses (rule clauses, in particular) as grammar rules of the form $A \longrightarrow B_1 \ldots B_n$, and proof trees as the derivation trees assigned to strings of fact clauses by this "grammar".

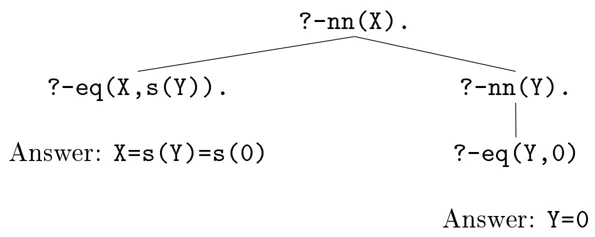


Figure 4.2: Another way of answering `?- nn( X ).`

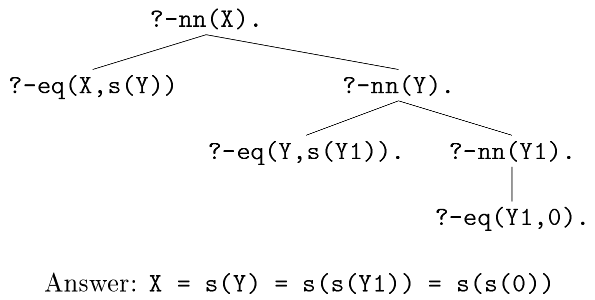


Figure 4.3: Yet another way of answering `?- nn( X ).`

Now let us extend our program yet again, introducing a new 2-place predicate symbol `lt/2`, which we intend to mean "less than".

```
lt(X,Y) :-
      eq(X,0),
      eq(Y,s(_)).
lt(X,Y) :-
      eq(X,s(X0)),
      eq(Y,s(Y0)),
      lt(X0,Y0).
```

> *More Prolog Syntax*: The underscore character appearing in the first clause of `lt/2` is the *anonymous variable.* Every time it appears it causes Prolog to generate a new variable that it hasn't used yet. Note that `f(X,X)` implicitly constrains both arguments of $f$ to be identical; this is *not* true of `f(_,_)`, quite the contrary! The anonymous variable is used in this example because we are only interested in pattern-matching. We are not actually interested in the value of the argument of $s$; we only wish to ensure that it has one. Note that the meaning of this definition would not change if we replaced "_" with $Z$, say, but the code is more readable this way.

The definition of `lt/2` presents us with no new problems or concepts. It is based on the fact that for any pair of natural numbers $X$ and $Y$,

$$X < Y \iff 0 < (Y - X).$$

Zero is less than any natural number other than zero, and all natural numbers other than zero match the pattern `s(_)`, that is, they are all successors of some other natural number. Zero is the only natural number which is not someone else's successor. The base clause expresses this fact. What remains is to prove that, given arbitrary inputs for `X` and `Y`, we will only reach the base of the recursion when the second argument has been reduced to `Y` minus `X`.

We actually prove this fact backwards. If a given goal is truly an instance of the less-than relation, that is, for given $X$ and $Y$, if it truly is the case that $X < Y$, then it must be a fact that $0 < (Y - X)$. Now it is also a fact that

$$X < Y \Rightarrow (X + 1) < (Y + 1),$$

for any pair of natural numbers $X$ and $Y$. So if $0 < (Y - X)$, then $0 + 1 < (Y - X) + 1$ (which equals $(Y + 1) - X$ of course), and $0 + 1 + 1 < (Y + 1 + 1) - X$, and so forth. If we do this exactly $X$ times, we will have established that $0 + X < (Y + X) - X$, that is, $X < Y$.

Though rather informal, this style of reasoning closely parallels mathematical proofs by *induction*. There is, in general, a strong connection between inductive proofs and recursive definitions (in fact, sometimes recursive definitions are called inductive definitions, especially in older works). This is why so many computer scientists favor recursion as a way of implementing repetetive computational procedures: the recursive style of definition makes available all the tremendously powerful tools that mathematicians have developed for doing proofs by induction.

Consider now the definition of `lte/2`, given below, and intended to mean "less than or equal to."

```
lte(X,Y) :- lt(X,Y).
lte(X,Y) :- eq(X,Y).
```

Since we already have definitions of "less than" and "equal to" all we need is some way of representing "or". We get this from the non-determinism of Prolog. Prolog is free to search for a solution using either clause, and as long as a solution exists using one of these clauses, Prolog will find it. A little more formally, a Prolog program is equivalent to a conjunction of implications like the following.

$$(lt(X, Y) \Rightarrow lte(X, Y)) \wedge (eq(X, Y) \Rightarrow lte(X, Y))$$

This is provably equivalent to the formula

$$(lt(X, Y) \vee eq(X, Y)) \Rightarrow lte(X, Y).$$

*Exercise*: Consider the following laws from propositional logic.

**DeMorgan's Laws** 1. $\neg(p \wedge q) \Leftrightarrow (\neg p \vee \neg q)$

2. $\neg(p \vee q) \Leftrightarrow (\neg p \wedge \neg q)$

**Distributivity** 1. $p \wedge (q \vee r) \Leftrightarrow (p \wedge q) \vee (p \wedge r)$

2. $p \vee (q \wedge r) \Leftrightarrow (p \vee q) \wedge (p \vee r)$

**Implication Defined** $(p \Rightarrow q) \Leftrightarrow (\neg p \vee q)$

Use them to show that

$$((q \Rightarrow p) \wedge (r \Rightarrow p)) \Leftrightarrow ((q \vee r) \Rightarrow p).$$

Finally, consider the following definition of `plus/3`, a 3-place predicate defining addition.

```
plus(X,Y,Z) :-
      eq(X,0),
      eq(Y,Z).
plus(X,Y,Z) :-
      eq(X,s(X0)),
      eq(Z,s(Z0)),
      plus(X0,Y,Z0).
```

This code implements the addition operation on natural numbers. Normally we think of addition as a two-place function, rather than a (three-place) predicate. In Prolog, "functions", that is, term-expressions other than constants and variables, are data structures, and "real" functions are computed via predicates. This situation is unusual enough to warrant treatment at some length.

## 4.1 A Note on Functions and Predicates

As we just noted, functions, that is, compound terms or non-trivial expression trees with function-symbols labeling their roots, are *data structures*. They are not *evaluated*. So for example the goals

```
?- p( 2+2 ).
```

and

```
?- p( 4 ).
```

are completely different goals. Prolog will not evaluate the expression $2 + 2$ unless you tell it to, explicitly. That is, unless you provide Prolog with a theory of arithmetic which has among its consequences that $2 + 2$ is (arithmetically) equal to 4, then all Prolog knows is that these are two distinct expression trees. The expression $2 + 2$ is *semantically* identical to the expression 4, at least if we interpret '+', '2' and '4' in the normal way, but it is *syntactic* identity that Prolog cares about.

In Prolog it is the *predicates* that actually get evaluated, so our intuitive notion of a function is usually implemented in Prolog as a predicate. There is no logical difficulty with this. Formally, we can always write

$$f(x_1, \ldots, x_n) = x_{n+1}$$

as the predication

$$p(x_1, \ldots, x_n, x_{n+1})$$

as long as we then define it in such a way that it is true, given $x_1, \ldots, x_n$, just in case (a) there is a unique $x_{n+1}$ making $p$ true and (b) the value of that $x_{n+1}$ is exactly the value of $f(x_1, \ldots, x_n)$. That is, $p$ is defined so as to be true only in the case where

$$p(x_1, \ldots, x_n, f(x_1, \ldots, x_n)).$$

This is what we are doing with the predicate `plus/3`. We say "let `plus/3` be defined so that, for natural numbers $X$, $Y$, and $Z$, `plus(X,Y,Z)` is true whenever $Z = X + Y$." Here is the code for `plus/3` again.

```
plus( X, Y, Z ) :-
      X = 0,
      Z = Y.
plus( X, Y, Z ) :-
      X = s(X0),
      Z = s(Z0),
      plus( X0, Y, Z0 ).
```

Here, in the standard functional language of ordinary mathematics, are the conditions this code is meant to establish, more or less as they would appear in a text on number theory.

$$0 + Y = Y \tag{4.1}$$

$$s(X0) + Y = s(X0 + Y) \tag{4.2}$$

It will now be our task to prove that the Prolog code corresponds to these specifications. This is an important goal in its own right: we must always be prepared to prove that our programs are correct implementations of their specifications. Here, however, we are more interested in the proof because it sheds a lot of light on how to translate back and forth between the language of functions and the language of predicates.

We begin with the first clause. We want to show that, if $X = 0$, then $X + Y = Y$. We know from the intended interpretation of the predicate that

$X + Y = Z$, that is, $Z$ is intended as a name for the sum of $X$ and $Y$. So $X + Y = Y$ and $X + Y = Z$ can only both hold if $Y = Z$, which is exactly the condition our program enforces. Put another way, we have shown that if we begin our computation in a state in which $X = 0$, then, after executing a call to `plus(X,Y,Z)`, we will wind up in a state in which $Y = Z$. Put yet another way, we are guaranteed that, for natural numbers $X$, $Y$, and $Z$, the following Prolog goal will always succeed.

```
?- X=0, plus(X,Y,Z), Z=Y.
```

It would be nice if we could just use Prolog in the same way to prove the correctness of the second clause. We would then like to be able to prove the goal:

```
?- X=s(X0), plus(X,Y,Z), Z=s(X0+Y).
```

Or, more compactly,

```
?- plus(s(X0),Y,s(X0+Y)).
```

This goal corresponds almost exactly to the condition in (4.2). But we know already that this goal will not succeed, because `X0+Y` is syntactically distinct from the representation of the number which is the sum of the numbers denoted by $X0$ and $Y$. It is, in fact, syntactically distinct from the representation of any natural number in our theory.

So here is the basic rule to follow. Wherever we mention an operation and we intend it to stand for its *value*, we must replace the operational expression (`X0+Y` in this example) with a variable, and call a predicate to compute the value of that variable from its arguments. In this example, we replace `X0+Y` with `Z0`, and call `plus(X0,Y,Z0)` to compute its value.

Note that we do not do this with expressions like `s(X0)` or `s(Z0)`. This is because we do not intend to evaluate the successor function. It is part of our representation of natural numbers. (Or, to put it another way, "natural numbers" are just a subset of the expression trees, namely all and only those expression trees which are either of the form `0` or of the form `s(X)`, where `X` is another member of the set of expression trees which we are calling "natural numbers".)

The same procedure applies in constructing the predicate `exp/3`, to implement exponentiation. The first clause

```
exp( X, Y, Z ) :-
      Y = 0,
      Z = s(0).
```

is straightforward, again. It transparently implements the arithmetical condition that $X^0 = 1$, and we could prove this directly in Prolog by issuing the goal

```
?- Y=0, exp(X,Y,Z), Z=s(0).
```

Or, more directly,

```
?- exp(X,0,s(0)).
```

But when `Y = s(Y0)` for some `Y0`, then things are complicated again. We would like to be able to establish that $X^{Y0+1} = X * X^{Y0}$. But we cannot issue a goal like

```
?- exp( X, s(Y0), X*(X^Y0) ).
```

Or rather, we can issue it, but it doesn't mean what we intend it to. Once again, this is because it includes expressions in places where what we really want is the *semantic value* of the expression. In particular, where we have written `X*(X^Y0)` we do not mean the expression tree whose root is labeled '*' and whose subtrees are the expression trees `X` and `X^Y0`. We actually mean the (successor-representation of) the natural number we get by multiplying `X` by the result of raising `X` to the `Y0` power.

So, we apply our strategy again. We replace the arithmetic expression with a variable, `Z`, and we call the relevant predicate to compute the value of `Z`: `times(X,X^Y0,Z)`. But wait! This call to `times/3` still contains an arithmetic expression—there is an expression tree where what we really want is (a data structure representing) the arithmetic *value* associated with the expression tree. So we must replace `X^Y0` with another variable, `Z0`, and call a Prolog predicate to compute the value of `Z0`: `exp(X,Y0,Z0)`. Now we are done. We have "unpacked" all the arithmetic expressions, all of the "function-calls" in our original specification, and we get the clause:

```
exp( X, Y, Z ) :-
     Y = s(Y0),
     times( X, Z0, Z ),
     exp( X, Y0, Z0 ).
```

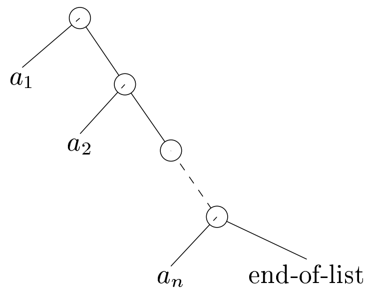
# Chapter 5

# Lists

Over and over again we will find it useful to arrange our symbolic expressions in *sequences*. For this purpose we will use the data type *list*. It is important to recognize, however, that the concept of a sequence is far more general than the concept of a list (by which we mean a particular data structure for representing sequences). So before plunging into the syntax of lists and the construction of list-processing predicates let us first be clear on the intended *semantics* of this structure. A sequence is a lot like a set, except that its elements appear in a fixed order, and the same symbol can appear any number of times. In order to define what a sequence is, we first have to have available to us an *alphabet* or *lexicon* of symbols, which will become the *members* of our sequences. A sequence over an alphabet $\Sigma$ can then be thought of as a function from the natural numbers to $\Sigma$. That is, we think of the sequence $a_1 a_2 \ldots a_n$ as being a way of writing a function $A$ which is such that $A(1) = a_1, A(2) = a_2, \ldots, A(n) = a_n$. If $n = 0$ then the sequence is empty. Given any two sequences $a_1 \ldots a_m$ and $b_1 \ldots b_n$ we can *concatenate* them to yield a new sequence $a_1 \ldots a_m b_1 \ldots b_n$.

A *list*, on the other hand, is essentially a uniformly right-branching expression tree such that every node has exactly two daughters, and the last terminal node (i.e., the rightmost terminal node) is labeled by a special official end-of-list symbol. The other nodes can be labeled with anything we please (even other lists). Figure 4 illustrates informally the kind of structure we have in mind. Note that the place of an element in a sequence is a function of its depth in the corresponding tree; only two elements have the same depth—$a_n$ and the end-of-list marker—and the end-of-list marker is not an official part of the corresponding sequence, so the mapping from depths to sequence position is indeed a function. Or rather, there is a unique correspondence between "official" leaves on the tree and the natural numbers (namely depth) and a unique correspondence between sequence elements and the natural numbers (sequential position), and they both give the same result. So we know (just in case it wasn't already intuitively obvious!) that we can use this sort of tree to represent a sequence.

**Figure 4** An expression tree with the structure of a list.




## 5.1 How to write down lists

Lists are so common in Prolog that the language provides a special syntax for them. The empty list is written `[]`. Keep in mind that this is treated as a single constant symbol, although it does not obey the standard rules for constant-names. A non-empty list can be written in one of two ways. The first way of writing a list is this: the list consisting of the expressions `a`, `b` and `c` can be written as `[a,b,c]`. This way of writing a list makes it appear, in this case, as if it were a flat structure of some sort, that is, an expression tree with three subexpressions. The list `[a,b,c,d]` would seem to be an expression tree with four subexpressions, and so on. However, this is *not* the way Prolog actually looks at lists. To Prolog all lists are expressions trees with just *two* subexpressions, which we refer to as its *head* and its *tail*. Abstractly, we can think of lists as being any data type satisfying the following definition.

```
list( L ) :- L = [].
list( L ) :-
      head( L, _ ),
      tail( L, Tail ),
      list( Tail ).
```

The head can be anything we like. The tail must be itself a list. The second way of writing lists makes this clearer. The list with head `Head` and tail `Tail` can be written `[Head|Tail]`. This is the notation we use to actually write the "official" definition of this new data type.

```
list( L ) :- L = [].
list( L ) :-
      L = [Hd|Tl],
      list( Tl ).
```

Thus lists are recursive data structures, like the notation we defined for natural numbers in the last chapter. Like the natural numbers, there is no *a priori* bound on the size of a list. The recursive definition allows us to be vague about the length of a list: if the tail of a list is a variable, for example, then it can later be instantiated to another list of any size, or even to a list whose own tail is itself a variable.

Even the "bar-notation" `[H|T]` is different from the kinds of expression trees we are used to. In fact, although Prolog always prints out lists in this more readable notation, *internally* it treats lists just as it treats all other expression trees. Essentially, Prolog uses a functor '.'/2 in place of the bar-notation. So we have the following three equivalent ways of describing the same data object.

`[a,b,c]`     `[a|[b|[c|[]]]]`     '.'(a, '.'(b, '.'(c, [])))

Note that the bar-notation and the comma-notation can be mixed in the same term:

`[a,b,c] = [a|[b,c]] = [a,b|[c]] = [a,b,c|[]]`

As usual, '=' means 'unifies'. So apparently we have here the exceptional case where several objects which are syntactically distinct actually unify. In fact this is no exception. The rules for unification are defined on Prolog's internal representations (i.e., the dot-notation). All of these expressions are different "surface" ways of writing the same "underlying" expression tree.[1] These expressions are strictly cosmetic; they are for *you*, not for Prolog.

In general, the bar notation is only used when the tail of a list is a variable. Note also that `[a,B]` and `[a|B]` are very different objects. The list `[a,B]` is a list of length 2 whose second element happens to be a variable. The list `[a|B]` is a list of indeterminate length. The tail of `[a,B]` is `[B]`. The tail of `[a|B]` is just `B`. Consider, for example, the following short Prolog session.

```
?- [a,B] = [a|C].

C = [B]
yes
```

(Question: why didn't I write "`?- [a,B] = [a|B].`" ?)

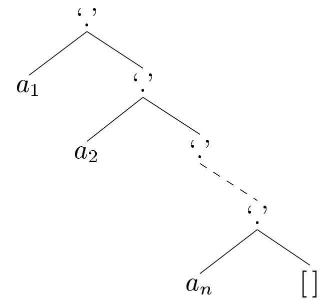
Note further that the bar-notation allows us to write terms which are not lists. The term `[a|b]` is a way of writing a perfectly well-formed expression tree, which could also be written `'.'(a,b)`. However, this expression is not a list. In particular it is *not* the list `[a,b]`, which corresponds to the expression `'.'(a,'.'(b,[]))`, alias `[a|[b]]`. As you can see, the two dot-expressions do not unify.

We noted at the beginning of this chapter that the data structure *list* was a special sort of tree: binary, and uniformly right-branching. So, for example, the list `[a1, ... ,aN]` would be represented internally as something like the expression tree in figure 5

Now consider the proof tree for the goal `?- list([a1, ... ,aN]).`, given in figure 7, based on the code for `list/1` given in figure 6. The structure

[1] In fact it seems wrong to refer to them as "expressions". We probably should reserve the name "expression" for things which obey the rules we set out earlier. The above objects are then something like different "ways of writing an expression".

**Figure 5** The expression tree representation of the list $[a_1, \ldots, a_n]$.

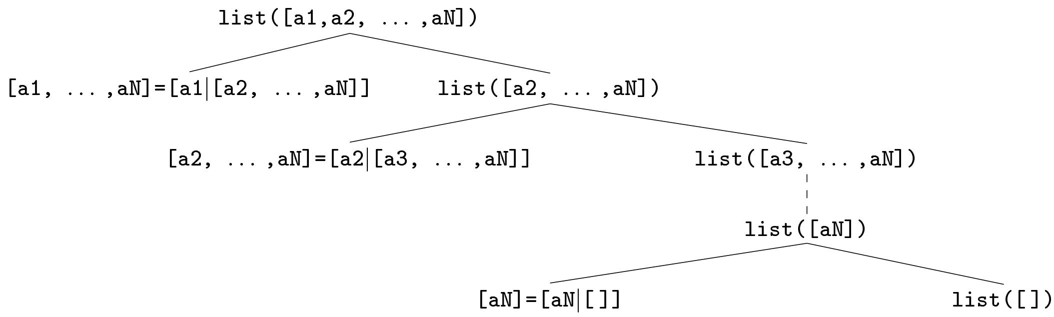



of the proof tree is almost point for point identical to the structure of the expression tree. This should come as no surprise, really. In a sense, the purpose of `list/1` is to prove that its input is a list by mapping it piece by piece to a list-structured expression tree. Put another way, the proof of `?- list([a1, ... ,aN]).` represents a *search* of the input expression tree to verify that every point in that input tree corresponds to a point in a list-shaped expression tree. And, very generally, a search through a particular data element is very likely to reflect the structure of that data element.

**Figure 6** Code for the data type predicate `list/1`.

```
list( [] ).
list( L ) :-
     L = [LHead|LTail],
     list( LTail ).
```

**Figure 7** A proof tree for the goal `?- list([a1, ... ,aN]).`



## 5.2 Comparing lists and natural numbers

The non-standard aspects of Prolog's list notation may make these structures seem very unfamiliar, but—notation aside—there is a great deal about lists which should seem very familiar from our work with natural numbers. Consider for instance the parallels in the predicates defining the data structures. (See table 6.) Essentially, the empty list corresponds to zero, and the operation of adding a new element on the front of the list corresponds to the operation of applying another instance of the successor function.

**Table 6** Definitions of `list/1` and `nn/1`.

```
list( L ) :-              nn( X ) :-
      L = [].                   X = 0.
list( L ) :-              nn( X ) :-
      L = [H|T],                X = s(Y),
      list( T ).                nn( Y ).
```

Another example will make the relation even clearer: in table 7 we present the predicate `len/2` to compute the length of a list, in successor-notation. Observe that the base clause of `len/2` combines the base cases of both `nn/1` and `list/1`. Similarly, the recursive clause of `len/2` combines the recursive clauses of `nn/1` and `list/1`. In effect we have taken the two very simple data-type definitions and combined them into a quite useful little predicate. (The predicate `length/2` is in fact a built-in predicate in Prolog; it returns the length of a list in standard arabic numerals. Note that our predicate `len/2` is *generative*. This is not true of the Prolog built-in predicate `length/2`.)

**Table 7** Definition of `len/2`.

```
len(L,N):-
      L = [],
      N = 0.
len(L,N):-
      L = [H|T],
      N = s(M),
      len(T,M).
```

If these two data structures are really so similar, then we would expect similar operations to be defined similarly. Take for example addition. The analogous operation on lists is list *concatenation*, the operation that takes a list $[a_1, \ldots, a_m]$ and a list $[b_1, \ldots, b_n]$ and maps them to a list $[a_1, \ldots, a_m, b_1, \ldots, b_n]$. (Note that the operation of concatenation yields a list whose length is the sum of the lengths of its operands—this is an essential clue that concatenation really is the generalization of addition to sequences.) The predicate `append/3` whose definition is given in table 8 does just this.

**Table 8** Definition of `append/3`

```
append(L1,L2,L3):-
     L1 = [],
     L2 = L3.
append(L1,L2,L3):-
     L1 = [Hd1|Tl1],
     L3 = [Hd3|Tl3],
     Hd1 = Hd3,
     append(Tl1,L2,Tl3).
```

As we might expect by now, the base clause of `append/3` is essentially the same as that of `plus/3`, except of course that we have substituted the empty list [] for 0. Furthermore, the recursive clause is largely identical to the recursive clause for `plus/3`. Borrowing the $\oplus$ symbol for a moment, we might express the logic of the recursive clause in functional or operational terms as follows.

$$[X|Y] \oplus Z = [X|(Y \oplus Z)]$$

Though we have invented some notation here without rigorously explaining it, the relation of concatenation to addition should be intuitively clear.

Now consider the definition of `shorter_than/2`, which is true of a pair of lists just in case the first is shorter than the second.

**Table 9** Definition of `shorter_than/2`.

```
shorter_than(X,Y):-
     X = [],
     Y = [_|_].
shorter_than(X,Y):-
     X = [_|XTail],
     Y = [_|YTail],
     shorter_than(XTail,YTail).
```

The code for `shorter_than/2` should be compared to the code for `lt/2`. In the base clause we use `[_|_]` in place of `s(_)`; it is a term unifying with all and only the nonempty lists, that is, in a sense, the lists which are strictly greater ("lengthier") than the empty list. The recursive clauses are also similar. We defined `lt/2` based on the fact that if $X < Y$, then $1 + X < 1 + Y$. Here we define `shorter_than/2` based on the fact that if $X \prec Y$ then $Z_1.X \prec Z_2.Y$ (temporarily making use of "$\prec$" to represent the shorter-than relation, and writing the '.'-functor infixed). In words, if $X$ is shorter than $Y$, then the result of sticking some $Z_1$ on the front of $X$ is sure to be shorter than the result of sticking a $Z_2$ on the front of $Y$. (For example, `[a]` is clearly shorter than `[1,2,3]`; so it stands to reason that `[X|[a]]` should be shorter than `[Y|[1,2,3]]`, and indeed, writing these more normally, it is obvious that `[X,a]` is shorter than `[Y,1,2,3]`.) Note that we have defined this relation on the number of elements, and *not* on the number of symbols required to type the list. In the current example, the first list remains shorter than the second even if we instantiate `X` to `[alpha,beta,gamma]`, and we instantiate `Y` to `alpha`.

The moral of this story is that, once we know that two structures are similar in some way, then we immediately inherit many of the procedures we used for the old structure for use with the new structure. Because we are constructing our programs on a firm logical foundation, we do not have to concern ourselves much with the internal workings of `append/3`, for example. Once we know that concatenation of lists is logically analogous to addition of natural numbers, then we simply make the relevant translations and we are more or less assured that the new predicate will work as we intend it to.

However, there are differences between the list predicates we have defined so far and the natural numbers predicates from the last chapter, and the differences are as instructive as the similarities. Consider now the definition of `prefix/2`, which we define so that `prefix(X,Y)` is true of two lists `X` and `Y` whenever `X` is a (proper) prefix of `Y`, that is, if `X` is of length $n$, then the initial $n$ members of `Y` define a list identical to `X`. The code is given in table 10. It can readily seen to be a straightforward extension of `shorter_than/2`. In fact it defines a relation which is a subset of the shorter-than relation. For example, `[a,b]` is a prefix of `[a,b,c,d]`, and clearly also shorter, but `[X,a]`, though shorter than `[X,1,2,3]`, is not a prefix of the latter.

**Table 10** Definition of `prefix/2`, for proper prefixes.

```
prefix(X,Y):-
      X = [],
      Y = [_|_].
prefix(X,Y):-
      X = [XHead|XTail],
      Y = [YHead|YTail],
      XHead = YHead,
      prefix(XTail,YTail).
```

Given the parallelism between forming `s(X)` from a natural number `X` and forming `[H|X]` from a list `X`, we should not be surprised that there is a line-for-line correspondence between `shorter_than/2` and `lt/2`. Note, however, that the definition of `prefix/2` requires an extra line, asserting that `XHead = YHead`. There is more to a list than just the number of members that it has; we care also about the identity of those members. Put another way, there is really only one way to form a larger natural number from a smaller natural number `X`: add another application of the s-function. But there are as many ways of forming new lists from a list X as there are expression trees—any one of them could serve as the head of the new list [H|X]! Essentially, prefix/2 has more constraints on it than shorter_than/2, which really only cares about the length of a list. Note also that the latter predicate is much less useful in practice than prefix/2.

Another way to look at the same distinction is via several slight modifications of the basic data type definition, given in table 11. The two predicates list_of_nn/1 and list_of_lists/1 are satisfied if their argument is a list all of whose members are of a particular type. Once again, they emphasize the same point we tried to make with prefix/2: unlike natural numbers, *lists have members.*

**Table 11** Definitions of list_of_Type/1 predicates.

```
list_of_nn(L):-
      L = [].
list_of_nn(L):-
      L = [H|T],
      nn(H),
      list_of_nn(T).

list_of_lists(L) :-
      L = [].
list_of_lists(L) :-
      L = [H|T],
      list(H),
      list_of_lists(T).
```

This (rather simple-minded) observation leads us to our next list-processing predicate, presented in table 12. We intend that member(E,L) be true of an expression E and list L whenever E occurs in L.

**Table 12** Definition of member/2.

```
member(E,L):-
      L = [E|_].
member(E,L):-
      L = [_|LTail],
      member(E,LTail).
```

The following innovation appears in the definition of member/2. The base clause is no longer the case where a particular argument unifies with the empty list []. Instead, the recursion ends if the head of the list argument (the second argument) is unifiable with the first argument. This difference is characteristic of predicates whose primary task is *searching*: processing ends when (or rather, *if*) the desired term is found. In more logical terms our two sorts of recursion contrast a certain *universal* flavor with an essentially *existential* flavor. For example in our data type definitions like `nn/1` or `list/1` we must examine every element of the data structure to verify that it is legitimate. But in search procedures like `member/2` we are more interested in knowing whether or not there exists a term of a certain sort in a particular collection of data. We do not need to consider the whole collection of data to verify this claim. Also note that just as the negation of an existential sentence has a universal flavor, and vice versa, so too if `E` does *not* appear in `L`, then the entire list will be searched before Prolog finally fails, and if a certain term is *not* a list (e.g., `[a|b]`), then there is no need to examine the entire term before failing.

### 5.2.1 Recursion and induction

We can actually formalize this concept of membership in a way that will be useful later on. So far we have looked at recursive definitions in a rather "top-down" manner. It is instructive to consider them from a more "bottom-up" perspective, as follows. A recursive definition of a set is a way of defining a set by starting with some basic elements, specifying some functions that create new elements out of old ones (the *generating functions*), and taking our recursively defined set to be the closure of the basic elements under the generating functions. Suppose, then, that we wish to define the two sets *NN* and *Lists*. We give the elements of their recursive definitions below.

- Definition of *NN*.
  1. $Basic(NN) = \{0\}$
  2. $Gen(NN) = \{\mathcal{F}_s\}$, where $\mathcal{F}_s(t) = s(t)$ for $t \in NN$.
- Definition of *Lists*.
  1. $Basic(Lists) = \{[]\}$
  2. $Gen(Lists) = \{\mathcal{F}_{exp}\}$, where $\mathcal{F}_{t_1}(t_2) = [t_1|t_2]$, for $t_1$ an expression, $t_2 \in Lists$.

Observe that the set of generating functions contains only a single element in the definition of *NN*, but contains an infinite number of elements in the definition of *Lists*. On the other hand, the generating functions in both definitions are *monadic*, that is, they take only a single argument. The recursive structure of these elements is in both cases *linear*. We will see when we cover trees in the next section how we can generalize both natural numbers and (hence) lists so that they are multiply recursive, simply by defining generating functions which are *polyadic*, i.e., which take more than one argument. In any case, we have here a very compact account of both the similarities and differences between natural numbers and lists: both are based on monadic functions, but one is based on a single monadic function while the other is based on multiple monadic functions.

The notion that lists "have members" corresponds to the fact that it makes sense to ask, for lists, which generating functions were used to construct them. For natural numbers such a question makes no sense: it is a question with only one possible answer.

There is a form of inductive proof which generalizes mathematical induction to this setting, called *structural induction*. We will not be so interested in inductive proofs in this course (but they are important for proving the correctness of an implementation!); we examine them here (briefly!) chiefly for the light they shed on constructing recursive predicates, which is our main topic. The principle of structural induction says essentially this: we can prove that a property $P$ holds of all the members of an inductively defined set $S$ by doing the following.

1. Take all the elements of $Basic(S)$, and prove that $P$ holds of each of them.

2. Take all of the generating functions $\mathcal{F}_{f_i}/n$ (where $n$ as usual gives us the arity of the function) and prove that, whenever $t_1, \ldots, t_n$ have property $P$, then $\mathcal{F}_{f_i}(t_1, \ldots, t_n)$ has property $P$.

This approach generalizes to the case where $Basic(S)$ or $Gen(S)$ have more than one member. The idea is to show that the basic elements have a property P and that P is preserved under all the ways of building larger elements.

We have actually already used the principle of structural induction covertly. In the section on Natural Numbers we claimed to be using the principle of mathematical induction (prove $P(0)$ and then prove $P(n) \Rightarrow P(n+1)$); in fact, strictly speaking, we were using the principle of structural induction (prove $P(0)$ and then prove $P(n) \Rightarrow P(s(n))$, i.e., $P(\mathcal{F}_s(n))$). Since $s(n)$ was intended to mean exactly the same thing as $n+1$ the deceit was harmless, but now that we know the principle of structural induction, we can be more precise about the techniques we have employed.

The relation between recursive definitions of sets and recursively-defined procedures is fairly easy to state at this point. First note that we can associate with any set $S$ a *characteristic predicate* $p_S$. We define $p_S(x)$ to be true if and only if (iff) $x \in S$. Clearly, one way to prove that $x \in S$ is first to identify which generating function $\mathcal{F}_f/n$ created $x$, and which $n$ expressions $t_1, \ldots, t_n$ $\mathcal{F}_f/n$ used to create $x$. Then we prove that each of the $t_i$ are in $S$, in the same way.[2]

Clearly the definitions of the data-type predicates `nn/1`, `list/1`, `list_of_nn/1` and `list_of_lists/1` are of this sort. The important thing to recognize is that pretty much *all* procedures manipulating lists or natural numbers in successor notation, for example, follow this pattern. That is, corresponding to the notion of a proof by structural induction we have the important notion of *recursion on a structure*. For example, we might say that `append/3` is implemented using *recursion on a list*, in particular, recursion on the list that is its first argument.

[2]If this reminds you of the relation between a grammar and a recognizer, it should! The rules of a grammar can be seen as a set of generating functions, with the elements of the lexicon as basic elements. Capitalizing on this similarity we can greatly extend the power of our grammatical formalisms, by incorporating a wider array of generating functions, while still being able to use the computational techniques developed for simple context-free grammars.

This means that the base case will typically be the base case for the basic data type definition. For example, if a predicate is constructed as a recursion on a list, then its base case will typically be the case where the relevant argument unifies with the empty list. Similarly the recursive step(s) will typically involve some processing of the head element, and recursion on the tail. As noted, this is the case with `append/3`: the recursion is on the first argument; the base case deals with how to do concatenation when the first argument is [ ]; the recursive step deals with how to do concatenation involving a non-empty first argument in terms of its tail.

We have already noted that the base case for `member/2` is *not* this "typical" base case, i.e., though `member/2` is implemented as a recursion on a list, its base clause does not represent the case where the list has become empty. So let us examine in some detail how a relation like `member/2` fits into this scheme of definition. In the process we may get a better understanding of how this style of definition and proof can apply to cases more complex than just the data type definitions.

For the moment let us think of `member/2` as defining a set, namely the set of all lists which have the first argument (call it `X`) as a member. We can imagine generating such a set in the following way. We do not want to begin the construction with the empty list, because the empty list cannot contain the element we are looking for. We want to choose as our basic elements a set of lists which are in some clear sense the "smallest", most elementary possible lists which are guaranteed to possess the property of interest (namely, the property of containing `X`). We have chosen as our basic elements the set of all lists which have `X` as their head. This set is infinite, of course, so we might expect that `member/2` should have an infinite number of base cases as a result, but in fact this is not true. We appeal here to the magic of Unification, and note that the single term `[X|_]` represents in a single expression every one of the lists which have `X` as their head. That is, every list which has `X` as its head can be represented as `[X|_]`$\theta$, for some substitution $\theta$.

It should be obvious that the generating functions for lists preserve the property of containing `X`. That is, if $t$ is a list containing `X`, then $\mathcal{F}_{exp}(t)$ will also be a list containing `X`, no matter what our choice of *exp* happens to be. So if we select as the generating functions for the set of "`X`-containing lists" the generating functions for *Lists*, then we have an almost immediate inductive proof of the correctness of `member/2`: the set of basic elements has the relevant property, and the property is preserved by the generators. QED.

In fact, this account of `member/2` is still insufficient. Technically, `member/2` does not define a set of lists, but rather a set of *pairs* of expressions, the first element of which appears as a member of the second element. Nonetheless our reasoning is essentially sound. We need only make the following modifications. First, our basic elements must be *pairs* of expressions, and we are interested in just those pairs of expressions which are such that the first element is the head of the second element. Once again, we can appeal to the magic of Unification: in spite of the fact that there are an infinite number of expressions, each of which can be the head of an infinite number of lists, every single one of these pairs unifies with the pair `(X,[X|_])`, since `X` is a variable. So indeed our base clause, which we can compactly represent as the unit clause

```
member( X, [X|_] ).
```

does indeed define the set of basic elements that we want.

For the recursive step, we need a set of generating functions which map element-list pairs to element-list pairs, instead of just mapping lists to lists. But this is still simple to do, because we want to map any particular pair $(t_1, t_2)$ to the pair $(t_1, \mathcal{F}_{exp}(t_2))$, that is, we do not want our generating functions to modify the expression we are looking for in any way. So we can define the generating functions for member-pairs based on the generating functions for lists: for every list generating function $\mathcal{F}_{exp}$, there is a corresponding member-pair generating function $\mathcal{F}^2_{exp}$ which is defined as follows.

$$\mathcal{F}^2_{exp} : (t_1, t_2) \mapsto (t_1, \mathcal{F}_{exp}(t_2))$$

So, if `(X,L)` is a member-pair, that is, if `member(X,L)` is true, then, given an arbitrary exression '_', we can construct a new member-pair `(X,[_|L])`, that is, `member(X,[_|L])` must also be true. And this is *exactly* how we have implemented the recursive clause of `member/2`.[3]

## 5.3 More members of the `member` family

Another important list-processing predicate, which involves searching, and which is therefore essentially a generalization of `member/2`, is `select/3`. The goal `select(Item,List,Remainder)` succeeds just in case (a) `Item` is a member of `List`, and (b) `Remainder` is a list consisting of all the members of `List` except for the occurrence of `Item` that we found.

```
select( Item, List, Remainder ) :-
      List = [Item|Remainder].
select( Item, List, Remainder ) :-
      List = [ListHead|ListTail],
      Remainder = [ListHead|RemainderTail],
      select( Item, ListTail, RemainderTail ).
```

[3]There is one further detail to consider: we have assumed so far that in all of our pairs of expressions, the second one was a list. In fact, our code does nothing to enforce this. For example, the goal

```
?- member( a, [a|b] ).
```

will succeed (though `member(b,[a|b])` will not). Our formal definitions reflect this; in particular, our definition of the basic elements says they just consist of all of the pairs of expressions such that the first element is the head of the second element, that is, such that the second element is the result of 'cons'-ing the first element onto some other expression. It is easy to modify the code for `member/2` so as to enforce that the second argument is a list, but this is not in general worth the extra computational effort, and we almost always will just use this "relaxed" or "approximate" definition in practice.

Note that the goal "`?- select( X, L, _ ).`" behaves exactly like a call to "`?- member( X, L ).`" The only real difference is that we maintain a record of all the items we found that were not the selected instance of `Item`. In `member/2` we just threw them away.

The next example generalizes `member/2` in a different dimension. The predicate which we will name `subset/2` assures that every member of one list is also a member of the other.

```
subset( L1, L2 ) :-
      L1 = [],
      list( L2 ).
subset( L1, L2 ) :-
      L1 = [L1Hd|L1Tl],
      member( L1Hd, L2 ),
      subset( L1Tl, L2 ).
```

Note that "subset" is somewhat of a misnomer here. If `L1` and `L2` do in fact represent sets, then `subset/2` does implement the subset relation. However, it also functions if `L1` and/or `L2` are not sets, i.e., when some expression occurs more than once in one or both of the lists.

There is another sense in which lists are not sets: they are ordered. Two sets should always be considered equal if they contain the same members, but two lists will only be equal if they contain the same members *in the same order*. It is sometimes useful to have a predicate which implements a notion of "sublist" which preserves order. That is, `L1` is a sublist of `L2` if all of the elements of `L1` occur in `L2` *in the same order* in which they occur in `L1`. So if `L1` were `[a,b]`, then `L2` could be `[a,37,2,b,4]`, for example.

```
sublist( L1, L2 ) :-
      L1 = [],
      list( L2 ).
sublist( L1, L2 ) :-
      L1 = [L1Hd|L1Tl],
      L2 = [L2Hd|L2Tl],
      L1Hd = L2Hd,
      sublist( L1Tl, L2Tl ).
sublist( L1, L2 ) :-
      L2 = [_|L2Tl],
      sublist( L1, L2Tl ).
```

This predicate is interesting in that it has two recursive clauses. The first one handles the case where we have found an instance of the head of `L1` in `L2`; the second clause handles the remaining case where we keep looking for an instance of the head of `L1` in `L2`.

Finally, here is another generalization of `member/2` which focuses on the orderedness property of lists. The predicate `earlier/3` is true of a pair of elements and a list just in case (an instance of) the first member of the pair appears earlier in the list than (an instance of) the second member of the pair.

```
earlier( First, Second, List ) :-
      List = [First|ListTail],
      member( Second, ListTail ).
earlier( First, Second, List ) :-
      List = [_|ListTail],
      earlier( First, Second, ListTail ).
```

Observe how this gives us a second notion of "less than" related to lists. First we had `prefix/2`, which was a "less than" relation on pairs of lists. Now we have also `earlier/2`, which is a "less than" relation on list elements, relative to a particular list.

## 5.4 "Modes" of Use

In the previous section we defined `append/3` as an operation that would concatenate two input lists and output the result. We can think of this usage as the *standard mode of operation* or just *standard mode*, for short. However, the advantage of using relations (i.e., predicates) rather than functions is that they are non-directional. Often there are other modes of operation besides the standard one that are useful. Consider for example the following goal.

```
?- append( _Left, [c|_Right], List ).
```

A little thought should convince you that this goal will succeed in exactly the same cases where

```
?- member( c, List ).
```

would succeed. That is, they are logically equivalent. In fact, we could have defined `member/2` as follows.

```
member( X, L ) :- append( _, [X|_], L ).
```

In this example, we are assuming that `append/3` is provided with its *third* argument as input, rather than its first (and for that matter most of its second) argument. In fact, we can use `append/3` in this way to match any pattern that we can express as a prefix of a list.

```
?- append( _Left, [c,_,e|_Right], [a,b,c,d,e] ).
```

So this multi-modal quality of predicates can lead to extremely economical use of code!

Consider another predicate which is in a sense built on this pattern-matching capability of `append/3`.

```
last( Element, List ) :-
      append( _, [Element], List ).
```

It should be clear on inspection that this predicate is true only when `Element` is the last element of `List`. Here the "pattern" we are matching is essentially structural, rather than depending on the content of the input list, but the principle of using `append/3` in this nonstandard mode as an engine for searching and decomposing lists remains the same.

As a further example, consider how we might redefine `earlier/3` in terms of `append/3`.

```
earlier( A, B, List ) :-
      append( _, [A|AfterA], List ),
      append( _, [B|_], AfterA ).
```

Note that the second call to `append/3` is exactly the same goal that would be generated by a call to

```
?- member( B, AfterA ).
```

The first call to `append/3` implements a relation which is sort of halfway between `member/2` and `select/3`: we throw away the elements of `List` which are to the left of `A`—just like in `member/2`—but we keep the elements to the right of `A`—just like in `select/3`.

Note that the inverse of a function is not guaranteed to be a function. In our terms, if the standard mode of operation of a given predicate is intended to implement a function—with only one solution—there is no guarantee that another mode will still have only one solution. For example there is only one way to concatenate the lists `[a,b]` and `[c,d]`, but the following goal has a number of solutions.

```
?- append( L1, L2, [a,b,c,d] ).
```

Sometimes this nondeterminacy can be put to good use. We have seen how to generate the members of a recursively defined set: simply call "`?- nn(X).`" or "`?- list(L).`" for example. We can also very easily construct predicates to generate the elements of arbitrary finite (and not too big!) sets, that is, sets for which generating functions may not exist.

```
fruit( X ) :-
      member( X, [apples, oranges, cherries, bananas] ).
```

A call to "`?- fruit(F).`" will then generate one after another all the fruits that the computer knows about.[4] We can use other members of the `member/2`-family in the same way: `select/3` is especially useful in this regard, but `subset/2` can also be used to generate the subsets of a given set, which is sometimes necessary.

[4]For the fun of it, you might try the following goal:

```
?- fruit(X), write(X), fail.
```

## 5.5 Finite State Automata

Now that we have lists, we can represent strings of symbols, e.g., words and sentences. We can define a *formal language* extensionally as a set of strings. Quite possibly the set will be infinite, but we require that all the strings that make it up be of finite length. And of course we require that our description of this infinite set be merely finite. This requirement, that we be able to describe infinite sets (and infinite objects in general) with finite means is fundamental to both formal linguistics and the theory of computation, and *Formal Language Theory* has for several decades marked the area where these two sciences meet and interact.

We have seen one way of constructing formal languages, when we first began to construct the language underlying Prolog, and again when we looked at how recursively defined sets could be built up out of basic elements and generating functions. We now look at a slightly different way of building up sets, particularly suited to building up sets of strings.

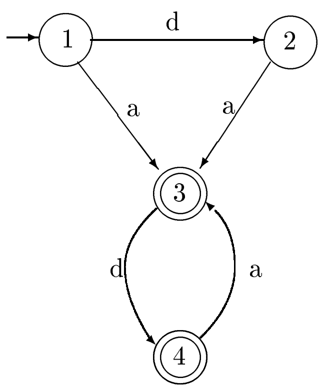
Informally, a *finite state automaton* (FSA, for short) is an abstract mathematical description of a machine whose function is to *recognize* the strings of a particular formal language. A string (sometimes called a "word") is considered to be in the language defined by the automaton if and only if it is recognized by the automaton. We assume that the symbols of the input string are written on a *tape*, which must be finite in length, but can be arbitrarily long. We divide the tape up into *cells*: each cell contains a single symbol from the input string.

An FSA has a particularly primitive sort of memory. It is built so that at any point in time it is in one of a finite number of *states* (hence its name). Its "memory" is limited to the number of distinct states it can be in. Put another way, an FSA cannot distinguish between all the different ways there might be to get to a particular state. Roughly, all computations that put it in state $q_1$ are the same as far as the FSA is concerned, but they are all different from all of the computations which would put it in some other state. All it can do is to read the current tape-cell, examine itself to see what state it's in, and then (a) move to the next tape-cell and (b) simultaneously change to a new state. What its new state will be is determined by a *transition relation*, which is really the heart and soul of the language-definition that the FSA encodes.

An FSA defines a language in the following way. One of its states is distinguished as the *start state*. Then a subset of states (possibly including the start state) are designated as possible *final states*. We start the machine in its designated start state, looking at the first cell of the input tape. If there is a sequence of transitions from state to state such that, after scanning the tape, the FSA is in one of its designated final states, then the string written on the tape is in the language recognized by the FSA. Otherwise it is not. (There are actually two ways for a string to be ruled out: the FSA can end up in a non-final state, or it can at some point in the middle of the string find itself in a state from which there is no possible transition, given the current symbol.)

FSAs are usually depicted as in figure 8. Circles represent states; they are numbered so that we can refer to particular states easily. Double circles

**Figure 8** A finite state automaton for recognizing a very childish language.




represent final states. Arcs represent possible transitions, and labels on the arcs are taken from the underlying alphabet or lexicon of the language. So, for example, if our machine of figure 8 is in state 3 and reads a ‘d’, then it moves to state 4. The arc with no label and no state-of-origin indicates the designated start state (state 1, in this case).

We can see that, given the string ‘dada’, this automaton will start in state 1, scanning a ‘d’. The only possible transition is then to state 2; now the FSA is in state 2, scanning an ‘a’, so the only possible transition is to state 3, whence reading a ‘d’ takes the FSA to state 4, whence reading another ‘a’ takes it back to state 3. There is no more input, and the machine is in a final state, so ‘dada’ is in the language that it recognizes.

The string ‘d’ would “strand” the machine in state 2, a non-final state, and the string ‘dadd’ would strand the machine in state 4 reading a ‘d’, for which no transition is defined. These strings are therefore not in the language.

More formally, a finite state automaton $A$ is defined as below.

$$A = \langle \Sigma, Q, q_0, F, \delta \rangle$$

$\Sigma$ is the underlying alphabet ($\Sigma = \{d, a\}$ in this example), $Q = \{1, 2, 3, 4\}$ is the set of states, $q_0 = 1$ is the start state, $F = \{3, 4\}$ are the final states and $\delta$ is the transition relation.

$$\delta \subseteq Q \times \Sigma \times Q$$

We can also think of $\delta$ as a transition *function* which maps state-symbol pairs to sets of states.

$$\hat{\delta}(q, \sigma) = \{q' \mid \delta(q, \sigma, q')\}$$

Note that if the output of the $\hat{\delta}$ function is always a singleton set, that is, a set with only one member, then the automaton is *deterministic*. FSA’s do not need to be deterministic: from any given state, reading any given symbol, it is perfectly legal to have several transitions. Note, however, that the automaton of figure 8 is deterministic.

It is usual to define an extended transition relation which takes strings of symbols instead of individual symbols as its second argument. For $a \in \Sigma$, $w \in \Sigma^*$ and $\epsilon$ the empty string, we have the following definition of $\delta^*$.

$$\begin{aligned}
\delta^*(q_i, \epsilon, q_j) &\iff q_j = q_i \\
\delta^*(q_i, aw, q_j) &\iff (\exists q_k)[\delta(q_i, a, q_k) \wedge \delta^*(q_k, w, q_j)]
\end{aligned}$$

The relation $\delta^*$ is thus essentially the *reflexive, transitive closure* of the relation $\delta$. Accordingly, a string $w$ is in the language recognized by $A$ (denoted $L(A)$) just in case

$$\delta^*(q_0, w, q) \wedge q \in F. \tag{5.1}$$

From this formal definition it is an almost trivial matter to implement a finite state automaton in Prolog. First, we implement the transition relation using the predicate `delta/3`, essentially by just reading off the arcs in figure 8.

```
delta( 1, d, 2 ).
delta( 1, a, 3 ).
delta( 2, a, 3 ).
delta( 3, d, 4 ).
delta( 4, a, 3 ).
```

Then we define the start and end states appropriately.

```
start_state(1).

final_state(3).
final_state(4).
```

We will use lists to encode strings, so the empty string $\epsilon$ becomes the empty list `[]` and the string $aw$ is represented as a list `[A|W]`. So we can define `delta_star/3` accordingly.

```
delta_star( Q, [], Q ).
delta_star( Q0, [A|W], Q ) :-
      delta( Q0, A, Q1 ),
      delta_star( Q1, W, Q ).
```

Finally, given our compact definition of recognition in (5.1), we can implement `recognize/1` as follows.

```
recognize( String ) :-
      start_state( Q0 ),
      delta_star( Q0, String, Q ),
      final_state( Q ).
```

And that is that. Now for example we can type in goals like

```
?- recognize( [a,d,a,d,a] ).
```

and have Prolog tell us whether or not the given input string is in the language of our automaton or not. For example, the above goal should succeed, but the following goal should fail.

```
?- recognize( [a,d,a,d,d] ).
```

Note, before we continue, that if we wish to change the language defined by the automaton, we need only change the clauses defining `delta`/3, `start_state`/1 and `final_state`/1. The clauses defining `delta_star`/3 and `recognize`/1 come from the general definition of what it means to be a Finite State Automaton, and remain the same for *all* FSA's that we might want to implement.

### 5.5.1 On Enumerating Infinite Sets

As you might expect by now, we can use `recognize/3` not only to recognize strings input by the user, but also to *generate* all the strings of the language.

```
?- recognize( X ).

      X = [d,a] ;
      X = [d,a,d] ;
      X = [d,a,d,a] ;
```
*and so on until you lose patience.*

Well, this is almost true. In fact there is a technical sense in which Prolog will *not* enumerate all the strings of the language, and it has nothing to do with the infiniteness of the set involved. "Enumerate" means something special in mathematics. In particular it means that if we generate a sequence, even though the sequence may be infinite, any particular member of that sequence will be listed after a finite amount of time. This is clearly true of our definition of `nn/1`, for example. For any arbitrary natural number $n$ its successor-notation-name will be generated in approximately $n$ steps. Here however we are in a different situation. The "enumeration" order for strings in this automaton is

$$da, dad, dada, \dots, a, ad, ada, \dots$$

where each occurrence of " ... " stands for an infinitely long sequence. Clearly, then, the string $a$ will not in fact be listed after any finite amount of time.

Of course this language has only countably many members. That is, there is no sense in which this language is somehow twice as big as the set of natural numbers, for example, just because the given automaton, left to its own devices, will generate the language as two infinite sequences instead of one. The problem is not with the language but with the order in which we list its members. If we listed them as

$$a, da, ad, dad, ada, dada, adad, \dots \tag{5.2}$$

for example then there would be no problem. That is, we list all of the strings of length 1, then all the strings of length 2, and so on. In this case it should be clear that any string of the language will in fact be listed after some finite number of steps.

In fact we can accomplish this reordering without any deep modifications to Prolog or our program. We simply issue the goal

```
?- len( L, _ ), recognize( L ).
```

The calls to `len/2` will generate list templates (lists filled with anonymous variables) in order of length: first `[]`, then `[_]`, then `[_,_]`, and so forth. Then `recognize/1` will see if there are any strings in the language that match these templates. Keep in mind that Prolog, when backtracking to find more solutions, retries the most recent goal first, so only when it has run out of strings of length $k$ to recognize will it backtrack all the way to `len/2` and start work on strings of length $k+1$. This will in fact give us the enumeration order in (5.2), which *is* a proper enumeration.

Note that we have actually accomplished something a little deeper than just smoothing out the generating behavior of `recognize/1`. Recall that normally Prolog performs a depth first search of its search tree, and indeed it is doing so here as well. But the depth first search of the search tree for the goal

```
?- len( L, _ ), recognize( L ).
```

corresponds exactly to the *breadth-first* traversal of the search tree for the simpler goal

```
?- recognize( L ).
```

We have in effect tricked Prolog into running breadth-first instead of depth-first!

### 5.5.2 A linguistically deeper automaton

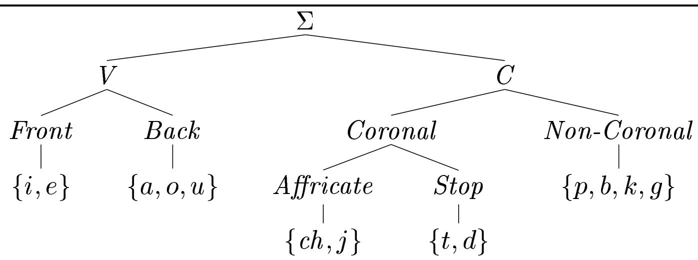
Suppose we wish to define a formal language which approximates a natural language with the following phonological property. In this language, coronal stops ($t$ and $d$) appear to be "palatalized" before front vowels ($i$ and $e$). That is, a $t$ might appear in a word like *atogu*, but instead of a word like **ategu*, we would see only *achegu*. Assume further that this language has only CV syllables, with the added possibility of words beginning with a vowel and ending in a consonant. Note that it follows from this description that words cannot end in affricates. So *chit* is a word but **chich* is not.

Let $\mathfrak{A}$ be a $\Sigma$-automaton, where $\Sigma$ is defined as in figure 9, defined by the following Prolog clauses.

```
% states( [1,2.1,2.2,3,4.1,4.2] ).

start_state( 1 ).
```

**Figure 9** The alphabet $\Sigma$




```
final_state( 3 ).
final_state( 4.2 ).
```

and the transition relation `delta/3` is defined as follows.

```
delta(1,X,2.1) :- affricate( X ).
delta(1,X,2.1) :- non_coronal( X ).
delta(1,X,2.2) :- stop( X ).
delta(1,X,2.2) :- non_coronal( X ).
delta(1,X,3) :- vowel( X ).
delta(2.1,X,3) :- front( X ).
delta(2.2,X,3) :- back( X ).
delta(3,X,4.1) :- affricate( X ).
delta(3,X,4.1) :- non_coronal( X ).
delta(3,X,4.2) :- stop( X ).
delta(3,X,4.2) :- non_coronal( X ).
delta(4.1,X,3) :- front( X ).
delta(4.2,X,3) :- back( X ).
```

The transition relation is based on the automaton for the "*dada*-language" which we examined previously. That automaton gives us our basic CV structure. We need to split states 2 and 4 to handle the different types of vowels and their effects on consonants, so we now have states 2.1, 2.2, 4.1 and 4.2. Note that state 4.2 is a final state, but state 4.1 is not: this prevents us from ending a word in an affricate consonant. Note also that this automaton is nondeterministic: in states 1 and 3, reading a non-coronal consonant, automaton $\mathfrak{A}$ can move in either of two directions, since a non-coronal consonant is free to occur before either type of vowel.

The auxiliary clauses which we have used to name subcategories of the alphabet $\Sigma$ are defined as follows.

```
vowel( X ) :- front( X ).
vowel( X ) :- back( X ).

front( i ).    front( e ).
```

```
back( a ).    back( o ).     back( u ).

consonant( X ) :- coronal( X ).
consonant( X ) :- non_coronal( X ).

coronal( X ) :- affricate( X ).
coronal( X ) :- stop( X ).

affricate( ch ).     affricate( j ).

stop( t ).     stop( d ).

non_coronal( p ).     non_coronal( b ).
non_coronal( k ).     non_coronal( g ).
```

*Exercise:* Japanese exhibits a similar, but slightly different and slightly more complex phonological process. There (simplifying tremendously) coronal stops are replaced by these palatal affricates ($t \longrightarrow ch,\ d \longrightarrow j$) before $i$, and they are replaced by *alveolar* affricates before $u$ ($t \longrightarrow ts,\ d \longrightarrow dz$, something like that anyway). How would you modify automaton $\mathfrak{A}$ so that it recognized/generated legal words of 'pseudo-Japanese'?

# Chapter 6

# Trees

## 6.1 Trees: Lists with Bushy Tails

If all of our data structures (and even our programs!) are ultimately expression trees, then we should expect to spend a lot of time manipulating trees. Indeed, inasmuch as natural numbers in successor notation and lists are expression trees, we have already spent some time doing just that. But lists and natural numbers are rather special cases of trees. These structures are of course *technically* trees, but they're really not very 'tree-like'. What does this mean? What does it mean to be "essentially tree-like?" Let's begin by comparing the technical recursive/inductive definitions of successor-notation and lists, and see how we can make the concepts we find there more general.

- Definition of $NN$:
  1. $Basic = \{0\}$,
  2. $Gen = \{\mathcal{F}_s\}$, where we define $\mathcal{F}_s$ as follows:

$$\mathcal{F}_s(t) = s(t).$$

- Definition of $Lists$:
  1. $Basic = \{[\,]\}$,
  2. $Gen = \{\mathcal{F}_{exp} \mid exp \text{ is an expression tree}\}$, where we define $\mathcal{F}_{exp}$ as follows:

$$\mathcal{F}_{exp}(t) = [exp|t].$$

We observed in the last section that these structures are similar in that their generator functions take only a single argument. To obtain trees, we simply relax this restriction and allow generator functions which take multiple arguments. So with successor notation and lists we have trees with an essentially *linear* structure, because the functions which generate them are recursive in only one argument position. Trees, whose generator functions are *polyadic*, not necessarily monadic, are *multiply recursive* and essentially nonlinear because their generator functions are recursive in several argument places. So we have essentially the following hierarchy of generalizations:

- The natural numbers form a mathematical structure with a single monadic function.
- The lists form a mathematical structure with *many* monadic functions.
- The trees form a mathematical structure with *many polyadic* functions.

For our purposes it is perhaps best to think of a tree as being like a 'sequence' with one head and multiple tails, each of which is (recursively) itself a 'sequence' with one head and multiple tails, each of which is ...

```
list( L ) :- empty( L ).
list( L ) :-
      head( L, _ ),
      tail( L, LTail ),
      list( LTail ).

tree( T ) :- leaf( T ).
tree( T ) :-
      root( T, _ ), % ''head''
      subtrees( T, SubTrees ), % ''tails''
      forest( SubTrees ).  % Is each tail a tree?

forest( LT ) :- empty_forest( LT ).
forest( LT ) :-
      first( LT, LT_First ),
      rest( LT, LT_Rest ),
      tree( LT_First ),
      forest( LT_Rest ).
```

Essentially, we treat the root of a tree as we would treat the head of a list: this is the main similarity between trees and lists. On the other hand, the multiply recursive nature of trees means that a whole new dimension is introduced into our treatment of a tree's tails. That is, the definition of the tree data type given here is *doubly recursive*. Intuitively, we must travel recursively *down* the tree and also *across* the sequence of subtrees at any given node.

### 6.1.1 How to write trees

In the case of lists, there was a special representation already built into Prolog. In the case of trees we are on our own. So the way of encoding trees which I will present to you here should in no way be considered the absolute and inviolable standard way to do things. There is much more variety among programs and programmers in representations for trees than there is in representations of sequences. We will use the following scheme to encode trees.

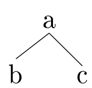
First, we define a forest to be a *list* of trees. So we can define `empty_forest/1` as the empty list, `[]`, and `first/2` and `rest/2` as the head and tail of a list, respectively. So the definition of `forest/1` can be rewritten as follows.

```
forest( []).
forest( [T1|RestTs] ) :-
      tree( T1 ),
      forest( RestTs ).
```

A *tree* is the pairing of a root with a (possibly empty) forest of subtrees. So we need some 2-place functor to tie these two data items together. Here we will use `'/'/2`, the division symbol. Because this can be written infixed between its arguments, it makes tree terms a little more readable. Once again: *there is nothing sacred in this representation*; in particular, one might prefer a prefixed functor like `t` or even `tree`. Using the scheme we have adopted here, though, we would implement `tree/1` as follows.

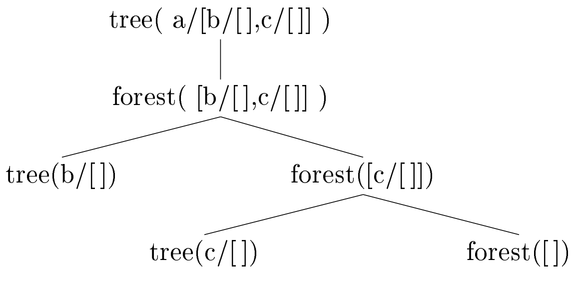
```
tree( _Root/[]).  % A leaf has no subtrees.
tree( _Root/SubTrees ) :-
      SubTrees = [_|_],  % SubTrees is non-empty;
      forest( SubTrees ).
```

For example, we would represent the tree




as the Prolog term `a/[ b/[], c/[]]`. The proof that this is a tree is straightforward and is summarized in figure 10.

**Figure 10** Proof tree for the goal: `?- tree(a/[b/[],c/[]]).`




The definition I have given in `tree/1` is very general. Often we can take advantage of particular circumstances to derive more restricted definitions. One of the most common subtypes of tree is the *binary tree*. A binary tree is a tree in which every node has exactly two or zero subtrees. We have already seen one example of this structure: lists are a sort of binary tree, except that they are only recursive in their second subtree. We can represent binary trees as 3-place functions, with one argument for the root, as usual, and the other two argument places for the left subtree and the right subtree.

```
bintree( bt( Root, [], []) ).
bintree( bt( Root, LTree, RTree ) ) :-
      bintree( LTree ),
      bintree( RTree ).
```

Clearly we can generlize this to trinary, quaternary and in general $n$-ary trees, for any fixed $n$, as follows.

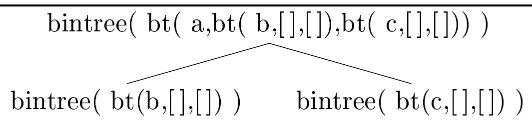
```
ntree( n( Root, [], ... , []) ).
ntree( n( Root, T1, ... , Tn ) ) :-
      ntree( T1 ),
      ... ,
      ntree( Tn ).
```

Our original example was a binary tree. Recoding it as a `bintree` would yield the following term.

```
bt( a, bt( b, [], []), bt( c, [], []) )
```

The representation of the data structure is less readable, perhaps (though no 'one-dimensional' representation of a tree will ever be *really* readable!), but we can process binary trees without ever calling anything like `forest/1`. That is, instead of one call to `tree/1` and one to `forest/1`, we have two calls to `bintree/1`. The proof tree corresponding to figure 10 is given in figure 11. Note that the proof tree in this case much more closely reflects the structure of the underlying tree. With our more flexible slash-notation we have to focus on the calls to `tree/1` and ignore the calls to `forest/1`, treating them as essentially just some sort of glue.

**Figure 11** Proof tree for the goal `?- bintree( bt( a,bt( b,[],[]),bt( c,[],[])) ).`




## 6.2 Climbing Around in a Tree

From time to time we will need to process a tree in a way which requires that we visit every node in the tree exactly once. This would be the case if, for example, the tree represented an arithmetic expression which we wished to evaluate, or a boolean expression which we wished to test for satisfiability. The simplest such application is the case where we just want to list the labels of all the nodes in a tree, or print them out. We will begin with binary trees, and then return to the general case.

There are essentially three ways to traverse a binary tree, each yielding a sequence of nodes with exactly the same members, but in a different order. These three traversal orders, *preorder inorder* and *postorder*, are described informally in table 13.

**Table 13** The three basic traversal orders, for binary trees.

| | | |
|---|---|---|
| preorder | — | process the root,<br>then the left subtree,<br>then the right subtree; |
| inorder | — | process the left subtree,<br>then the root,<br>then the right subtree; |
| postorder | — | process the left subtree,<br>then the right subtree,<br>then the root. |

Consider then the three implementations of `printtree/2` given in table 14. The first will print the list of nodes in preorder, the second in inorder and the third in postorder. (The Prolog builtin predicate `write/1` prints out a term on the computer screen. The term ' ' is just a space typed between quotes; it is a legal constant, since anything between quotes is a legal constant.) Note that everything is identical in all three versions except the order of the subgoals in the recursive clauses.

With general trees the situation is only slightly more complex. In place of two calls to `printtree` we need to call a second predicate `print_trees` which will print out sequences of trees of any length. Then the code for the recursive clause of `printtree_1` becomes

```
printtree_1( Label/Subtrees ) :-
      printlabel( Label ),
      print_trees_1( Subtrees ).
```

Then the definition for a postorder traversal just reverses the order of the goals in the body of `printtree_1`. The definition of an inorder traversal is more complicated. In the first place, given a tree with, say, four subtrees, we have the option of processing the first subtree, then the label, then the rest, or else the first then the second then the label then the rest, and so on. So for example we might use a clause like

```
printtree_2( Label/[LeftCorner|Rest] ) :-
      printtree_2( LeftCorner ),
```

**Table 14** Printing a tree, in preorder, inorder and postorder.

```
printtree_1( bt(Label,[],[]) ).
printtree_1( bt(Label,Left,Right) )
      :-
      printlabel( Label ),
      printtree_1( Left ),
      printtree_1( Right ).

printtree_2( bt(Label,[],[]) ).
printtree_2( bt(Label,Left,Right) )
      :-
      printtree_2( Left ),
      printlabel( Label ),
      printtree_2( Right ).

printtree_3( bt(Label,[],[]) ).
printtree_3( bt(Label,Left,Right) )
      :-
      printtree_3( Left ),
      printtree_3( Right ),
      printlabel( Label ).

printlabel( X ) :- write( X ),
      write( ' ' ).
```

**Figure 12** Pretty-printer output.

```
a
    b
        c
        d
    e
```

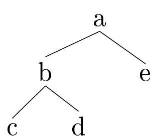
```
        printlabel( Label ),
        print_trees_2( Rest ).
```

The definitions of the various print_trees_$i$ differ only which version of `printtree` they call.

### 6.2.1 Pretty-printing Trees

A slightly more sophisticated—and extremely useful—example is the following. It "pretty-prints" a tree, that is, prints it in a format which reflects not only the content of the tree, but also its structure. With `printtree/1` all we got was the content.

`Pretty_print/1` prints out a tree as follows. We define a tab stop to be four spaces. Then each label will be printed as many tab stops from the left margin as it has ancestors in the tree. That is, the root node will be printed at the left margin (0 ancestors, 0 tab stops), its immediate subtrees will be printed 1 tab stop from the margin, and so on. For example, the tree `a/[b/[c/[],d/[]],e/[]]`, i.e.,




will be printed as in figure 12.

We begin by defining a "shell predicate" `pretty_print/1`, which simply initializes the tab stop to zero and calls the main predicate `pp_tree/2`. The code is given in table 15; it follows closely the data structure definition `tree/1`, as we would expect.

Several new predicates are introduced in this example which are built in to Prolog, which is why we did not present their definitions in table 15. First the predicate `is/2`, which is defined as an infix operator, and so is written between its two arguments, does arithmetic evaluation. It uses the computer's own native arithmetic abilities, and so is much more efficient than the code

**Table 15** Simple pretty printer for trees.

```
pretty_print( Tree ) :-
      Tab = 0,
      pp_tree( Tab, Tree ).

pp_tree( Tab, Label/[]) :- % Print a
      leaf.
      print_label( Tab, Label ).
pp_tree( Tab, Label/SubTrees ) :-
      SubTrees = [_|_],
      print_label( Tab, Label ),
      NewTab is Tab + 4,
      pp_subtrees( NewTab, SubTrees
      ).

pp_subtrees( _, []).
pp_subtrees( Tab, [Next|Rest] ) :-
      pp_tree( Tab, Next ),
      pp_subtrees( Tab, Rest ).

print_label( Tab, Label ) :-
      tab( Tab ),
      write( Label ),
      nl.
```

we defined in the section on Natural Numbers. However, this efficiency is only gained with some sacrifice. The "predicate" `is/2` is not truly relational: it requires absolutely that its first argument be a variable or a numeral and its second argument be a term which (a) represents an arithmetic expression and (b) *contains no uninstantiated variables.* Note that the first argument can be an instantiated variable, in which case a call to `is/2` simply checks that the number instantiating its first argument is indeed the value of the expression in its second argument. However, if any variable term in the second argument has not been instantiated (to a number or other fully instantiated arithmetic expression), then *Prolog will halt and report an error.* The predicate `is/2` is thus non-logical in its behavior, and therefore should be used with care or the logical interpretation of your program will disappear.

The remaining builtin predicates used in defining `pretty_print/1` handle the output of information directly to the terminal screen while the program is running. Logically, they have the same meaning as `true/0`, a constant which always succeeds. Procedurally, however, they have side effects, namely they cause the user's terminal to display the given information. The builtin `tab/1` simply prints the number of spaces specified in its single argument. The builtin `write/1` writes out the term given in its argument. Note that ' ' (a space enclosed in single quotes) is a perfectly legitimate term, as is ', ' (a comma followed by a space, all enclosed in single quotes): anything enclosed in single quotes counts as a constant term. So we can do a fair amount of formatting this way, though it is not necessary here. Finally, the builtin `nl/0` simply prints a carriage return ("nl" stands for "new line").

`Pretty_print/1` is an example of a preorder traversal of a tree.

## 6.3 Generalizations of some predicates on lists

A number of the things we can do with lists we will want to do also with trees. For example, we had `member/2`, which would search lists for us.

```
member( X, [X|_] ).
member( X, [_|L] ) :-
      member( X, L ).
```

The analogous predicate for trees we will call `tmember/2`. It is defined so that `tmember( X, Tree )` is true just in case `X` is the label on a node in `Tree`. There are actually two senses in which something can be a "member" of a tree: it can be a node in the tree, or it can be a subtree of the tree. It is the former we are most interested in here, but in fact the two problems are closely related, and we begin by defining `subtree/2`.

```
subtree( Tree, Tree ).
subtree( LittleT, BigT ) :-
      BigT = _/SubTrees,
      member( MiddleT, SubTrees ),
      subtree( LittleT, MiddleT ).
```

Since our representations of trees are in part constructed out of lists, it should be no surprise to see `member/2` appear in our predicates for searching trees. Here now is the code for `tmember/2`.

```
tmember( X, X/_ ).
tmember( X, BigT ) :-
      BigT = _/SubTrees,
      member( MiddleT, SubTrees ),
      tmember( X, MiddleT ).
```

As promised, it is extremely similar in form to `subtree/2`. After all, nodes are just the roots of subtrees and subtrees are just the set of nodes dominated by their roots.

I have argued that trees are rather like lists in two dimensions. We have seen that we can use our standard list predicates to good effect by applying them to the lists of subtrees at any particular node. We can mimic this "horizontal" use of list processing in the "vertical" dimension in a fairly direct manner once we provide ourselves with a definition of a *path* in a tree. The following predicate, which defines paths, can be used either to determine if a given list of expressions labels a branch from the root to the frontier of a tree or else to generate/enumerate the paths in a given tree.

```
path( [X], X/[]).
path( [X|Xs], X/STs ) :-
      member( ST, STs ),
      path( Xs, ST ).
```

So, for example, we could reimplement `tmember/2` as follows.

```
tmember( Node, Tree ) :-
      path( P, Tree ),
      member( Node, P ).
```

The following predicate gives the "yield" of a tree, that is, the list of leaf-labels, in order.

```
yield( Label/[], [Label] ).
yield( _/SubTrees, Yield ) :-
      SubTrees = [_|_], % Not a leaf.
      yield_star( SubTrees, Yield ).

yield_star( [], []).
yield_star( [Next|Rest], Yield ) :-
      yield( Next, Y1 ),
      yield_star( Rest, Y2 ),
      append( Y1, Y2, Yield ).
```

This predicate represents a postorder traversal of the tree. The yields of the subtrees are determined, and then the yield of the current tree is determined as a function of those yields (i.e., their concatenation).

The final member of this family of predicates finds "cuts" through a tree. A cut—sometimes called a "sentential form"—is a sequence of nodes (here identified with their labels) which taken all together dominate the entire string, and none of which dominate any of the others. Given a grammar and a rewriting system, the sentential forms are all those strings of terminals and nonterminals that one can get by rewriting the start symbol. One sees immediately that the yield of a tree is just a special case of a cut which contains the labels of only leaf nodes. Alternatively, a cut is just the yield of a truncated version of the tree. Not surprisingly, then, the code for these two predicates is almost identical, differing mainly in that `cut/2` is permitted to end its recursion before it reaches a leaf (and also in that the order of the arguments has been reversed).

```
cut( [X], X/_ ).
cut( C, _/SubTrees ) :-
      SubTrees = [_|_],
      cut_star( C, SubTrees ).

cut_star( [], []).
cut_star( C, [SubTree|Rest] ) :-
      cut( C1, SubTree ),
      cut_star( C2, Rest ),
      append( C1, C2, C ).
```

## 6.4 Tree Geometry

Trees, as we have defined them here, are largely made up out of the immediate dominance relation (symbolized by '/') and the immediate precedence relation (symbolized by '.', the list constructor). Often, however, we are interested in other relations that can hold among the nodes of a tree. One simple example is the remote dominance, or *ancestor* relation, which is the reflexive transitive closure of the immediate dominance (alias "parent") relation. That is, every node is an ancestor to itself, and one node is the ancestor of another if there is a chain of parent-links which connects them.

```
ancestor( A, B, Tree ) :-
      subtree( A/ASubs, Tree ),
      subtree( B/_, A/ASubs ).
```

Unfolding the code for `subtree/2`, we can see that `ancestor/3` could also be defined as follows.

```
ancestor( A, A, A/_ ).
```

```
ancestor( A, B, A/ASubs ) :-
      member( MiddleT, ASubs ),
      subtree( B/_, MiddleT ).
ancestor( A, B, _/SubTrees ) :-
      member( MiddleT, SubTrees ),
      ancestor( A, B, MiddleT ).
```

Note that this is a generalization of the list predicate `earlier/3` which can be defined in the same two ways.

```
earlier1( A, B, List ) :-
      suffix( [A|ARest], List ),
      suffix( [B|_], ARest.

earlier2( A, A, [A|_] ).
earlier2( A, B, [A|ARest] ) :-
      member( B, ARest ).
earlier2( A, B, [_|Rest] ) :-
      earlier2( A, B, Rest ).
```

We can also re-re-implement `ancestor/3` in terms of `path/2`.

```
ancestor( Hi, Lo, Tree ) :-
      path( P, Tree ),
      earlier( Hi, Lo, P ).
```

The `ancestor/3` relation is a *partial order*. That is, it is a relation which is (a) reflexive (A is an ancestor of A), (b) transitive (if A is an ancestor of B and B is an ancestor of C, then A is an ancestor of C), and (c) antisymmetric (if A is an ancestor of B and B is an ancestor of A, then A equals B). The first two conditions are built into the definition in a transparent manner. Antisymmetry follows, but the demonstration is a little more devious, and is clearest if we consider the first definition of `ancestor/3`.

If we can extend the immediate dominance relation to a partial order, then we ought to be able to extend the immediate precedence relation to a partial order as well. However, this is not quite as straightforward as the definition of `ancestor/3`. Essentially, A precedes B if we can climb up the tree from A and climb up the tree from B and eventually reach a point at which some ancestor of A and some ancestor of B are both daughters of the same node. Then, if A's ancestor is earlier than B's ancestor, A precedes B.

```
precedes( A, B, Tree ) :-
      Tree = _/SubTrees,
      earlier( ATree, BTree, SubTrees ),
      tmember( A, ATree ),
      tmember( B, BTree ).
```

We can also implement `precedes/3` using `cut/2`, in the same way we used `path/2` to reimplement `ancestor/3`:

```
precedes( A, B, Tree ) :-
      cut( C, Tree ),
      earlier( A, B, C ).
```

# Chapter 7

# Graphs

## 7.1 Practically everything is a graph

The last data structures we will consider before moving on to more process-oriented topics are *graphs*. Graphs are, formally, collections of objects, called *nodes*, together with a collection of line segments which connect pairs of nodes, called either *arcs* or *edges*. Graphs can be either *directed* (in which case we use arcs to connect the nodes) or *undirected* (in which case we use edges). We have already seen several examples of directed graphs: trees are a special case of directed graph, and also the transition diagrams for finite state automata and Turing machines are directed graphs. Both the nodes and the edges of graphs can bear labels. Finite state automata, for example, bear labels on their arcs. Syntax trees (e.g., phrase structure trees in linguistics, or graphical representations of expression trees in general) are examples of graphs with labels on their nodes. There exist also many graphs in which both nodes and arcs are labeled. One important example of this is *typed feature structures* which play a large role in computational linguistics and, increasingly, in logic programming in general. Examples of undirected graphs are rarer in computational linguistics, although they are very common in artificial intelligence research. A road map which indicates distance in kilometers between points is an example of an undirected graph: the distance is the same no matter which direction you travel. There are many applications of this essential idea, of *weighted* graphs, in knowledge representation, and these are important in computational linguistics in representing the lexicon, especially. Another important class of undirected graphs are *constraint graphs*, which are used to help solve many hard combinatorial problems, where the choice of a value for one variable has consequences for the possible values of other variables. Then nodes can be used to represent the variables, and arcs are labeled with *constraints* that relate these variables.

Formally, graphs are very simple. They are just binary relations over an arbitrary set. Note that this set, the set of nodes, can really contain anything at all. As long as we do not need to worry too much about the internal details of our nodes, they can be pretty much anything. Given such a set—let us call it $N$, for "nodes"—we consider next the set of all pairs of nodes, that is, the cartesian product $N \times N$. A binary relation $R$ on $N$ is just a subset of $N \times N$:

$$R \subseteq N \times N$$

If this set of pairs, this relation, is *symmetric*, that is, if for every pair $(x, y)$ which we find in $R$ we can also find its reversal $(y, x)$, then the graph $\langle N, R \rangle$ is *undirected*.

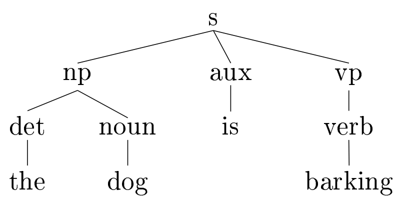
We can add more structure to this basic notion easily. For example, we could add a set of *node labels*—let us call it $\Sigma$—and a function $\lambda : N \to \Sigma$ which assigns labels to nodes. Similarly, we could define a (partial) two-place function mapping pairs of nodes (i.e., edges or arcs) to some predefined set of arc labels. Alternatively, we can extend the arc-relation to a 3-place relation like `arc(Node1,Label,Node2)`. This has the advantage that it actually "looks like" an arc in a, well, *graphical* representation of a graph.

Consider, for example, the expression

```
s(np(det(the),noun(dog)),aux(is),vp(v(barking)))
```

This corresponds to the expression tree which we can draw as in figure 13 This

**Figure 13** A phrase structure tree is a graph too.




tree in turn corresponds to a graph of the following sort. The choice of node names is arbitrary. We can, for example, use tree-addresses to name the nodes of the graph.

$$N = \{\epsilon, 1, 2, 3, 11, 111, 12, 121, 2, 21, 3, 31, 311\}$$

Then our graph has the following edges.

$$R = \{(\epsilon, 1), (\epsilon, 2), (\epsilon, 3), (1, 11), (1, 12), (11, 111), (12, 121), \\ (2, 21), (3, 31), (31, 311)\}$$

Furthermore, given a set of labels like

$$\Sigma = \{\text{the, dog, is, barking, s, np, det, noun, aux, vp, verb}\}$$

then we can define a node-labeling function as follows.

$$\begin{aligned} \lambda : \quad \epsilon &\mapsto s \\ 1 &\mapsto np \\ 11 &\mapsto det \\ 111 &\mapsto the \\ 12 &\mapsto noun \\ 121 &\mapsto dog \\ &\vdots \end{aligned}$$

Note that this function is itself a graph! Its nodes are drawn from both $N$ and $\Sigma$ of the previous expression-tree graph, and its arcs are defined by the $\mapsto$ relation. It is, to be sure, a rather special sort of graph: every node has at most one arc leading out of it, for example. (This is what makes a relation a function, after all.) But it is a legitimate graph nonetheless and many general-purpose graph processing algorithms actually make perfect sense when applied to the graph of a function.

Trees too are rather special instances of graphs. First, they are *rooted.* That is, there is a single node from which all other nodes can be reached by travelling along the arcs from node to node. Furthermore, though any given node can have any number of arcs leading away from it, no node ever has more than one arc leading into it. (The root has none, every other node has one.) Equivalently: there is exactly one path from the root to every node of the tree. Finally, in all of the examples we have considered, the nodes of the tree were interpreted as being in some definite left-to-right order. This is in fact not always assumed, even for trees: what we have been looking at are often called, specifically, *ordered* trees. Still less is this a condition on graphs in general. Finally, a tree contains no cycles. That is, though every node is an ancestor of itself, no node is a *proper* ancestor of itself. If you travel down the tree you will never find yourself above where you started! So trees are rooted, ordered, directed, acyclic graphs having for every node a unique path from the root.

### 7.1.1 Representing Graphs

In Prolog, all data structures are encoded as logical terms, and logical terms, being expression trees, are *a fortiori* trees (ordered trees, in particular). Graphs, in their full generality, disobey many of the conditions that define trees. There can be many "roots", that is, there can be any number of nodes from which all other nodes are accessible. There may be nodes which are not accessible from *any* other node. Since a node can have arbitrarily many incoming as well as outgoing arcs, it is no longer true that every node has a unique parent. One result of this is that it very often makes no sense to think of nodes as being left-to-right ordered. Finally, we need to allow for loops: now we have structures with a finite number of nodes and a finite number of edges, but having also infinitely long paths! So the very first problem we face is that we can no longer encode the data of interest directly as Prolog terms. We will have to define terms (i.e., expression trees) which bear a more abstract relation to the actual data objects. What this means in practical terms is that we can no longer count on the data structure definition to give us the skeleton of an all-purpose graph-processing algorithm. In writing solutions to problems over graphs we have to concentrate more on the intended semantics of our data structures and less on their raw syntactic structure.

Let us try to get clearer what we mean by "abstractness" here, because it will have definite computational consequences. When we design data structures we have in mind beforehand some particular kind of mathematical structure which we want to represent. Individual data structures then *denote* individual structures of the relevant sort. Up until now this denotation relation has been pretty direct and concrete. Let us look at some examples.

We write $[\![\texttt{T}]\!]$ for the denotation of a Prolog expression tree `T`. Then we can assign denotations to successor terms in terms of the structure of ordinary arithmetic with the following rules.

$$\begin{array}{rcl} [\![\texttt{0}]\!] & = & 0 \\ [\![\texttt{s(N)}]\!] & = & 1 + [\![\texttt{N}]\!] \end{array}$$

Observe first that every natural number has a unique successor term representation. Also, the clauses in the definition of the denotation function exactly mirror the clauses in the data type definition. In a similar way we can assign denotations to lists in terms of strings. A full definition is complicated by the fact that we can have lists of lists; we restrict ourselves here to the case of lists whose members are all drawn from a particular alphabet, given in advance. Then, writing $u^\frown w$ for the concatenation of string $u$ with string $w$, we have the following rules.

$$\begin{array}{rcl} [\![\texttt{[]}]\!] & = & \varepsilon \text{ (the empty sequence)} \\ [\![\texttt{[H|T]}]\!] & = & \texttt{H}^\frown[\![\texttt{T}]\!] \end{array}$$

Finally, we can assign denotations to tree data structures in terms of trees. Again, we restrict ourselves to the case where the node labels of our trees range only over a set of symbols given in advance.

$$\begin{array}{rcl} [\![\texttt{R/[]}]\!] & = & \texttt{R} \\ [\![\texttt{R/[T1, ... ,Tn]}]\!] & = & \begin{array}{ccc} & \texttt{R} & \\ [\![\texttt{T1}]\!] & \ldots & [\![\texttt{Tn}]\!] \end{array} \end{array}$$

With both lists and trees there is a unique data structure for each individual in the target structures, and the clauses defining the denotation function mirror closely the clauses defining the data type.

Finally, we think of lists and trees as ways of storing information. In this connection we note that there is a difference between a position in a list, for example, and the term which is stored there, and the same thing holds for trees. In particular, a single data item can be stored in many different positions in a list or a tree. But "the third position in list `L`," defines a unique individual: two things which are both the third position in list `L` must necessarily be the same thing. Similarly with trees: there may be many different occurrences of the label `np` in a tree, but there is at most one node which is the second daughter of the first dauther of the second daughter of the root. In what follows we will refer to as *labels* the data items we are storing in a structure, that is, the things which can appear any number of times. We will refer to as *names* the descriptions we use to pick out individual storage locations. We observe immediately that there is a one-to-one relation between storage locations in a list or tree data type and individuals (string positions or tree nodes) in their intended denotations.

These are all properties which we will not be able to maintain when we try to represent (general) graphs.

The simplest representation (though not the most efficient) is to encode the set of graph nodes as a list of node names and the arc relation as a list of pairs of node names. For example, the arc relation for the tree in figure 13 could be represented (rather sloppily) as the list

```
[arc(s,np), arc(np,det), arc(det,the), arc(np,noun),
                 arc(noun,dog), ... ]
```

Note that this encoding of the graph collapses the arc relation and the node labeling function. We can only use this trick when each node has a unique label! If for example we constructed the phrase structure tree for the sentence *the dog is barking at a cat*, which contains a second noun phrase—*a cat*—then we might have two nodes both mapped to the symbol `np`. Then we would risk confusing the paths leading out of the first `np`-node with those leading out of the second! The difference between a node *name* and a node *label* is a little subtle but it boils down to just exactly this point. Many nodes can share the same label, but if two nodes have the same name then they are the same node.

In this case we would need something like the less-readable approach illustrated below, where `R` encodes the arc relation of the graph and `L` encodes the labeling function (as another graph).

```
R = [arc(e,1), arc(1,11), arc(11,111), arc(1,12), arc(12,121),
      ... ],
L = [arc(e,s), arc(1,np), arc(11,det), arc(111,the), arc(12,noun),
      arc(121,dog), ... ]
```

Of course, these two graphs do not have to be kept distinct.

```
LR = [arc(arc(e,s),arc(1,np)), arc(arc(1,np),arc(11,det)), ...
      ]
```

We can make this easier to read! We could use an infix function symbol for the labeling arcs,

```
LR2 = [arc(e-s,1-np), arc(1-np,11-det), ... ]
```

or, perhaps even better, we could reverse the order of the arguments, treating the node names as "subscripts" on the category labels.

```
LR3 = [arc(s-e,np-1), arc(np-1,det-11), ... ]
```

We can, of course, also use an infix function symbol for the dominance arcs. Let's say, oh, to pick one at random, suppose we use '/'?

```
LR4 = [(s-e)/(np-1), (np-1)/(det-11), (np-1)/(noun-12), ... ]
```

Now things are maybe starting to look more familiar. We can take one more step in this direction, in fact, if we associate with each node a list of its outgoing arcs:

```
LR5 = [(s-e)/[np-1,aux-2,vp-3], (np-1)/[det-11,noun-12],
      (det-11)/[the-111], (the-111)/[], ... ]
```

This is pretty much as close as we can get to our representation of trees as '/'-terms, and still allow ourselves a representation which is flexible enough to encode general graphs.[1] This kind of representation gives a Prolog version of a common approach to representing graphs using *adjacency lists*. We have a list of structures which pair each node with a list of its immediate "neighbors" in the graph. This representation is abstract in the sense that there can be several occurrences of a given node-name in this structure: once on the left of the '/' symbol and maybe many times in the adjacency lists of other nodes. Yet each occurrence of this node-name expression is meant to denote exactly the same node in the graph which the data structure "means". So there are many nodes in the adjacency list expression tree which correspond to a particular individual node in the intended graph. We have thus the following correspondence.

| Expression Tree Nodes | | Graph Nodes |
|---|---|---|
| names | | |
| labels | ⟵ | names |
| | | labels |

Therefore our graph-processing predicates must adhere to this intended interpretation which is not given "for free" in the data structure. This means that the essential processing step in a graph-predicate will be one in which a node-name found in some adjacency list is used as a key to access that node's own adjacency structure from the list.

More concretely, suppose we want to get from node `a` to node `c` via node `b`. First, the symbol `b` must appear in `a`'s adjacency list.

```
Graph = [ ... ,a/[ ... ,b, ... ], ... ]
```

Then the node named `b` must have its own record in the graph structure:

[1] Observe that we have lost an explicit representation of the left-to-right order of our nodes. This can be recovered in one of two ways: either from the tree addresses assigned to each node, or, if we have been careful about the order in which nodes are entered in the list, we can recover it from there as well. Otherwise, we must have one graph for the dominance relation and another for the precedence relation; i.e., a graph with one set of nodes, but two types of arc.

```
Graph = [ ... ,b/BsNeighbors, ... ]
```

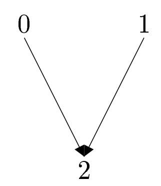
Presumably `c` is one of `b`'s neighbors, that is, `BsNeighbors = [ ... ,c, ... ]`. Finally, `c` has its own record in the graph `Graph`. Once we access that record, we can say that our procedure has moved from `a` to `c` in the graph. The key step, semantically, is the one in which we behave as if the occurrence of `b`, say, in `a`'s adjacency list was directly connected to the occurrence of `b` in the expression that `b/BsNeighbors` is meant to match. Compare this with the situation in tree structures. The "occurrence" of a node as one node's daughter and the "occurrence" of that node as the root of its subtree are in fact the same occurrence of that symbol. In the case of graphs we actually have to do some extra processing (namely by searching the graph for an occurrence of `b/BsNeighbors`) to enforce this *semantic* identity of two distinct (but, okay, very similar looking) *syntactic* objects.

There are some minor variations on this theme which are sometimes useful. First, we can easily accomodate arc labels in this representation. We simply code entries in the adjacency list as (arc-label, node) pairs, and nodes, as before, as (node-name, node-label) pairs. So, for example, we might have

```
[(1-s)/[subj-(2-np),pred-(3-vp)], ... ]
```

In some circumstances we may be able to use the position of a node's adjacency record in the graph as the name of that node. Suppose that we have the unlabeled graph in figure 14. We can represent this simply as the following list of

**Figure 14** A sample graph with no labels.




lists.

```
[[2],[2],[]]
```

We can use calls to a suitable version of `nth/3` to access nodes. (By "suitable" we mean that in this case it must treat the first element of a list as element number 0.)

Note before we go on that there is one other way of representing graphs which may have occurred to you already if you thought about the finite state automaton example. Since the arc-relation is, after all, a relation, and since Prolog programs are, after all, ways of defining relations, we can represent graphs directly as Prolog programs.

```
arcs( s-e, [np-1,aux-2,vp-3] ).
arcs( np-1, [det-11, noun-12] ).
arcs( det-11, [the-111] ).
arcs( the-111, [] ).
     ⋮
```

This obviously won't work if we are using a graph which we intend to modify, or create from scratch.[2] But it works well when we are given a graph which represents some static data structure which we want to use, but not to modify. The finite state automaton's `delta`-relation is a good example of this. Similarly, a graph representing distances between cities is also a good example of such static data. For most natural language applications, a semantic network encoding the lexicon may be expected never to change while a program is running, though here, with the possibility of sophisticated learning algorithms and so forth, the case is not quite so clear cut.

## 7.2 First the Simple Cases

Let us set out first to define the data structure for an unlabelled graph. Essentially, the structure we are interested in is just a list of `NodeName/Neighbors` terms, where `Neighbors` is a list of node names. We refer to these terms as *adjacency records*. We choose as well to impose the additional condition that every node name appearing in a neighbors-list must have its own adjacency record somewhere in the graph structure.

```
graph(G):-
    forall_nodes(G,G).

forall_nodes([],_).
forall_nodes([_Nm/Nbrs|Rest],Graph):-
    for_all_neighbors(Nbrs,Graph),
    forall_nodes(Rest,Graph).

for_all_neighbors([],_).
for_all_neighbors([N1|Ns],Graph):-
    member(N1/_,Graph),
    for_all_neighbors(Ns,Graph).
```

Observe that `for_all_nodes/2` will have the effect of verifying that every member of the graph is in the form of an adjacency record. Then `for_all_neighbors/2` will make sure that every neighbor of the current node has an adjacency record.

The graph must appear twice as an argument. The first instance acts as a sort of *agenda*, which tells us which adjacency records remain to be checked. We do a recursion on this structure, so it shrinks as processing proceeds. The

[2] Well, actually it will work. Prolog does allow programs to modify themselves. But you're not supposed to know about that yet!

second argument position never changes. In effect the graph acts as a global variable which is carried around throughout the procedure, so that we can refer to it as needed.

We now consider a simple graph traversal predicate. Note that, for the first time, our data type definition is *not* a traversal predicate. It processes adjacency records in the order in which they are found in the graph data structure, and that order need have no relation whatsoever to the way in which graph nodes are connected to each other. It does visit every node in the graph, so in that sense it is a traversal, but the traversal visits nodes in an essentially random order.

The traversal predicate we now examine is an example of *depth-first search* of a graph. We begin with the simplest case, a case where the graph is rooted (and so a fortiori connected as well) and acyclic. Note that a rooted acyclic graph can have only one root. Suppose it had two. Call them nodes $i$ and $j$. Then, by definition of a root, $j$ is reachable from $i$ and likewise $i$ is reachable from $j$. But then $i$ is reachable from $i$, via $j$, and so there is a loop, which contradicts the assumption. We make the further simplifying assumption that the root is always the first record in the graph. The predicate `df_traversal/2` takes a graph and returns a list of the nodes visited, in order.

```
df_traversal(Graph,[Root|Visited]):-
    Graph = [Root/Neighbors|_], % get the root's neighbors
    dft(Neighbors,Graph,Visited).

dft([],_,[]).
dft([N|RestNs],G,[N|Visited]):-
    member(N/Nbrs,G),
    dft(Nbrs,G,VisitedFromN),
    dft(RestNs,G,VisitedFromRest),
    append(VisitedFromN,VisitedFromRest,Visited).
```

Compare this with the equivalent code for a tree structure.

```
df_traversal(Tree,[Root|Visited]):-
    Tree = Root/Subtrees, % get the root's subtrees
    dft(Subtrees,Visited).

dft([],[]).
dft([N/Subs|RestNs],[N|Visited]):-
    dft(Subs,VisitedFromN),
    dft(RestNs,VisitedFromRest),
    append(VisitedFromN,VisitedFromRest,Visited).
```

The main difference is that with trees, when we pick a node it comes along with all of its subtrees. So the call to `member/2` in `dft/3` is not needed in `dft/2`, and as a consequence we do not need the extra argument.

This definition of depth first traversal works as advertised as long as the graph is rooted and acyclic. What do we do if we are given a graph which does not meet these conditions? Suppose first that we are given an unrooted graph. Then for any node $x$ in the graph there is at least one node which cannot be reached from $x$. A fortiori, then, that node cannot be reached from $x$ by a depth first traversal. The strategy to follow in this case is to start from some arbitrary node, visit all the nodes that are reachable from that node (taking them in depth first order); then pick some other arbitrary node from among those that we were not able to visit, and start again from there. We simply iterate this procedure until we have visited every node.

Now consider the problem of cyclic graphs. Assume we have a graph such as that in figure 15, on page 100. The problem there is that, starting from `a` we might go on to node `b`, and then we might choose `a` again from `b`'s adjacency record. Now the whole procedure repeats itself, and it can continue to do so as long as we like. Basically, cyclic graphs are bottomless, so doing a depth-first search is bound to get us in trouble! The strategy here is based on the following observation. Once we have visited node $x$, then, by the nature of a depth first search, we are certain to visit $x$'s neighbors. So if at any point we encounter $x$ again, we might as well just skip it. We know we've been that way before already and there is nothing new to find there. Loops are a special case of this situation. Given a finite graph, we cannot possibly get into an infinite traversal if we never visit any node twice.

It seems, then, that the strategies for solving both problems have an interesting symmetry.

- For unrooted graphs, we need to know which nodes we *have not* visited.
- For cyclic graphs, we need to know which nodes we *have* visited.

So it would appear that the same strategy would suffice for both cases. We consider first the problem of cycles.

## 7.3 Dealing with Cycles

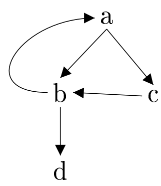
Suppose we have the graph illustrated in figure 15. The set of nodes is just the four element set $\{a, b, c, d\}$, and the arc relation is as given below.

```
arcs( a, [b,c] ).
arcs( b, [a,d] ).
arcs( c, [b] ).
arcs( d, [] ).
```

Naively, we might define a `path/3` predicate for this graph in essentially the same way that we implemented it for trees, that is, by simply taking the reflexive transitive closure of the `arc` relation, compactly encoded by the `arcs/2` predicate.

```
path( X, Y, [] ) :- X = Y.
path( X, Y, P ) :-
```

**Figure 15** A sample graph.




```
        arcs( X, ArcList ),
        member( Z, ArcList ),
        P = [Z|PTail],
        path( Z, Y, PTail ).
```

However, because there is a loop in the graph, taking us from node `b` back to node `a`, there is an infinite branch in the search tree for `path/3`. It gets worse! Because the upward pointing arc—from `b` back to `a`—will be found first by `member/2`, before the downward pointing arc to `d`, Prolog will find this infinite branch before it finds any other. And because the clause for `b` is earlier than the clause for `c` in the definition of `arcs/2`, note that Prolog will even fail to find a path from `a` to `c`! Clearly, if we want to be able to find paths in an arbitrary graph, something extra has to be done.

Recalling the example of finite state automata, where we looked at an automaton that gave us essentially the same problem, we already know one solution we can apply. We observe that the data structure definition predicate `list/1` will enumerate the lists in order of increasing length. Given that paths are encoded as lists of node names, we can redefine the problem as follows.

```
list([]).
list([_|L]):- list(L).

?- list(P), path(a,Y,P ).
```

This technique, where we simulate breadth-first search using repeated bounded depth-first searches actually has a name: it is called *iterative deepening*. Here each successive call to `path/3`, in effect, deepens the search by one level of the search tree.

This solution, by the simulation of iterative deepening, still finds an infinite number of (ever longer) paths from `a` to `d`, for example. Almost all of these paths, however, are arrived at by following a loop in the graph. There are only two ways of reaching `d` from `a` which are truly unique: `[a,b,d]` and `[a,c,b,d]`. All the others are arrived at by following the loop. What want to do here is not to find all of the paths but rather all of the *cycle-free* paths.

A neat way to do this is to note that, after all, we are computing a path from one node to another. To find out if we have visited a particular node yet, we should be able to just search the path we are constructing to see if that node is on it yet. However, path/3 is not designed in such a way that we can do this intuitively appealing thing. The part of the path that we have already seen is not represented as a variable in the code, so we have no access to it; the only part of the path which we can actually examine is the tail of the path, the nodes we are going to visit in the future, and this is no help at all to us! So we modify the code slightly so as to keep track of the nodes we have actually visited, giving us path/4, below.

```
path( X, Y, P ) :- % path/3 redefined
    path( X, Y, [X], P ).

path( X, X, Path, Path ).
path( X, Y, PIn, POut ) :-
    arcs( X, L ),
    member( Z, L ),
    nonmember( Z, PIn ), % Been here before?
    path( Z, Y, [Z|PIn], POut ).
```

This code uses a technique, called *accumulator pairs*, which is used everywhere in logic programming, and which we will examine in some detail when we study the control and optimization of logic programs. Here we note only that, while the logical, "declarative" semantics of this program may be at this moment somewhat obscure, it should be reasonably clear what it does. If we type in a goal like

```
?- path( a, d, P ).
```

then this code will begin by initializing the "path-so-far" argument to a list containing only the starting point of the search. Then we call path/4, which does the real work. When we finally reach our destination, and the base clause applies, then we simply pass the contents of the "path-so-far" argument PIn over to the output argument POut. This is then passed up to the calling goal and from it to the goal that called it, and so on up the chain of recursive calls to path/4 and thence to path(a,d,P), our original goal. Note that whereas in path/3 we actually built up the path "on the way up out" of the recursion, here we are building it up "on the way down in." This means that at every point, we can see where we have been so far, by examining the PIn variable. In this case we do that with a call to nonmember/2.

As usual, using nonmember/2 requires defining neq/2, in this case for node names. We can do that in the usual way, with a collection of unit clauses like "neq(a,b)." and so forth. But neq/2, like any binary relation, can be seen as defining a graph. This graph has the same set of nodes as the original graph of figure 15, but different arcs. There is an arc between two nodes in the "neq-graph" whenever they are different nodes. Note that this relation is symmetric, and so technically the neq-graph is our first example of an *undirected* graph, but we can still use the same adjacency list representation scheme, as follows.

```
neq( Node1, Node2 ) :-
```

```
    neq_arcs( Node1, L ),
    member( Node2, L ).

neq_arcs( a, [b,c,d] ).
neq_arcs( b, [a,c,d] ).
neq_arcs( c, [a,b,d] ).
neq_arcs( d, [a,b,c] ).
```

With this code in place, we can now safely search in a graph, even when it contains cycles.

This approach works well when the graph is represented as a Prolog program, with an `arcs/2` predicate defined by a collection of unit clauses. There is another even neater approach which we can apply when we represent our graphs using adjacency lists. This approach uses `select/3` to mark a node as visited by actually *removing that node from the graph.* Suppose, then, that our graph is encoded as an instance of the `graph/1` predicate.

```
graph( 1, [
    a/[b,c],
    b/[a,d],
    c/[b],
    d/[]
    ] ).
```

(This technique is very useful in general for encoding a set of test data, especially when the data involve very complex Prolog terms.) Now we want to define a `path`/4 predicate so that we can issue queries like

```
?- graph( G ), path( a, d, G, P ).
```

Accordingly, we reimplement `path` so that it includes an argument position for the graph itself.

```
path( From, To, Graph, [From|Path] ) :-
    p( From, To, Graph, Path ).

p( X, Y, _G, [] ) :- X = Y.
p( X, Y, G, [Z|P] ) :-
    select( X/Arcs, G, SmallerG ),
    member( Z, Arcs ),
    p( Z, Y, SmallerG, P ).
```

Note that each time through the recursive clause of `p/4` we reduce the size of the graph by removing one adjacency record from it. The `select/3` predicate fails when its list argument is empty, so we can only call clause 2 a finite number of times before it fails to apply, and clause 1 is nonrecursive, so we can only call it once in any case. So no matter which clause of `p/4` Prolog selects, the recursion is bound to terminate. From this one sees immediately that we cannot possibly get caught in a loop: that would contradict the theorem that this program is bound to terminate.

Now we can use the same logic to reimplement our depth first traversal predicate so that it handles (rooted) cyclic graphs. The only essential difference comes from the fact that this is a traversal predicate rather than a search predicate. With `p/4`, if the call to `select/3` fails (the node has already been visited), then the whole call to `p/4` fails, we backtrack and try to find a different path. With `dft/4`, if the call to `select/3` fails, we must simply ignore that node and continue. Therefore we need a call to `nonmember_/2` so as to be certain that we only ignore nodes that we really have already visited. Similarly, where `p/4` calls `member/2` to nondeterministically select one way to extend its current path, `dft/4` must execute a recursion on the entire list of neighbors, so as to check all ways to traverse the graph.

```
df_traversal(GIn,[Root|Route]):-
    GIn = [Root/Neighbors|GInRest], % get the root's neighbors
    dft(Neighbors,GInRest,_GOut,Route).

dft([],RemainingG,RemainingG,[]).
dft([N|RestNs],GIn,GOut,[N|Route]):-
    select(N/Nbrs,GIn,SmallerG),
    dft(Nbrs,SmallerG,EvenSmallerG,RfromN),
    dft(RestNs,EvenSmallerG,GOut,RfromRest),
    append(RfromN,RfromRest,Route).
dft([N|RestNs],GIn,GOut,Route):-
    nonmember_(N,GIn), % already visited
    dft(RestNs,GIn,GOut,Route).
```

The predicate `nonmember_/2` which is called in clause 3 of `df_search_2/3` is not quite the same as the predicate `nonmember/2` we have seen before, although it is about as much different from that predicate as its name implies. The trick is that we are asking whether or not a *node* is still present in a *graph* and a graph is not a list of nodes, but a list of node/adjacency-list pairs. The code for `nonmember_/2` looks like the following and assumes that `neq/2` has been implemented.

```
nonmember_( _, [] ).
nonmember_( X, [Y/_|L] ) :-
    neq( X, Y ),
    nonmember_( X, L ).
```

So we just pull out the node-part of each node/list pair and check *that* for inequality with `X`.

## 7.4 Routes without Roots

With trees, we could find any node simply by enumerating the paths from the root. We can now do the same for graphs in a way which makes us immune from loops, but we still must assume that there is at least one root. With general graphs we are not so fortunate. There may be many pairs of nodes in a graph between which there is no path.[3] So the various implementations of the `path` predicate are not yet as much help to us as they could be. The path problem introduced us to ways of dealing with loops, but we still have no general-purpose way to be sure that we have visited every node in a graph.

Suppose that we are given a graph `G`. We can nondeterministically select a node `N` from `G`, and begin a traversal of `G` from `N`. Unless we are very lucky, both in our choice of `G` (it is connected and rooted) and of `N` (it is one of `G`'s roots), a traversal of `G` starting from `N` will not reach all of `G`'s nodes. However, we can iterate the procedure on the remaining unvisited nodes, biting off a chunk of `G` at each iteration, until we have finally visited all of `G`'s nodes.

The definition of `df_traversal/2` on page 103 is almost all we need. In particular, the subroutine `dft/4` is already set up to return, in its third argument, whatever remains of the graph when it is done. In the previous definition we simply threw away whatever remained of the graph after the top level call to `dft/4` from `df_traversal/2`. So the only modification we need is to replace the old 'one shot' definition of `df_traversal/2` with a new recursive one, which calls itself repeatedly until the remainder of the graph is empty. Also, note that instead of starting with the first node in the graph, we call `select/3` to choose a starting point nondeterministically. Since we cannot guarantee that our graph has roots, we cannot guarantee that its first element is such a root. So as far as we know one starting point is as good as another. (Actually, some starting points are much better than others, but there is no easy way to know in advance which those are ... ) This means that this predicate has many more solutions than the rooted version. We can restrict it to return only one solution by always choosing the first node in the graph as our starting place. Finally, the call to `append/3` in the recursive clause of `df_traversal/2` inserts an '`*`' to mark the points at which a new "pseudo-root" was selected. This divides the output into what we might think of as "depth-first-reachable components", approximately.

```
df_traversal( [], [] ).
df_traversal( G0, [N|Route] ) :-
    select( N/Ns, G0, G1 ),  % select N as a starting point
%    G0 = [N/Ns|G1], <- fewer solutions
    dft( Ns, G1, G, Route1 ), % search from N's neighbors
    df_traversal( G, Route2 ),
    append(Route1,[*|Route2],Route).

dft( [], G, G, [] ).
dft( [N|Rest], G0, G, [N|Route] ) :-
    select( N/Ns, G0, G1 ),  % N is unvisited; mark it visited
```

[3] As a useful "boundary case" consider the graph with an empty arc-relation: no node is connected to any other! Or the other end of the scale, where every pair of nodes are symmetrically connected to each other, i.e., where the arc relation equals $N \times N$. Here any node at all could serve as a root, and there are loops everywhere. We will need to handle both of these cases with one solution.

```
        dft( Ns, G1, G2, RouteL ), % go down from N
        dft( Rest, G2, G, RouteR ), % now go across
        append(RouteL,RouteR,Route).
    dft( [N|Rest], G0, G, Route ) :-
        nonmember_( N, G0 ),  % N was visited
        dft( Rest, G0, G, Route ). % so ignore it and keep going
```

As in the various path predicates, we use the graph-reduction technique to mark nodes as visited. So we use select/3 to access the graph data structure, which simultaneously checks for membership in the graph and deletes the requested element if it is a member (and of course failing if the requested item is not an element of the graph).

This predicate also seves as another illustration of the *accumulator pair* technique. The careful reader will have noted that we always speak of "the" data structure representing the graph, when in fact we use a multitude of different variables to represent it. For example, in clause 2 of dft/4, "the" graph appears as the value of G0, G1, G2 and G, each of which are variables with possibly quite distinct values. For now it is probably best to understand accumulator pairs in a procedural manner. From this point of view one can see these chains of graph-valued variables as tracking the evolution of the graph over time. This "evolutionary" point of view is considered procedural rather than logical because the introduction of a notion of time brings with it a notion of order: "first it looks like this, then we change it and it looks like that." But the logical interpretation of a clause body is a conjunction of sub-goals, and conjunction is commutative, and so order-free. In the next chapter we will consider the logical semantics of accumulator pairs, and discover thereby some of the most powerful techniques of logic programming. But for now, one should just try to follow how each successive "G-variable" marks a phase in the evolution (here "evolution" = "reduction") of the graph.

The general structure of df_trav/2 is essentially the same as any other simple recursive predicate. In particular, since graphs are implemented as lists of a certain sort and the depth-first search task is essentially a universal task requiring that the entire graph be traversed, it is completely unsurprising that the base case is the case where the graph is empty. The recursion here is not, however, exactly the familiar recursion-on-a-structure. In particular, each step of the recursion does reduce the structure, but the nature of this reduction bears no relation to any set of generating functions which might be used to build up the set of graphs out of primitive elements. Abstractly, we are treating graphs as essentially being built up out of "depth-first-reachable" components, sets of nodes which are reachable by a depth-first traversal. Each call to dft/3 removes one of these components (a set of nodes, together with all of the arcs leading out of those nodes), so each recursive call to df_trav/2 does reduce the size of the graph structure, so the recursion is "well founded" and the program will terminate. However, once again, the recursion is on an abstract structure, an interpretation of the graph as being made up of "depth-first-reachable components" which is not directly related to any feature of the actual Prolog data structure we use to implement and describe the graph.

### 7.4.1 Other search orders

Consider again the code for `dft/3`, in particular clause 2.

```
dft( [], G, G ).
dft( [N|Rest], G0, G ) :-
    select( N/Ns, G0, G1 ),  % N is unvisited; mark it visited
    dft( Ns, G1, G2 ), % go down from N
    dft( Rest, G2, G ). % now go across
dft( [N|Rest], G0, G, Ts ) :-
    nonmember_( N, G0 ),  % N was visited
    dft( Rest, G0, G, Ts ). % so ignore it and keep going
```

Clause 2 is the clause that does most of the actual work. If we were to apply this general algorithm to a particular problem (as we will in the next subsection) this is the clause where we would do the actual computing that adds information to the output variable(s), since this is the clause in which we have just accessed a brand new unvisited node and its neighbors.

This clause is doubly recursive, calling itself twice on different inputs. This is to be expected if we consider graphs to be generalizations of trees, since the main computational feature of trees is the double recursion ("down" and "sideways") which they require. The only wrinkle is that the double recursion is a call to the very same predicate, but this is not a computationally very significant difference. But now we ask ourselves: does it make a difference in which order we call these two recursions? Well, yes and no. We still visit all the reachable nodes, but we visit them in a very different order, namely *breadth-first.* Informally, we think of `Rest` as being the *old* nodes, nodes we have known for some time that we must eventually consider, while `Ns` are *new* nodes, that we have just discovered we must consider. Depth first strategies can be characterized intuitively as *Explore the new stuff first; postpone the old stuff,* while breadth first strategies can be characterized intuitively with the slogan *Explore the old stuff first; postpone the new stuff.*

Now consider the following slight modification of `dft/3`, clause 2:

```
dft_b( [N|Rest], G0, G ) :-
    select( N/Ns, G0, G1 ),  % N is unvisited; mark it visited
    append( Ns, Rest, Agenda ),
    dft_b( Agenda, G1, G ).
```

This is equivalent to the old version. In particular, the choice of breadth-first vs. depth-first search order can still be made with an almost trivial modification, namely we replace

```
append( Ns, Rest, Agenda )
```

with

```
append( Rest, Ns, Agenda )
```

We think of the list Agenda as ordering the tasks which our program has to perform. If we treat Agenda as a *stack*, that is, always adding things at the front, and removing them from the front, then we get the depth-first version illustrated here. On the other hand, computer science offers us an alternative to stacks called *queues*, which are sequences where we remove things from the front, as with stacks, but we add things *at the back*. This means that, as a new task is added to the back of the queue, it will not be selected for processing until all the tasks that were already in the queue when it was added have been selected for processing. So, the first version of append/3—append( Ns, Rest, Agenda)—adds new material (Ns) to the front, treating Agenda as a *stack*, while the second version—append( Rest, Ns, Agenda )—adds the new material to the back, treating Agenda as a *queue*. We could in fact substitute for the terms "depth-first" and "breadth-first" the terms "stack-based" and "queue-based".

Note that there is nothing sacred about stacks and queues. There are many more ways than these to order a collection of tasks. In particular, both strategies always select for processing the first (leftmost) item. This is reflected here in that we unify Agenda with [N|Rest], selecting its head, N, for processing. Consider yet another version of this basic logic:

```
dft_c( OldAgenda, G0, G ) :-
    choose( N, OldAgenda, Rest ),
    select( N/Ns, G0, G1 ),
    append( Ns, Rest, NewAgenda ),
    dft_c( NewAgenda, G1, G ).
```

Here we have replaced the extraction of N from the agenda by means of unification with [N|Rest] with an explicit call to a yet-to-be-defined predicate choose/3. Our current version of graph traversal would simply implement choose/3 as follows.

```
choose( N, [N|Rest], Rest ).
```

However, we could make our choice of N arbitrarily sophisticated, by modifying the definition of choose/3.

```
choose( N, List, Rest ) :-
    select( N, List, Rest ),
    okay( N ).
```

The above clause implements a version of choose/3 which nondeterministically selects an N which is okay, whatever we decide that that should mean.

The last version of depth first traversal which we will consider here returns a *depth-first search forest*, instead of a list. This gives us a more informative picture of how the graph was traversed. In particular, it tells us how we got to the nodes we visited. The idea is this. Depth-first search does not visit any node twice. Therefore, for any node, we can provide a unique sequence of arcs which were used to reach it. This must be true because if the sequence of arcs were not unique, then there would have to be some node which was accessed twice, which we have explicitly avoided doing; so we derive a contradiction. Therefore, for each starting place that we select in `df_trav/2`, there is a unique path to every node which was visited on the corresponding call to `dft/3`. This is sufficient to show that the set of nodes which are "reachable" from some node `N` form a tree: the set is rooted (in `N`) and every member of the set has a unique path from the root. In addition we have that the set forms an ordered tree, since for every node in the tree its immediate subtrees are visited in some definite order. So each starting node defines a tree, so a sequence of starting nodes defines a sequence of trees. This sequence is a *depth-first search forest*. It is unique for every choice of sequence of starting nodes, but note that that choice is not unique for a given graph, since it is determined by calls to the nondeterministic predicate `select/3`. So any given graph can have many different search forests.[4]

```
    df_trees( [], [] ).
    df_trees( G0, [T1|Ts] ) :-
        select( N/Ns, G0, G1 ),
        T1 = N/Ts,
        df_trees_aux( Ns, G1, G, Ts ),
        df_trees( G, Ts ).

    df_trees_aux( [], G, G, [] ).
    df_trees_aux( [N|Rest], G0, G, [T|RestTs] ) :-
        select( N/Ns, G0, G1 ),
        T = N/SubTs,
        df_trees_aux( Ns, G1, G2, SubTs ),
        df_trees_aux( Rest, G2, G, RestTs ).
    df_trees_aux( [N|Rest], G0, G, Ts ) :-
        nonmember_( N, G0 ),
        df_trees_aux( Rest, G0, G, Ts ).

/*
```

## 7.5 Graph Unification—Simultaneous traversal of two graphs

In this section we explore a more sophisticated application of depth frist graph traversal, namely the unification of typed feature structures represented as graphs, or TFS unification for short. This is based on a straightforward depth

[4]Note that if we make the selection of the starting node deterministic, say by always choosing the head of the current graph structure, then the sequence of starting nodes is now fully determined by any particular graph structure. Note further, however, that any given *graph* can be represented by several different *graph structures*, that is, lists of adjacency lists. For example, randomly permuting the list of adjacency records does not in any way affect which graph it represents.

first traversal algorithm, but this time we will be traversing *two graphs*, and we we will be doing it *simultaneously*.

### 7.5.1 An introduction to typed feature structures

We begin with a short description of what we will mean here by "typed feature structure". Then we will consider how to represent them using a graph data type.

We define typed feature structure terms with the following recursive type equations.

$$\langle \textit{term} \rangle = \langle \textit{tfs-struct} \rangle \vee \langle \textit{variable} \rangle \tag{7.1}$$

$$\langle \textit{tfs-struct} \rangle = \langle \textit{type-name} \rangle / \textit{list}(\langle \textit{fv-pair} \rangle) \tag{7.2}$$

$$\langle \textit{fv-pair} \rangle = \langle \textit{feature-name} \rangle \texttt{-} \langle \textit{term} \rangle \tag{7.3}$$

Observe that the definition is close to the definition of logical terms: we have structures and we have variables. (We treat constants as structures of arity 0; in this case that means type names associated with empty lists.) We emphasize that typed feature structures are in many respects generalizations of logical terms; or, alternatively, logical terms are special cases of typed feature structures: Functors correspond to types and features are represented positionally. So the main new features we have to deal with are the possibility of unifying two structures with different numbers of arguments, and the possibility that corresponding arguments may appear in different positions in their respective structures, provided they have the same feature name.

Typed feature structures give us a convenient way of representing constraints that might hold of a structure. For example, consider the three rules in table 7.1, given in the PATR-II formalism for writing constraint-based phrase structure grammars. (See Shieber, 19?? ... ). Here we see a number of constraints expressed in terms of equations. We can express essentially the same constraints using typed feature structures. In fact, the PATR-II version makes no direct use of types except to represent "constants" like *np* or *third*. For purposes of illustration we will add "functor" types `nd`, `lbl`, and `ref` (for "phrase structure node," "label" and "reference", respectively) so that now all of our feature structures are typed. The result is given in table 7.2.

We can also represent such grammatical constraints with graphs. Considering the PATR-II style rules in table 7.1, it is clear how to construct a graph from the path expressions and equations there. Paths correspond to paths through a graph, except that the path reflects the *arcs* traversed rather than the nodes visited. Equations between paths and symbols indicate that the node found at the end of the path is labeled with the symbol and equations between paths indicate that both paths lead to the same place. Typed feature structures can be thought of as graphs in the same way. Again, arcs are labeled with features and now all nodes are labeled with types. Just as with Prolog terms, the appearance of a variable in two places is taken to mean that those two nodes are the same. That is, referring to the `s`-example in table 7.2, the node reached by following

$$
\begin{array}{rcl}
0 & \rightarrow & 1 \quad 2 \\
\langle 0 \text{ cat} \rangle & \doteq & s \\
\langle 1 \text{ cat} \rangle & \doteq & np \\
\langle 2 \text{ cat} \rangle & \doteq & vp \\
\langle 1 \text{ agr} \rangle & \doteq & \langle 2 \text{ agr} \rangle
\end{array}
\qquad\qquad
\begin{array}{rcl}
0 & \rightarrow & 1 \quad 2 \\
\langle 0 \text{ cat} \rangle & \doteq & vp \\
\langle 1 \text{ cat} \rangle & \doteq & v \\
\langle 2 \text{ cat} \rangle & \doteq & np \\
\langle 0 \text{ agr} \rangle & \doteq & \langle 1 \text{ agr} \rangle
\end{array}
$$

$$
\begin{array}{rcl}
0 & \rightarrow & \text{“nature”} \\
\langle 0 \text{ cat} \rangle & \doteq & np \\
\langle 0 \text{ agr per} \rangle & \doteq & third \\
\langle 0 \text{ agr num} \rangle & \doteq & sing
\end{array}
$$

Table 7.1: Some grammar rules, PATR-II style.

```
nd/[0-lbl/[cat-s/[]],
    1-ps/[0-lbl/[cat-np/[],
                 agr-X ]],
    2-ps/[0-lbl/[cat-vp/[],
                 agr-X ]]]
```

```
nd/[0-lbl/[cat-vp/[],
           agr-X ],
    1-ps/[0-lbl/[cat-v/[],
                 agr-X ]],
    2-ps/[0-lbl/[cat-np/[]]]]
```

```
nd/[0-lbl/[cat-np/[],
           agr-ref/[per-third/[],
                    num-sing/[]]]]
```

Table 7.2: Some grammar rules, TFS style.

```
nd/[0-lbl/[cat-s/[]],
    1-ps/[0-lbl/[cat-np/[],
                 agr-X ]],
    2-ps/[0-lbl/[cat-vp/[],
                 agr-X ]]]
```

```
[nd/[0-2, 1-3, 2-4],
 lbl/[cat-5],
 nd/[0-6],
 nd/[0-7],
 s/[],
 lbl/[cat-8, agr-9],
 lbl/[cat-10, agr-9],
 np/[],
 top/[],
 vp/[]
 ]
```

Table 7.3: From feature structures to graph data structures.

the `agr` arc from the "subject" label node is identical to the node reached by following the `agr` arc from the "predicate" label node. The important thing to keep in mind is just this, that *shared variables indicate shared structure.*

There is an extra aspect to typed feature structures which is not readily apparent by comparing them to grammar rules like those in table 7.1. Namely, we can add to our grammar an elaborate `type hierarchy` which can tell us, for example, that transitive verbs are a subtype of verb or that proper names are a subtype of noun phrase. Basically, statements of type inclusion ("type $t$ is a type $s$,") are implications and support a rich array of possible inferences. In what follows we will make use of only a tiny fragment of these possibilities. We will assume one special type which we will call `top`, which is at the very top of the hierarchy: everything is a `top`. The rest of our types are all distinct and bear no relation to each other. We will make use of typed feature structures of the form `top/[]` as simple placeholders: they indicate that a particular node must exist, but we know no more about it than that it exists, and that it may be found at the end of one or another arc.

So let us consider now how to encode these graphs using our adjacency list notation. Each TFS corresponds to a node in the graph, namely the node which is its root. Node names will just be positive integers, and we adopt the convention that the node named $i$ will be stored in position $i$ of the graph. In practice, this means that we access the graph with calls to `nth/3` rather than `member/2`. As an example, consider table 7.3. There node 1 of the graph corresponds to the entire feature structure on the left, that is, node 1 represents the root of a feature structure of type `nd` whose `0` attribute has as its value the feature structure corresponding to node 2, a structure with type `lbl`, whose `cat` feature has as its value a feature structure of type `s`, corresponding to node 5, and so on.

### 7.5.2 The unification operation on graphs

In order to unify two feature structure terms we make the simplifying assumption that both terms are substructures of a single graph. We can always make this assumption since graphs, in general, are not required to be connected. We do require that whatever substructures are meant to represent typed feature structures be rooted and hence connected. But the entire graph need not be. We presume then that prior to the execution of the unification procedure both terms to be unified ("unifcands") have been stored in a single graph. So the arguments to a call to the unification procedure will be node names in the input graph, rather than entire terms.

In a rather deep sense this overall graph corresponds more closely to computational *working memory* than to a grammatical representation. Node names, following our naming convention, are addresses in this memory. So we can think of adjacency records like `lbl/[cat-8,agr-9]` as arrays of pointers, with feature names as their indices. Note however that there is a very important way in which Prolog terms (like the lists we are using as our basic data structures) are not like a computer's memory. We can instruct a computer to replace the contents of memory address $x$ with something new, but we cannot do that to a logical term. There is no assignment operation in Prolog. Instead, when we need to write programs, i.e., definite clause theories, about objects that evolve over the course of a computation, we use Prolog terms to denote *states* of this object, and write predicates which tell us how one state of the object is related to its next state. A simple example is the familiar list processing predicate `select/3`, which is a relation between two "states" of a list. The graph predicates which mark nodes as visited by removing them from the graph are a more elaborate example in which the successive states of the graph are indicated by a long string of graph-valued variables.

We will implement here a species of *destructive* unification. That means we will take our two terms (or subgraphs), construct a most general unifier and then replace the root of the first subgraph with a pointer to the root of the second, and replace the second subgraph with the unifier we have just computed. This means we will enrich our data structure with a new kind of expression, this "pointer" which I just referred to. Pointers will be represented by terms of the form `@N`, where `N` is a node name, that is, an address in our overall graph.

This adds a level of abstraction in the way our data structure represents an intuitively given graph. On the one hand our data structure is an ordinary sort of graph in which we have either labeled nodes with a collection of labeled outgoing arcs (i.e., ordinary adjacency records) or else special sorts of unlabeled nodes with a single unlabeled outgoing arc (i.e., our "pointers"; a pointer term of the form `@7` represents an unlabeled node with a single unlabeled arc leading to node `7`).

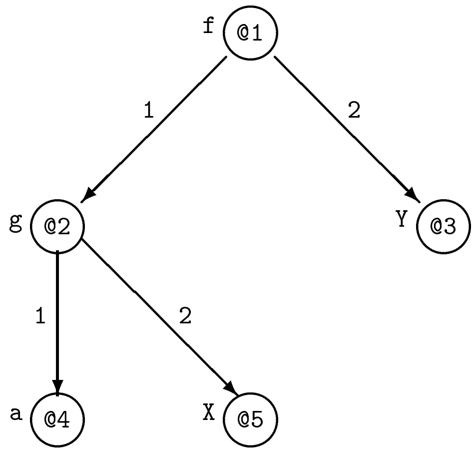
On the other hand, this graph over two sorts of nodes is meant to represent a graph over nodes only of the first sort. We can recover our intended graph by collecting together a node of the first sort (i.e., a node with a standard adjacency record) and all of the nodes of the second sort which point to it, forming an equivalence class. Each such "pointer equivalence" class corresponds to a single node in the intended graph and so, transitively, to the single feature structure of which that node is the root.

### 7.5.3 The unification algorithm

Our main predicate will be a four place relation `unify(Pointer1, Pointer2, GraphIn, GraphOut)`, which unifies the two structures whose roots are found in `GraphIn` at locations `Pointer1` and `Pointer2`. On output the most general unifier can be found at either location `Pointer2` or at `Pointer1`, which now just contains `@Pointer2`. Assuming we have a procedure for pretty-printing typed feature structures, we could print out our solution by feeding it `Pointer2` and `GraphOut`.

Let us start with the simple case of a Prolog term, for example `f(g(a,X),Y)`. This can be represented as a graph, as in figure 16. (Node names are of the form `@1`, etc., to distinguish this use of integers from the use of integers as features.) Considering the figure, it is clear how to represent such a term using

**Figure 16** The term `f(g(a,X),Y)` as a graph.




our adjacency-list representation of graphs.

```
[ f/[1-2,2-3], g/[1-4,2-5], top/[], a/[], top/[] ]
```

Keep in mind that the number on the left of the '-' is a feature name, which is just an argument position since we are talking about Prolog terms. So the expression `1-2` means that the first argument of this structure is represented by node 2. To aid the reader we will put '`@`' signs before all node names, to distinguish them from feature names. The computer, however, is not subject to so much confusion as the human reader, so keep in mind that this practice is not followed in the actual implementation.

Suppose now that we wish to unify `f(g(a,X),Y)` with `f(Y,g(a,b))`. Stored in a graph, these terms look as follows.

```
[f/[1-@2,2-@3],g/[1-@4,2-@5],top/[],a/[],top/[],
 f/[1-@3,2-@7],g/[1-@8,2-@9],a/[],b/[]]
```

Note that variables only ever appear once, but we make a new copy of a constant whenever we see one in the input term. That is, variable names are treated as naming particular locations in memory.

Now we consider how to unify our two terms, that is, how to unify the two subgraphs rooted at locations `@1` and `@6`, respectively. The algorithm will proceed as follows. We access whatever is pointed to by our two arguments, and unify them, and then we replace the first one with a pointer to the second, and replace the second one with the result of unifying them.

In a little more detail, unifying two structures generates a list of subproblems, which come in four varieties, which we will name `unify`, `type`, `point` and `copy`. A `type(T1,T2,Loc)` problem is solved by computing the greatest lower bound of `T1` and `T2` in the type hierarchy and assigning that type to the node at location `Loc`. Recall that we use a particularly simple type hierarchy which has the effect that the greatest lower bound (g.l.b.) of `top` and any `T` is just `T`, the g.l.b. of a type `T` and itself is just itself, and the g.l.b. of types `T1` and `T2` when neither one is `top` and they are not identical is undefined (i.e., the `type` problem has no solution). A point(Loc1,Loc2) problem is solved by storing `@Loc2` in location `Loc1`. A `copy(Arc,Loc)` problem is solved by adding the feature-location pair in `Arc` to the arcs of the node stored at location `Loc`. A `unify(Loc1,Loc2)` problem is solved by generating a sequence of `type`, `copy`, `unify` and `point` subtasks. Note in particular that `unify` tasks do not directly manipulate the graph themselves.

So in our particular example, we proceed as follows. We represent the state of the computation by displaying both the current state of the graph and the tasks remaining to be executed. (This corresponds to displaying the constraint base and the resolvent in tracing Prolog programs.) In the first step we are unifying two structures, so we generate a list of unification tasks for each pair of corresponding arguments, plus a point task, which will replace the first structure with a pointer to the second.

```
Graph = [f/[1-@2,2-@3],g/[1-@4,2-@5],top/[],a/[],top/[],
         f/[1-@3,2-@7],g/[1-@8,2-@9],a/[],b/[]].
Tasks = unify(@1,@6).

Tasks = type(f,f,@6),unify(@2,@3),unify(@3,@7),point(@1,@6).
```

In this example the node `@3` (which corresponds to variable `Y` in the textual form of these terms) is mentioned twice; in the first case this is because it is the first argument of term `@6` and in the second case it is because `@3` is also the second argument of term `@1`.

The second move of the game executes the `type` subtask, which leaves the graph unchanged. Then we execute our first unification subtask, `unify(@2,@3)`.

The type at @3 is top, so the result of the type task is to reset it to g. Since the structure at location @3 has no arcs, all arcs from location @2 are copied to @3, and no new unification subtasks are generated.

```
Tasks = unify(@2,@3),unify(@3,@7),point(@1,@6).

Tasks = type(g,top,@3),copy(1-@4,@3),copy(2-@5,@3),
        point(@2,@3),unify(@3,@7),point(@1,@6).

Graph = [f/[1-@2,2-@3],@3,g/[1-@4,2-@5],a/[],top/[],
         f/[1-@3,2-@7],g/[1-@8,2-@9],a/[],b/[]].
Tasks = unify(@3,@7),point(@1,@6).
```

The execution of the four tasks generated by unify(@2,@3) has the effect that first argument (@2) is replaced with a pointer to the second (@3) and the solution is placed in position @3.

The next task is another unify task. Note how the value at location @3 has changed, and how this corresponds to the binding of variable Y. Now the unify task involves two structures with nonempty adjacency lists, so we generate the usual set of type, unify and point tasks.

```
Tasks = type(g,g,@7),unify(@4,@8),unify(@5,@9),point(@3,@7),
        point(@1,@6).
```

The execution of the type task leaves the graph unchanged. The next step is again the unification of two structures, so we generate some more subtasks.

```
Tasks = unify(@4,@8), unify(@5,@9),point(@3,@7),point(@1,@6).

Tasks = type(a,a,@8),point(@4,@8),
        unify(@5,@9),point(@3,@7),point(@1,@6).
```

Notice that in this setting the unification of two “constants” is just like the unification of two structures or of an unbound variable and a structure. That is, we replace the occurrence of a/[] in @4 with a pointer @8. Since the unificands have no arguments, no unify subproblems are generated.[5]

```
Graph = [f/[1-@2,2-@3],@3,g/[1-@4,2-@5],@8,top/[],
         f/[1-@3,2-@7],g/[1-@8,2-@9],a/[],b/[]].
Tasks = unify(@5,@9),point(@3,@7),point(@1,@6).
```

The final unification step unifies top/[] at location @5 with b/[] at location @9.

```
Tasks = type(top,b,@9),point(@5,@9),point(@3,@7),point(@1,@6).
```

[5]Prolog itself behaves differently. Like variables, constants appear only once; constant names show up as addresses in a *symbol table*. We cannot do that here, since in the case of feature structures a/[] and a/[1-@1] do unify. Since we might unify different occurrences of a zero-ary feature structure with different other structures we cannot assume that every mention of a/[] refers to the same object.

No unification subproblems are generated, and we have no old unification subproblems still on our agenda, so we have only to update the graph and we are done.

```
Graph = [@6,@3,@7,@8,@9,
         f/[1-@3,2-@7],g/[1-@8,2-@9],a/[],b/[]].
Tasks = (empty).
```

To check our work we need to verify that locations `@1` and `@6` have been made identical and that that structure is indeed the most general unifier for the original terms. The first part is trivial: since `@1` points to `@6` we have certainly made the two structures identical. What have we built? Well, if we were to print out whatever structure is rooted in position `@6`, we would start by printing an `f` and an open parenthesis, then printing out whatever structure is to be found at position `@3`, printing out a comma, printing out whatever is to be found at position `@7`, and printing out a final closing parenthesis.

```
f( print(@3) , print(@7) )
```

Looking at position `@3` we find it is a pointer to `@7`, so we have to print whatever is there. Following our printing "algorithm", we would now have the "printing problem":

```
f( g( print(@8) , print(@9) ) , print(@7) )
```

The rest you can work out for yourselves! Clearly, we end up with the term `f(g(a,b),g(a,b))`, which is indeed the m.g.u. of `f(g(a,X),Y)` and `f(Y,g(a,b))`.

The situation is not much more complex for feature structures which do not correspond to Prolog terms. The only real difference is that now unifying two structures can result in the generation of `copy` problems. (In the Prolog case this only happens when we unify a structure on the left with an unbound variable on the right.) In particular, if we find that some feature `foo` appears in the first unificand and not the other, then we must simply copy the arc `foo-@wherever` to the structure we are building. So we are now in a position to extract a (syntactically and semantically) correct Prolog program from this informal description of how it should run. (A little more formally, and looking ahead to Part III, we are constructing a program by first constructing its proof trees, or, at least, the proof trees we think we want from it.)

### 7.5.4 The `unif.pl` Program

#### The Main Routine

As promised the main routine is essentially a simultaneous depth first search of two (sub-) graphs. The top-level predicate `unify(P,Q,G0,G)` takes two pointers (i.e., node names) `P` and `Q` which point to the roots of the unificands in the graph `G0`. Output is the modified graph `G`, in which both `P` and `Q` point to the root of their most general unifier.

```
*/
:- op(200,fx,@).

unify(P,Q,Graph0,Graph):-
    get(P,Graph0,PVal),
    get(Q,Graph0,QVal),
    unif(PVal,QVal,P,Q,Graph0,Graph).
/*
```

First unify/4 calls get/3 twice to fetch whatever is stored in locations P and Q from Graph0. Then control passes to unif/6, which handles the various different cases, depending on what we find in PVal and QVal.

There are two kinds of expression that can be stored as nodes in the graph—terms and pointers—and there are two subgraphs to consider so there are four possible cases for unif/6 to handle. In three of these cases at least one of the unificands is a pointer expression. In that case we simply get whatever it points to and continue in this wise until both arguments are terms. We keep track of which locations we are examining in the fourth and fifth argument positions. When we finally get two terms as unificands, we first compute the type of their unifier (if such a type can be found), then we call unify/4 recursively on each arc in the adjacency record of the first term (FS1). When the recursion on P's and Q's neighbors is complete, we take the graph as it is output from recursive_unify/6 (G2) and update it so that P now points to Q.

```
*/
unif(Typ/FS,@Q,P,_,G0,G):-
        get(Q,G0,QVal),
        unif(Typ/FS,QVal,P,Q,G0,G).
unif(@P,Typ/FS,_,Q,G0,G):-
        get(P,G0,PVal),
        unif(PVal,Typ/FS,P,Q,G0,G).
unif(@P,@Q,_,_,G0,G):-
        get(P,G0,PVal),
        get(Q,G0,QVal),
        unif(PVal,QVal,P,Q,G0,G).

unif(Typ1/FS1,Typ2/FS2,P,Q,G0,G):-
        type_inference(Typ1,Typ2,Q,G0,G1),
        recursive_unify(FS1,FS2,Q,G1,G2),
        point(P,Q,G2,G).
/*
```

In general there are three cases to consider when we are comparing the arcs of two feature structures. A given feature may appear in both unificands, only in the first one, or only in the second. However, since we are storing the result in the location of the second unificand, we can ignore the third case: all arcs belonging only to the second unificand are already correctly represented. In case a given feature F has values in both feature structures (clause 2), then we must unify those values. (This is where this algorithm takes on the appearance of a depth first traversal. Remember that unify/4 will ultimately call recursive_unify/6, giving us the familiar double recursion, first on the neighbors of V1 and then afterwards on its "siblings".) In case F is not present in FVs2, then we must copy F-V to FVs2, which is stored in the graph at location Loc, but we do nothing further with the structure at location V.

```
*/
recursive_unify([],_FVs,_Loc,G,G).
recursive_unify([F-V1|FVs1],FVs2,Loc,G0,G):-
    select(F-V2,FVs2,FVs2a),
    unify(V1,V2,G0,G1),
    recursive_unify(FVs1,FVs2a,Loc,G1,G).
recursive_unify([F-V|FVs1],FVs2,Loc,G0,G):-
    nonmember(feature,F,FVs2),
    copy(F-V,Loc,G0,G1),
    recursive_unify(FVs1,FVs2,Loc,G1,G).
/*
```

### Utility Predicates

Most of the remaining predicates are straightforward extensions of nth/3. The only significant exception is type_inference/5 which must compute the type of the new structure. This is done in the following way: the type of the unifier is defined to be the most specific possible type which is at least as general as the types of the unificands. Formally, we conceive of the type hierarchy as a lattice, ordered by the partial order "more general than". In such a setting, we are looking for the *greatest lower bound* of the two input types. Details of this calculation are found in the Test Data section. Like the list of particular features that are allowed, and the list of possible types, the type hierarchy can be freely varied according to the problem at hand, without disturbing the core algorithm.

```
*/
get(Ptr,Graph,Value):- nth(Ptr,Graph,Value).

point(P,Q,G0,G):- replace_nth(P,@Q,G0,G).

copy(Arc,Loc,G0,G):-
        add_arc_at_nth(Loc,Arc,G0,G).

type_inference(T1,T2,Loc,G0,G):-
        glb(T1,T2,T), % defined in test data section
        replace_type_nth(Loc,T,G0,G).
```

```
member(X,[X|_]).
member(X,[_|L]):- member(X,L).

nth(1,[X|_],X).
nth(K,[_|L],X):- K>1, J is K-1, nth(J,L,X).

replace_nth(1,X,[_|L],[X|L]).
replace_nth(K,X,[Y|L1],[Y|L2]):-
        K>1, J is K-1,
        replace_nth(J,X,L1,L2).

replace_type_nth(1,T,[_/FVs|G],[T/FVs|G]).
replace_type_nth(K,T,[X|G1],[X|G2]):-
        K>1, J is K-1,
        replace_type_nth(J,T,G1,G2).

add_arc_at_nth(1,F-V,[T/Arcs|Rest],[T/[F-V|Arcs]|Rest]).
add_arc_at_nth(K,F-V,[X|G1],[X|G2]):-
        K>1, J is K-1,
        add_arc_at_nth(J,F-V,G1,G2).

select(X,[X|L],L).
select(X,[Y|L1],[Y|L2]):- select(X,L1,L2).

nonmember(_Type,_,[]).
nonmember(Type,X,[Y|L]):-
        neq(Type,X,Y),
        nonmember(Type,X,L).

neq(feature,F1,F2-_):-
        features(Feats), % defined in test data section
        domain_neq(F1,F2,Feats).

domain_neq(X,Y,[X|Dom]):- member(Y,Dom).
domain_neq(X,Y,[Y|Dom]):- member(X,Dom).
domain_neq(X,Y,[_|Dom]):- domain_neq(X,Y,Dom).

/*
```

**Test Data**

This section contains code which is not considered part of the core program. In particular, besides test data, it includes the definitions of `features/1` and `glb/3`.

Greatest lower bounds in our type hierarchy are simple to compute. If one of the types is `top` then the other one is our solution. If neither one is `top`, then they must be identical. Otherwise no solution exists. A little informally, we are saying that the most specific type which is both a T and a top is T itself. (T in this case can of course also be top.) If T1 and T2 are different, and neither is top, then there are no types that are both T1 and T2 (so a fortiori there is no most specific such type), so the computation fails in this case.

```
*/
glb(top,T2,T2).
glb(T1,top,T1):-
        definite(T1).
glb(T,T,T):- definite(T).

definite(T):- member(T,[f,g,h]).

features([1,2,3,4,a,b,c,d]).


test(1):- test(1,1,5,X), nl, print(X), nl.
test(2):- test(2,1,3,X), nl, print(X), nl.

test(Graph,FS1,FS2,Result):-
    data(Graph,G0),
    unify(FS1,FS2,G0,Result).

data(1, % X=h/[],Y=h/[],f[a-g[a-X],c-g[a-X]]=f[a-g[a-Y],b-g[a-Y]].
        [f/[a-2,c-3], %1
         g/[a-4],
         g/[a-4],
         h/[],
         f/[a-6,b-7], %5
         g/[a-8],
         g/[a-8],
         h/[]
        ]).
data(2, % f[a-X,b-X]=f[a-g[b-Y,c-Z]].
        [f/[a-2,b-2],
         top/[],
         f/[a-4],
         g/[b-5,c-6],
         top/[],
         top/[]
         ]).

/*
```

**Running the test data**

```
porcupine:courses/97> sicstus
SICStus 2.1 #8: Tue Jul 25 14:59:57 MET DST 1995
| ?- ['unif.tex'].
{consulting /home/abacus/cornell/courses/97/unif.tex...}
{/home/abacus/cornell/courses/97/unif.tex consulted, 130 msec 19088 bytes}

yes
| ?- test(1,1,5,X).

X = [@5,@6,g/[a-4],@8,f/[c-3,a-6,b-7],g/[a-8],g/[a-8],h/[]] ? ;

no
| ?- test(2,1,3,X).

X = [@3,@4,f/[b-2,a-4],g/[b-5,c-6],top/[],top/[]] ? ;

no
| ?- halt.

porcupine:courses/97>
```

Note that in both cases the computed solution can be found by starting at the first member of the graph. So, in the first case, we start at location 1 which contains the pointer @5, so we first go to location 5, and continue from there.

*/

# Part III

# Control

Up to now, we have focused on learning how to define relations in Prolog in ways which are *correct.* Often we find that there are many different equally correct ways of defining a relation. So what's the difference? In this part of the course we will begin to see ways to answer that question based on the more particular question "which version performs better?" Note that the answers here won't always be clear cut. We will always have to balance simplicity and readability of code against raw speed, and speed against space usage. Nonetheless, in very many cases we will be able to make substantial improvements in the performance of the predicates we have designed so far.

Prolog *programs* are nondeterministic, but the Prolog *interpreter* which runs them runs on a deterministic computing machine, most often with a single central processing unit. Therefore there are a great deal of factors which influence the behavior of a Prolog program which go far beyond the essential question "Is the goal in fact a consequence of the program?"

What does "nondeterminism" mean, anyway? We saw, while we were considering graph-traversal problems, that the list of nodes we had yet to (attempt to) visit could be viewed as a sort of *agenda*, a list of *tasks* which had to be attempted. We saw various ways of ordering these "tasks": putting the new ones at the top of the agenda and at the bottom were the most obvious choices, but we also observed that we had considerably more flexibility here. We considered, briefly, the following general schema:

```
choose( Task, Agenda, Remainder ) :-
     select( Task, Agenda, Remainder ),
     okay( Task ).
```

This code can be read aloud as "Nondeterministically `select` a `Task` from the `Agenda` which is `okay`."

Mathematically, this just means "Select, as if by magic, a task which satisfies the relevant conditions." It is generally believed (though it has not yet been proven) that this magic cannot in fact happen. But we could get very close to it if we imagined assigning to a collection of processing units the task of finding an okay Task, in *parallel.* Then we would indeed find a correct solution in the same amount of time in which the magic machine would have found it, though with considerably more silicon devoted to the job. So now we have two ways of thinking about nondeterminism: as magic, and as a (massively) parallel simulation of magic.

Of course, very few actual computing machines have enough processors to implement this behavior directly. Instead they *simulate* the behavior of parallel machines, usually by a search through the tree of possible computations, using backtracking. And so we arrive at the third interpretation of "nondeterminism": as *(backtracking) search.* And this is the interpretation that makes the most sense for, say, SICStus Prolog, running on a Sun workstation. So when we describe the behavior of `choose/3` as "Nondeterministically `select` a `Task` from the `Agenda` which is `okay`," we *mean* something more like "`Select` the next `Task` from `Agenda`; if `okay` succeeds, then we are done; if `okay` fails, then backtrack into `select` and pick another `Task`; if we run out of `Task`s, then fail." So, when we finally find a choice that works, the program reaches the same conclusion as the magic machine would have. In fact if we just look at the proof tree, that is all we see; we see only the successes and ignore the failures. But in reality the Prolog interpreter may have had to do a considerable amount of *searching* for just the right combinations of choices.

So we don't have magic. We don't even have parallelism. The best we can do is simulate a parallel computer simulating magic. By searching, one at a time, all of the possible combinations of choices that the program and the goal have left open. Because our computing machines are so far removed from our abstract *model of computation*, there are many decisions that have to be made in actually interpreting a Prolog program/goal combination which are not reflected in the abstract model (magic, in this case).

There are two points in particular at which Prolog must commit to a choice *deterministically* where the magic machine (or even the massively parallel model) might not. First, we consider the current goal as an agenda: each literal in the goal corresponds to a computational task which must be executed if the goal is to be proven. One would like to be able to nondeterministically choose the literal which would "advance the computation the most," which means something like, a literal with only one possible proof (so, for example, a literal with most or all of its arguments ground is a good bet here), and which has the effect of binding variables in other literals, making them ready in turn. But what Prolog does is just to blindly select *the leftmost literal* in the goal. So one way we can optimize the performance of our Prolog programs is to find some way to modify our code so that the leftmost literal in the goal is in fact "ready" and "helpful".

Having selected a literal for processing, Prolog now considers yet another sort of list: it must select (again, "nondeterministically") a program clause to resolve the selected literal against. The program is not really like an agenda: it specifies various different ways of solving goals. It behaves as a big *disjunction*, whereas a true agenda is essentially *conjunctive*: it says "do this *and* do that ... " whereas a program says "you can do it this way *or* you can do it that way." But the computational features of the choice that must be made remain. Prolog must *nondeterministically* select a program clause whose head matches the selected literal (and which, to get the most out of our magic, should also ultimately lead to a solution). Of course, Prolog cannot know that it has tried all the possibilities without actually trying all the possibilities, and on a sequential machine it must do so sequentially, so some program clause must be picked first. Prolog alwyas picks *the first clause first.* So, once again, we have an opportunity for optimization: we want to make sure that the first clause in the definition of a predicate is the one with the best chance of leading to a short proof, i.e., a fast solution. It is for this reason that I have (so far without comment) always ordered the base clause(s) of a recursive predicate before the step clause(s).

Because Prolog selects the leftmost literal for processing, and because it replaces the literal with the body of the matching clause, i.e., the body of the clause becomes the leftmost literals of the new clause, the first goal in the body of a clause is the one which will be the next selected literal if that clause is selected from the program by the resolution rule. So in general we want to order goals in the body of a clause in such a way that the earliest goals are (a) cheap to compute and (b) have good constraining power over the remaining goals. For example, as you may have noticed, I always try to order the recursive calls in a predicate definition last, if possible, on the assumption that the nonrecursive (or recursive-on-a-smaller-structure) goals will run faster. Similarly, I always try to order *guards* first, since if they fail, there will be no need to perform any other computations in the rest of that clause-body. This also is should convince you to make your guards cheap to compute. Simple comparisons ($<$, $>$) are useful here, or non-recursive type-checking, as for instance in a literal like `constant(X)`.

However, this is only the tip of the iceberg, so to speak, and there are many more subtle arguments that can be made for why a predicate should be defined one way and not another. There are also facilities which Prolog provides which go beyond logic and which can, if used carelessly, destroy the semantics of a program but which can also, if used carefully, vastly improve the performance of a program without destroying its logic. Here we will study these as well, as we develop the idea of *Logic Programming as Controlled Deduction*.

# Chapter 8

# Differences

In this lecture, we will explore the notion of *differences*, which gives us a declarative semantics for various data structures based on the *accumulator pair* techniques we saw in the section on graphs. This *semantic* notion leads us to two new types of data structure: difference lists and difference structures. We will then see how we can, by careful use of appropriate data structures, minimize the cost of our programs' structure building operations. For example, we will find ourselves in the position of being able to do away with many calls to `append/3`, being able to concatenate sequences now in essentially constant time.

## 8.1 Length of the shortest proof as a measure of efficiency

Though we are interested in particular in the topic of differences, the topics we choose for this part of the course are chosen because they allow us also to explore the use of various techniques for analyzing and improving the performance of (otherwise logically correct) Prolog programs. We examine first a case where a simple comparison of *proof trees* suffices to establish significant differences in computational behavior. Proof trees are not often as useful a tool as search trees for optimizing Prolog programs, because they represent only one solution and not the search that went into finding that solution, nor any alternative solutions that the program might allow, but when they are useful they are very useful indeed. Because they illustrate only the processing which is *necessary* and *unavoidable* for establishing a certain goal, significant differences in the size of the proof trees for different implementations of a predicate are, well, a little extra significant.

The following code implements a simple predicate for reversing the order of elements in a list.

```
reverse( [], []).
reverse( [Head|Tail], Reversed ) :-
```

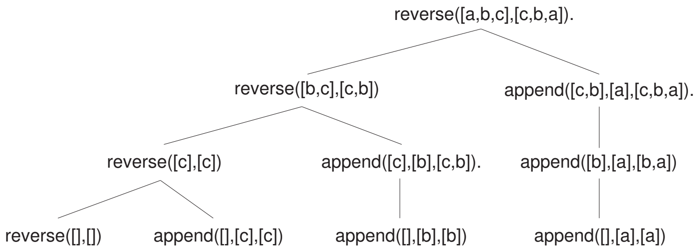



Figure 8.1: Proof tree for “naïve” reverse.

```
          reverse( Tail, RevTail ),
          append( RevTail, [Head], Reversed ).
```

A proof tree for the goal

```
?- reverse( [a,b,c], [c,b,a] ).
```

is given in figure 8.1.

We can see that many aspects of the geometry of this tree can be computed as a function of the size of the input list. For example, all of the (recursive) calls to `reverse/2` itself appear on the left branch of the tree. Clearly there will be exactly as many of these as there are elements in the input list, plus one for the empty list, when that is finally reached. Then, for each recursive call to `reverse/2` there will be a call to `append/3`. The proof trees for these `append/3` goals are nice and linear; they do not spread out into masses of new tasks but go straight for their goal. But there is still an awful lot of them! And each call to `append/3` (from right to left) takes a slightly longer list as input. Even worse, each call to `append/3` considers many of the same elements that previous calls had to work on, before it finally gets to new information about the output list. For example, the elment `c` is part of all but the very first `append/3`-input, and the element `b` is part of all but the very first and second lists.

Thus we can see that the code for `reverse/2` presented above, though it is very logical, is not at all efficient. The question we must ask ourselves as logic *programmers* is: can we make it more efficient? And the question we must ask ourselves as *logic* programmers is: can we do it without sacrificing logic? With this in mind consider the code for `reverse/2` and `reverse/3` presented below.

```
reverse( List, ReversedList ) :-
     reverse( List, [], ReversedList ).

reverse( [], Answer, Answer ).
reverse( [Head|Tail], AnswerSoFar, UltimateAnswer ) :-
     reverse( Tail, [Head|AnswerSoFar], UltimateAnswer ).
```

A proof tree for the same goal as before is presented in figure 8.2. The reduction in the size of the search tree is quite striking: it doesn't even branch

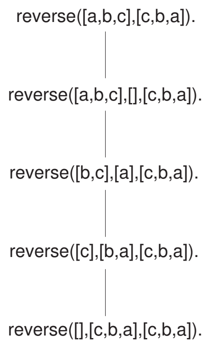



Figure 8.2: Proof tree for the modified reverse predicate.

at all, now! There are exactly as many nodes in the tree as there are elements in the input list, plus the one for the empty list. We have completely eliminated all those calls to `append/3`!

The logic underlying `reverse/3` is as follows. When we are looking at the first element of the input list, we are simultaneously looking at the last element of the reversed input list, and when we are looking at the second element of the input list, we are by that very fact also looking at the second-to-last element of the output list, and when we are looking at the third element of the input list, we are looking at the third-to-last element of the output list. The sole element of a one-element list is at once both its first and its last element. The second-to-last element of a two element list is at the same time its first element, and the third-to-last element of a three element list is at the same time *its* first element. As a rule, we have a very hard time inserting things in (or doing anything at all to) positions near the end of a list: we have to traverse the entire length of the list to find the position we're looking for. (That's what all those calls to `append/3` were for in naïve reverse.) But doing things to the *first* element of a list is trivially simple. As you can see, if we build up the output list starting with its last member, we will always be operating on its first element, instead of always operating on its last element, which is what we did in the "naïve" version of `reverse/2`.

The solution here is very like the approach we used in finding paths in a graph. Previously, we always constructed the output data structure *on the way up* out of the recursion. That is, given some recursive predicate `p`, the output of a call to `p` would be a function of the output of the embedded recursive call to `p`. In the definition of `reverse/3`, as in the definition of `path/4`, the situation is reversed. We construct the output data structure *on the way down* into the recursion. That is, given a recursive procedure `p` which calls itself, the value of the output variable in the *recursive* instance of `p` is a function of the value of the output variable in the *calling* instance of `p`—just the reverse of the situation we are used to.

That there is a big difference in doing things this way can be seen clearly in the base clauses for the naïve `reverse/2` and `reverse/3`. In the original code, the base clause initialized the output variable to the empty list. In the new code the base clause binds the "true" output variable (the third argument) to the "accumulator" variable with which we have been constructing the output up to that point. (Note that all of the output variable positions in all of the nested calls to `reverse/3` have been identified with each other during the descent into the recursion, so that as soon as the reversed list is passed to the output argument of the base clause it is bound to the output argument of the highest-level goal clause. In a sense, the output variable argument position just maintains a direct open channel between the eventual base of the recursion and the original calling goal.)

The pair of variables `AnswerSoFar` and `UltimateAnswer` is called an *accumulator pair*. The use of accumulator pairs and other closely related structures is extremely common in logic programming, and forms the basis of a number of very powerful techniques. However, the logic is correspondingly subtle. It is very easy to think of the base clause of naïve `reverse/2` as making sense on its own: the result of reversing the empty list is the empty list. At the moment the base clause of `reverse/3` doesn't make much sense except in the context of accumulator-pair processing. It is not so clear how to provide it with a nice declarative reading. The reading "when you reach the end of the input list, pass the structure in the accumulator over to the output variable," sounds rather procedural. Have we sacrificed logical clarity to gain efficiency? The (negative!) answer to this concern is found in the notion of *differences*.

## 8.2 Difference Lists

We have already seen that one very useful way to represent a sequence of data items is with a list. This technique has proven so useful that we have tended to identify the notion of *sequence* (a semantical notion) with the notion of *list* (a data structure). Here we will need to pry these notions apart again, and consider other ways to represent sequences. In particular we will now investigate representing sequences very abstractly as the *difference* between two lists. For example, the sequence $\langle a, b, c\rangle$ can be represented as the list `[a,b,c,d,e]` *minus the list* `[d,e]`.

Admittedly, this seems like a rather roundabout way to represent a sequence, but it forms the basis of some of the most powerful techniques in logic programming. In the first place, consider how we would represent the empty sequence. In a language with subtraction, zero can be represented as $x - x$, for any $x$. Similarly, the empty sequence is represented as the difference between `L` and `L`, for any list `L`. So, to summarize, `[a,b,c]` minus `[c]` represents the same thing as the list `[a,b]`, and `[a,b,c]` minus `[b,c]` represents the same thing as the list `[a]`, and finally `[a,b,c]` minus `[a,b,c]` represents the same thing as the list `[]`.

This gives us the purely declarative semantics that was missing from the definition of `reverse/3`. In particular, the base clause of `reverse/3` means exactly the same thing as the base clause of `reverse/2`, if we take the third argument minus the second argument to represent the same sequence as the second argument of `reverse/2`. The same holds for all the instances of the recursive clause as well. For example, as the first call to `reverse/3` in figure 8.2 we have

```
?- reverse( [a,b,c], [], [c,b,a] ).
```

Well, clearly `[c,b,a]` minus the empty list is just `[c,b,a]` again, so the intended semantics holds here: the sequence represented by the difference between the third and second arguments is the result of reversing the sequence represented (as a standard list) by the first argument. Similarly, we have as the second call to `reverse/3` the goal

```
?- reverse( [b,c], [a], [c,b,a] ).
```

And indeed the result of reversing the sequence denoted by `[b,c]` is the sequence denoted by (`[c,b,a]` minus `[a]`). And so on.

Now consider the most general form of a difference list. In order to represent the same sequence as is represented by the "standard" list `[a,b]` we could employ the difference-pairs (`[a,b,c]`,`[c]`) or (`[a,b,d]`,`[d]`) or (`[a,b,c,d]`,`[c,d]`) or any of an infinite number of other possibilities, but they all have the form:

```
([a,b|X], X).
```

This is the most general difference list representation of the sequence $\langle a, b\rangle$. Now consider this same observation from a slightly different angle. We already know that we can use `append/3` to compute the difference between a pair of lists.

```
difference(L1,L2,L3) :- append(L3,L2,L1).
```

That is,

```
difference(L1,L2,[a,b]) ⇐
append([a,b],L2,L1) ⇐
L1 = [a,b|L2].
```

So the most general difference *list* is seen to be the solution to the most general difference *problem*.

Now, we have seen that we can use `append/3` to compute list-differences. What is really wonderful is that we can use difference lists to compute concatenations. What is more wonderful is that we can use difference lists to compute concatenations *in constant time!* Consider the following code.

```
dl_append( L1Front, L1Back, L2Front, L2Back, AnsFront, AnsBack
     ) :-
     L1Front = AnsFront,
     L1Back = L2Front,
     L2Back = AnsBack.
```

This assumes that the input is given as difference lists, so there are twice as many arguments as there are to plain old everyday `append/3`. But given this circumstance, we have only three unifications to perform and we are done, *regardless of the size of the input lists.* For example,

```
?- dl_append([a,b,c|X],X,[d,e|Y],Y,AnsFront,AnsBack).
```

reduces to

```
?- [a,b,c|X] = AnsFront, X = [d,e|Y], Y = AnsBack.
```

The next few steps give us the following sequence.

```
(AnsFront = [a,b,c|X])
?- X = [d,e|Y], Y = AnsBack.
(AnsFront = [a,b,c,d,e|Y])
?- Y = AnsBack.
(Answer = ([a,b,c,d,e|Y], Y))
```

Difference lists are very useful when you need access to both ends of a sequence. The prototypical example of this is the data structure known as *queues.* Queues are so called "First In First Out" (FIFO) data structures, as opposed to *stacks*, which are "Last In First Out" (LIFO) data structures. Elements are added at one end of the queue and removed at the other, rather like people in the queue at the ticket booth of a movie theater, or like goods on a conveyer belt.

For the moment we will implement queues in Prolog directly as difference lists. More robust implementations, such as one can find in the libraries of most major Prolog systems, or in the Edinburgh public library, are slightly more complex, but still built on these ideas. We will stipulate that the front of the queue, from which we remove items, will be the head of the difference list, and we will add items to the tail.

```
dequeue( Item, OldQFront, OldQBack, NewQFront, NewQBack ) :-
      OldQFront = [Item|NewQFront],
      OldQBack = NewQBack.

enqueue( Item, OldQFront, OldQBack, NewQFront, NewQBack ) :-
      OldQBack = [Item|NewQBack],
      OldQFront = NewQFront.
```

The code for `dequeue/5` is straightforward: we remove the `Item` from the head of `OldQFront`, and the back of the queue remains unaffected. The code for `enqueue/5` is harder to grasp, since that is where we crucially rely on the magic of difference lists. An example will help to clarify things. Suppose we have the following goal.

```
?- enqueue( addme, [a,b,c|OldBack], OldBack, NewFront, NewBack
      ).
```

First we unify the original back of the queue, `OldBack`, with the list `[addme|NewBack]`.

```
OldBack = [addme|NewBack] ⇒
OldFront = [a,b,c,addme|NewBack]
```

Now we unify the old front and the new front (we have after all done nothing to the front of the queue—that is where we remove things from, not where we add things), yielding the resulting queue:

```
[a,b,c,addme|NewBack], NewBack.
```

The only processing we have done is a few unifications; in particular, no searching of the queue was necessary to find its tail end. In effect the second member of the difference list acts very much as a pointer to the tail of the queue.

There are some problems with difference lists, however, that you must remain aware of. First, and less serious, is that in order to convert a standard list to a difference list, we still have to traverse the entire input list.

```
list2dl( List, Front, Back ) :-
      append( List, Back, Front ).
```

So it would appear that we have done away with calls to `append/3` in one place only to introduce them again in another. However, it is rarely the case that such conversions need to be done. More often one represents the relevant sequence with a difference list right from the start.

A more serious problem is actually a serious problem with Prolog, not so much with difference lists themselves. You may recall that when we defined unification we were careful to rule out cases like the following, where a variable is unified with a term which contains an instance of that variable.

```
?- X = f(X).
```

At the time we remarked that this "occurs check", which detects ununifiable terms like these, was left out of Prolog for efficiency reasons. Up until now we have had no need to worry about "occurs-check violations," but that situation is about to change.

Suppose we want to write a predicate which needs to recognize instances of the empty difference list. We have needed to do this over and over with standard lists, for example in the base clauses of recursions on list structures. So we might reasonably expect that at some point we will have to do this with difference lists.

```
generic_base_clause( L, L, OtherArguments ).
```

In this example, we have defined the empty difference list as the identical pair of lists `(L, L)`, as we should. But notice what happens when we issue the goal:

```
?- generic_base_clause( [a,b|X], X, _ ).
```

Prolog will attempt to unify `[a,b|X]` with `X` and, since `X` is a variable, and unification always succeeds for variables, *the unification will succeed.* Of course, it should not. First, the difference list `([a,b|X], X)` is not empty and second this unification violates the (missing) occurs check. If the occurs check were in place, we would have no trouble here: the unification would block, as it should, until the first argument really was the end of the difference list (i.e., a variable unifiable with `X`). But since the occurs check is missing, we have as yet no way to prevent Prolog from executing what will result in a circular variable binding, a disastrous result!

There are various ways around this problem, but most of them require Prolog facilities that go beyond the logical core of the language. These will be dealt with in the chapter on Meta-Logical Predicates. For now, since we have no immediate need for recursion on difference lists, the reader need only keep this potential problem in mind for the moment, and not be too hasty to abandon standard list structures!

Let us now review some examples of predicates where we can use difference lists to dramatically improve performance. The predicate `tree2list/2` is a simple preorder traversal of a tree which returns a list of all the node labels it saw, in (pre-) order.

```
tree2list( RootLabel/SubTrees, [RootLabel|NodeLabels] ) :-
     t2l_star( SubTrees, NodeLabels ).

t2l_star( [], []).
t2l_star( [Tree|Trees], L0 ) :-
     tree2list( Tree, L1 ),
     t2l_star( Trees, L2 ),
     append( L1, L2, L0 ).
```

We can do away with the call to `append/3` if we can reimplement these predicates using difference lists to represent the sequence of node-labels, instead of using standard lists.

```
tree2list( Root/Subs, [Root|Labels0], Labels ) :-
     t2l_star( Subs, Labels0, Labels ).

t2l_star( [], L, L ).
t2l_star( [Tree|Trees], L0, L ) :-
     tree2list( Tree, L0, L1 ),
     t2l_star( Trees, L1, L ).
```

The conversion to `tree2list/3` is straightforward. We want to attach the root-label to the head of the sequence returned by `t2l_star/3`; attaching something as the head of a difference list is no different than attaching it as the head of a standard list. Likewise the conversion to the base clause of `t2l_star/3` is simple; we merely replace the second empty list in the base clause of `t2l_star/2` with the difference list representation of the empty sequence, (L,L). In the recursive clause we want to append the sequences returned by tree2list/3 and the recursive call to t2l_star/3. So, following the logic presented in the definition of dl_append/6, we identify the front of the list returned by tree2list/3 with the front of the output list, and we identify the front of the list returned by t2l_star/3 with the back of the first list. Finally, the back of the second list becomes the back of the output. Essentially we are saying: let the end of the sequence returned by tree2list/3 (i.e., L1) be identified with the beginning of the sequence returned by t2l_star/3 (i.e., let it be L1 also). We can do this with difference lists because the end of the first sequence is a variable—so of course we can identify it with any list we want—but wih standard lists the end of the sequence is a constant ([]). We require two data structures to represent the difference list because we need one instance of this tail-variable at the end of the sequence we want to describe, which is a position we cannot otherwise get at easily, and we need another instance of this variable in a position which we can get at easily. Then binding the instance of the variable we can get at easily (i.e., the second member of the difference pair) will immediately bind the variable at the tail of the list, which we could not have otherwise gotten to without recursively traversing the entire list.

Sometimes "optimizations" give us something more than just gains in speed and memory use. The tree2list/3 predicate is a good example of this. Its near relation, tree2list/2 only works properly in one mode, namely when the tree is instantiated and we solve for the list of nodes it contains. If we try to use the nonstandard mode in which we give it a list and try to solve for a tree, we find that the search tree contains an infinite branch. For example, if we issue the goal

```
?- tree2list( Tree, [a,b,c] ).
```

we get the single solution

```
Tree = a/[b/[],c/[]]
```

but if we press the ';' key, requesting more solutions, we fall into an infinite search, and never find the other solutions to this goal. Note that we can get tree2list/2 to work properly in this mode by moving the call to append/3 before the calls to tree2list/2 and t2l_star/2 in the recursive clause of t2l_star/2. However, if we do that we introduce an infinite branch into the *standard* mode of tree2list/2. We can't win! But the version with difference-lists avoids the problem of where to put the call to append/3 by avoiding the call to append/3 entirely. Hence it works properly in both modes. The moral of this story holds true for most optimizations. They work by re-shaping the search tree, pruning away useless or redundant subtrees. So they can also serve to re-shape the search tree in such a way as to prune away subtrees with infinite branches, giving better termination properties for Prolog implementations of logic programs.

This technique of using differences generalizes to structures other than lists. Anytime we have a data structure which contains deeply embedded positions which we might want to "fill in" later, we can create a *difference structure* in which we place variables in those positions, and keep in an accessible place copies of those variables.

We consider first a very simple example. We know already how much successor-term representations of the natural numbers resemble lists. So it will come as no surprise to find that the corresponding difference structures ("difference successor-terms"?) resemble difference lists. Consider the predicate `size/2` which follows, and which is designed to count the number of nodes in a tree.

```
size( R/Subs, s(N) ) :-
      forest_size( Subs, N ).

forest_size( [], 0 ).
forest_size( [T|Ts], N ) :-
      size( T, N1 ),
      forest_size( Ts, N2 ),
      plus( N1, N2, N ).
```

If this looks a lot like `tree2list/2` to you, it should! There every time we visited a node, we added its label to a list; here every time we visit a node we add an 's' to the successor term we are building up. In both cases, we have to visit a list of subtrees and somehow 'glue together' the information we uncover in those visits.

This predicate has the same structure as `tree2list/2`, and it has the same problems as well. We cannot use it to generate all the trees of a given size, because the search trees for such goals contain infinite branches. If we move the call to `plus/3`, we can get termination in this extra mode, but we lose termination in the standard mode. And once again the same solution is available to us. We merely employ "difference successor-terms" and eliminate the call to `plus/3` entirely in a manner exactly analogous to the elimination of the call to `append/3` in `tree2list/2`.

```
size( Tree, N ) :-
      size_( Tree, N, 0 ).

size_( R/Subs, s(N0), N ) :-
      forest_size( Subs, N0, N ).

forest_size( [], N, N ).
forest_size( [T|Ts], N0, N ) :-
      size_( T, N0, N1 ),
      forest_size( Ts, N1, N ).
```

As you can see from the code, the difference structure corresponding to successor terms is just a pair of expressions of the form

```
Front = s( ... s( X ) ... ),
        Back = X
```

This is a particularly simple form of difference structure which, like a difference list, has only a single 'slot' or extracted variable. So the representation of zero in the base clause for `forest_size/3` is the pair `N, N`, just as the empty sequence is `L, L` for difference lists. And we can add difference successor terms just like we append difference lists, by making some simple unifications: set the front of the answer to be the front of the first addend, and the back of the answer to the back of the second added, and then unify the back of the first addend with the front of the second. If

```
N0 = s(s(N1)),
```

and

```
N2 = s(s(s(N)))
```

and we insist that

```
N1 = N2
```

then the difference pair `N0-N` equals

```
s(s(s(s(s(N))))) - N
```

as we would hope. This is just like saying, in arithmetic,

$$(n_0 - n_1) + (n_1 - n) = n_0 - n$$

which is certainly true, one would hope!

The following example uses both difference lists and difference structures. Here the difference structures may have more than one extracted variable.

```
s( Proposition, L0, L ) :-
     np( Subject, L0, L1 ),
     vp( [Argument], Predicate, L1, L ),
     Argument = Subject,
     Proposition = Predicate.

vp( Args, Predicate, L0, L ) :-
     verb( [Arg1|Args], Predicate, L0, L1 ),
     np( Arg1, L1, L ).
vp( Args, Predicate, L0, L ) :-
     verb( Args, Predicate, L0, L ).

verb( [Theme,Agent], chase(Theme,Agent), [chased|L], L ).
verb( [Agent], laugh(Agent), [laughed|L], L ).

np( waldo, [waldo|L], L ).
np( wilma, [wilma|L], L ).
```

```
analyze( String, Meaning ) :-
      s( Meaning, String, [] ).
```

The essential logic of this example is simple and follows the form of the corresponding context free grammar very closely. A string, represented as the difference list `(L0,L)`, is of category `s` if the first part of it is of category `np` and the second part of it is of category `vp`. The meaning of the sentence is then just the meaning of the verb phrase (a predicate with a single free argument) applied to the meaning of the noun phrase (which takes the place of its argument). Thus, if the meaning of the noun phrase is `waldo` and the meaning of the verb phrase is `chase(Agent,wilma)`, then solving `waldo = Agent` will result in the sentence-meaning `chase(waldo,wilma)`. Similarly, if the meaning of the verb phrase is `laugh(Agent)`, and the meaning of the noun phrase is `wilma`, then solving `wilma = Agent` will result in the sentence-meaning `laugh(wilma)`.

With difference structures, we can "extract" as many subterms as we like from the structure. For example, the meaning of the verb *chased* is represented here as a two-place relation from which both arguments have been extracted. The first one is bound in the VP-rule, to the direct object, and the second one is passed up to the VP node, from whence it will be bound to the subject NP in the S-rule. Also, the same extracted variable can bind several positions.

```
solve_for_x( X, X, Formula, Value ) :-
      Value is Formula.

?- solve_for_x( 3, X, (X^2 + 3*X +1), Y ).
```

In this somewhat contrived example, the number 3 will be substituted for every occurrence of X in the given formula.

# Chapter 9

# Analysis Using Search Trees

Proof trees give us an extended picture of what Prolog must compute in order to arrive at a given answer. However, they do not tell us what steps Prolog has to go through in order to discover the proof so illustrated. In particular, we can use the length of the shortest proof as a lower bound on how much time it will take for Prolog to find an answer, but we would like to know in addition (a) what things Prolog tried to do and failed before finding the correct course and (b) what other solutions are available. *Search trees* give us a more global picture of what Prolog has to do to give an answer to a query.

A proof tree is a species of *AND-tree*: for a given node to represent a satisfiable constraint all of its daughters must also represent satisfiable constraints. A search tree is a species of *OR-tree*: a given node represents a satisfiable constraint if any one of its daughters represents a satisfiable constraint. This is what makes it suitable for displaying all of the options that Prolog must consider in order to return a complete answer to a given goal. A search tree (also called an *SLD-tree*, after the form of resolution that Prolog employs) is constructed according to the following rules.

- Each node is labeled with a goal clause.
- Given a frontier node labeled $\leftarrow G$, expand it by attaching as daughters all the resolvents that can be derived from $\leftarrow G$ by one resolution step on the currently selected atom. That is:
  1. Apply the selection function to $G$ to derive the selected goal $G_i$.
  2. Find all of the program clauses $A_j \leftarrow B_{j_1}, \ldots, B_{j_k}$ such that $G_i$ unifies with $A_j$ with most general unifier $\sigma_j$.
  3. Attach as a daughter to $\leftarrow G$ each new resolvent obtained by replacing $G_i$ with $B_{j_1}, \ldots, B_{j_k}$ and applying $\sigma_j$ to the result. That is, the $j$th daughter of $\leftarrow G$ will be:

$$\leftarrow (G_1, \ldots, G_{i-1}, B_{j_1}, \ldots, B_{j_k}, G_{i+1}, \ldots, G_n)\sigma_j$$

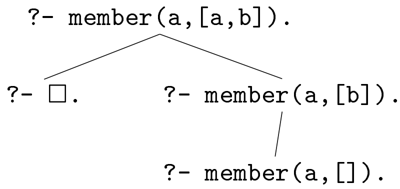



Figure 9.1: Search tree for the goal `?- member(a,[a,b]).`

- The terminal nodes of such a tree will be labeled either with the empty clause or with a clause $\leftarrow G$ for which the selected subgoal $G_i$ could not be unified with the head of any program clause. We call the former case a *success node* and the latter case a *failure node*. Clearly, there is a third possibility as well: there are non-terminating computations in Prolog, so there must also be search trees with infinite branches, that is, branches with *no* terminal node.

Recall that any given SLD resolution theorem prover has two major parameters: a *computation rule*, alias a selection function, and a *search rule*. Prolog is defined (roughly) as an SLD resolution theorem prover with a leftmost-first selection function and a first-clause-first search rule: given a predicate defined by several clauses, Prolog will always attempt to solve instances of that predicate using the first clause first. Given the definition of a search tree, you should observe the following: the search tree for a given goal is determined by the selection function, but not by the search rule. That is, if we change the selection function then the shape of the search tree will change too. However, changing the search rule only changes the order in which we visit the nodes of the search tree.

If we make the reasonable assumption that the daughter corresponding to the first matching clause is written down as the leftmost daughter in the proof tree (so in general the 'vertical' ordering of clauses in a program is reflected in a left-to-right 'horizontal' ordering of resolvents in a search tree), then we can characterize Prolog as executing a top-down (i.e., preorder), left-to-right traversal of the search tree.

Figure 9.1 represents the search tree for the goal `?- member(a,[a,b]).` It contains only two branches, one ending in a success node, the other in a failure node.

In some ways the "official" version of a search tree is still not informative enough. In the first place, it is usually helpful to see the variable bindings that take us from a parent node to a particular daughter. In the second place it is often helpful display *all* of the clauses for the currently selected subgoal, even when their heads do not unify with the selected subgoal. This is helpful when one is first learning Prolog, since it makes it clearer how the search tree relates to the program code; but it is also helpful in more advanced approaches to logic programming, where the notions of "head of a clause" and of unification are

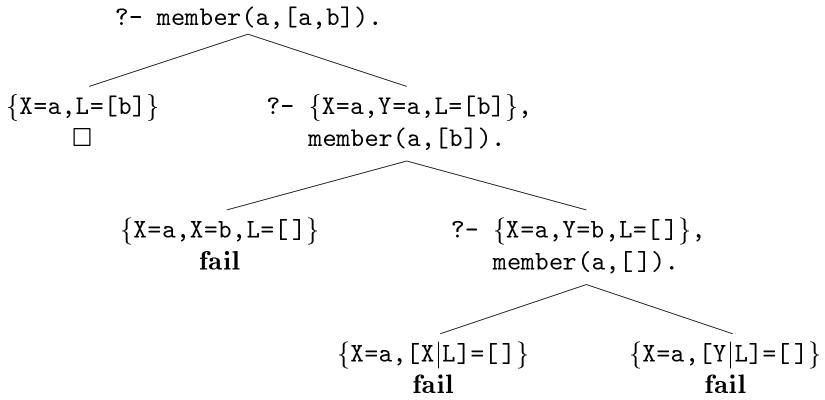



Figure 9.2: Search tree for `?- member(a,[a,b]).`, augmented with explicit displays of the unification problems.

generalized to allow clause-selection based on constraints other than *Term1* = *Term2*. In these "augmented" search trees we include the unification problems whose solution would (if a solution existed) allow search to proceed along that branch. Our previous example of figure 9.1 now looks like figure 9.2, given the following definition of `member/2`.

```
member( X, [X|L] ).
member( X, [Y|L] ) :-
      member( X, L ).
```

Once again, we have only one successful branch. The number of failed branches has increased, of course, because we are displaying all unification failures. Note that when we are using these "augmented" search trees all branches which terminate will terminate at such unification failures. Of course, displayed this way, search trees can become very large, even for small problems. Most often we will only display the variable bindings for branches where the unifications involved actually succeed, or where the variable bindings help to show which clauses were being resolved against.

The following example shows how search trees can be used to improve the performance of a Prolog program. We repeat here the code defining the data type tree, `tree/1`.

```
tree( R/[]).
tree( R/SubTrees ) :-
      forest( SubTrees ).

forest( []).
forest( [First|Rest] ) :-
      tree( First ),
      forest( Rest ).
```

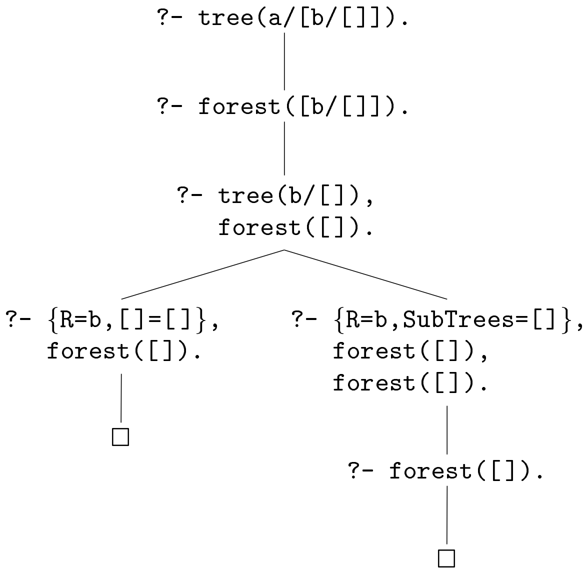



Figure 9.3: Search tree for the goal '`?- tree(a/[b/[]]).`' showing redundant search-paths.

A search tree for the rather elementary goal `tree(a/[b/[]])` is given in figure 9.3. We see immediately that there are two success branches in this tree. That is, there are two ways to prove that the data structure in question is a tree. This means that any predicate in which we embed this code will have twice as many solutions as it ought to have; or rather, every solution may be arrived at by two routes. The ambiguity of this predicate will be inherited by every program that uses it, and worse, by every predicate which generalizes this logic. Since this is a data type definition, this logic will appear in virtually every tree-processing predicate we write which must traverse an entire tree. So it is desirable to remove this redundancy.

There are actually a number of ways to do this. First, we could alter the recursive clause of `tree/1` so that `SubTrees` no longer unifies with `[]`:

```
tree( R/[Sub1|Subs] ) :-
      forest( [Sub1|Subs] ).
```

Alternatively we could redefine the first clause of `forest/1` so that the recursion bottoms out at the singleton list rather than the empty list:

```
forest( [A] ) :- tree( A ).
```

Of course now every singleton forest will unify with both `[A]` and also with `[First|Rest]`, so we would have also to modify the second clause of `forest/1`:

```
forest( [A1,A2|Rest] ) :-
```

```
        tree( A1 ),
        forest( [A2|Rest] ).
```

The most elegant solution, however, is simply to remove the first clause of `tree/1` altogether. Now a tree is any structure with major functor '/' and whose second argument is a forest of subtrees, possibly empty. Nothing else needs to be altered.

The only problem with this solution is that any predicate which involves tree traversal but which also requires special processing of the leaves of the tree will not be able to simply inherit the logic of the data type definition. For example, recall the definition of `yield/2` from the section on trees. (The version given here uses difference lists in place of calls to `append/3`.)

```
yield( Tree, List ) :-
        yield( Tree, List, []).

yield( Label/[], [Label|Yield], Yield ).
yield( _/SubTrees, Yield0, Yield ) :-
        SubTrees = [_|_], % Not a leaf.
        yield_star( SubTrees, Yield0, Yield ).

yield_star( [], Yield, Yield ).
yield_star( [Next|Rest], Yield0, Yield ) :-
        yield( Next, Yield0, Y1 ),
        yield_star( Rest, Y1, Yield ).
```

When we reach the leaf nodes of the input tree we have to add their labels to the output list; at non-leaf nodes we do nothing with the labels. So we need to be able to recognize the leaves of the tree. This code uses the first strategy: we only call the recursive clause of `yield/3` if the list of subtrees is nonempty, that is, the current node is not a leaf.

The next example offers us some more extensive opportunities for optimization. Repeated below is the code for `taddress/3`, from the section on trees, together with the code for `nth/3`, which it generalizes from lists to trees.

```
taddress( [], R/_Subs, R ).
taddress( [K], _R/Subs, Node ) :-
        nth( K, Subs, Node/_NSubs ).
taddress( [K|Ks], _R/Subs, Node ) :-
        nth( K, Subs, S_k ),
        taddress( Ks, S_k, Node ).

nth( 1, [X|_Rest], X ).
nth( N, [_X|Rest], Y ) :-
        M is N - 1,
        nth( M, Rest, Y ).
```

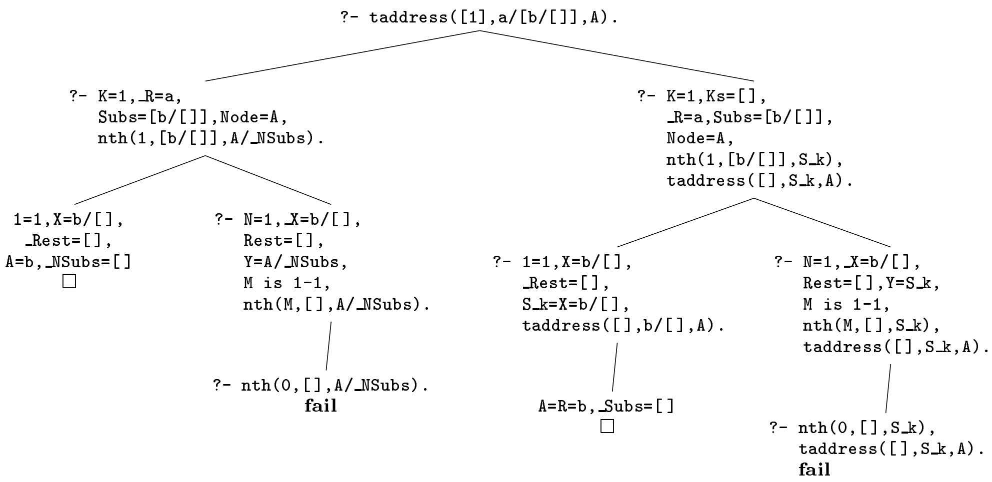



Figure 9.4: Search tree for the goal '`?-taddress([1],a/[b/[]],A).`'

A search tree for the goal `?-taddress([1],a/[b/[]],A)` is given in figure 9.4.

First note that the two success nodes are redundant. Each one finds the same answer in a different way. Once again there are several ways we can go to remove the redundancy. The problem is that when the first argument of `taddress/3` is a singleton list, then it unifies with either the second or third clauses. So one way we could improve matters would be to alter the third clause of `taddress/3` so that it only applied when the first argument was strictly longer than a singleton list.

```
taddress( [K1,K2|Ks], _R/Subs, Node ) :-
     nth( K1, Subs, S_k ),
     taddress( [K2|Ks], S_k, Node ).
```

This has the effect of pruning away the right-hand subtree in figure 9.4.

A better solution might be simply to remove the second clause of `taddress/3`. As we can see from the search tree, this case is adequately handled by the given recursive clause and the first base case. This makes the code much more compact. However, the number of steps to a success node is longer on the search-subtree that remains (the right hand subtree), though only slightly, and the three clause definition though longer is closer to the definition of a tree domain. There is thus little to choose between the two courses, and one's choice will ultimately be a matter of style, most likely.

We are not done yet! We have considered closely how to optimize the definition of `taddress/3`. Now let us look at the definition of `nth/3` on which it is based. When the first argument to `nth/3` is 1 then it unifies with the heads of both clauses. (Put another way, the head of the first clause unifies with the head of the second clause.) However, we never actually want to execute the second clause in this case. In fact, if we were performing only a recursion on the value of the first argument—instead of a simultaneous recursion on that value and the structure of the second argument—then we would be in serious trouble! That branch would be infinite! In particular, once the value of the first argument becomes zero or a negative number then the base case will never apply again; the value of the first argument will just keep on shrinking *ad infinitum.* So it makes a lot of sense to alter this clause as follows. First we put in a guard on the index argument, which prevents the recursive clause from being used if N sinks below 2. Second, since the clause can only succeed when its list argument has more than one member (i.e., two or more members), we alter the pattern which the list argument must match so that only lists with more than one member will match it. This last step actually removes the entire offending search tree(s), eliminating all failure nodes from the tree!

```
nth( N, [_X1,X2|Rest], Y ) :-
     N > 1,
     M is N-1,
     nth( M, [X2|Rest], Y ).
```

# Chapter 10

# Introduction to Program Transformations

## 10.1 Removing calls to =/2

Suppose we have a definition of append/3 like the following.

```
append( As, Bs, Cs ) :-
      As = [],
      Bs = Cs.
append( As, Bs, Cs ) :-
      As = [X|AsTail],
      Cs = [X|CsTail],
      append( AsTail, Bs, CsTail ).
```

Consider now the search tree for the goal

```
?- append( [a], [b,c], L ).
```

given in figure 10.1. Notice that every call to append/3 is binary branching. There are *always* two ways to solve any append/3 goal. This should make a lot of sense: the heads of the two clauses for append/3 are identical, so necessarily whenever the selected literal from the current goal matches one of them it must match the other as well. But notice also that in each case where append/3 is called, one branch fails. And this too will *always* happen when append/3 is called with its first argument ground, because one clause attempts to unify the first argument with the empty list and the other attempts to unify it with a nonempty list, and no ground term unifies with both the empty list and the nonempty list. So one of these two branches is bound to fail.

Now, consider closely what the Prolog interpreter is doing when we give it a goal like ?-append([a],[b,c],L). It selects the leftmost (and only!) literal from the goal, namely append([a],[b,c],L), and unifies it with the head of a program clause. Let us focus on the case where it is unified with the head of

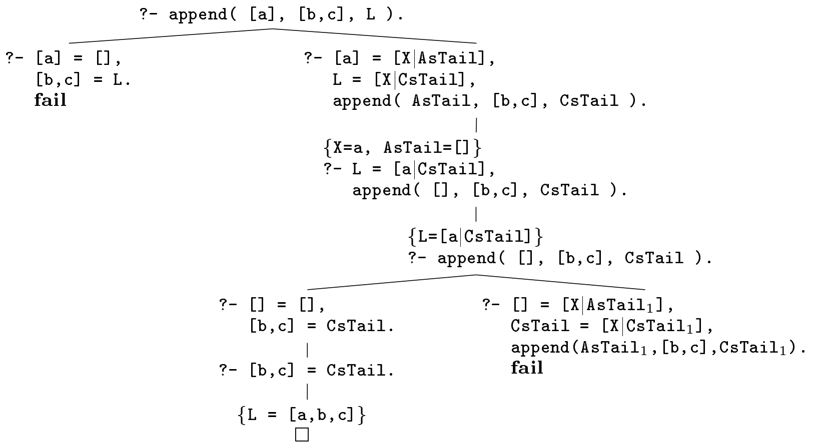



Figure 10.1: Search tree for the goal `?-append([a],[a,b],L).`

the second clause. So now `[a]` is bound to `As`. Immediately thereafter we call `=/2`, attempting to unify `As` with `[X|AsTail]`. Now unification, like equality in general, is *transitive*, so the result of these two unifications,

```
[a] = As, As = [X|AsTail]
```

is bound to be the same as the result of the single unification

```
[a] = [X|AsTail]
```

In particular, both have as consequences that

```
X = a, AsTail = [].
```

Since whatever is in the first argument position will ultimately have to unify with `[X|AsTail]`, we may as well not postpone this unification until the body of the clause is executed. Instead, by replacing `As` with `[X|AsTail]` in the head of the clause, we can make the unification take place as part of the process of matching a goal-literal with a clause-head. We can apply the same reasoning everywhere where calls to `=/2` appear in the definition of `append/3`, giving us the new, more compact implementation that follows.

```
append( [], Bs, Bs ).
append( [X|AsTail], Bs, [X|CsTail] ) :-
     append( AsTail, Bs, CsTail ).
```

A search tree for the same goal, but with this new implementation of `append/3`, appears in figure 10.2. Notice first that this tree is much narrower than the tree in figure 10.1: it doesn't branch at all! By moving the unifications up into the head of the clause, we have in effect pruned away to subtrees of the search tree. This is desirable since those subtrees contained only failure nodes. That is, we

```
   ?- append( [a], [b,c], L ).
                  |
{X=a,AsTail=[],L=[a|CsTail]}    ,
?- append( [], [b,c], CsTail ).
                  |
   {CsTail=[b,c],L=[a,b,c]}
                  □
```

Figure 10.2: Another search tree for `?- append([a],[b,c],L).`, with the new version of `append/3`.

have not lost anything by doing this, and we have gained a search tree which is much smaller, with many fewer choices to keep track of.

Not only is the tree narrower, it is also shallower. In particular, in the tree in figure 10.1, the path from the root to the success node contains seven nodes. But in the tree in figure 10.2, the path from the root to the success node contains only three nodes! So we have made improvements in two dimensions: we have removed a number of choicepoints which we didn't need, and we have shortened the number of steps to the shortest proof. And we did all this with an extremely simple transformation of our original program whose simplicity makes it easy to see that the logical semantics of a program will never be changed by such a transformation. The transformed program has the same consequences as the original program, but its *procedural* semantics is much improved; it is a much more efficient program.

There are actually two ways to view an optimization like this. The first is the obvious one: given a program with explicit calls to `=/2` we can very often improve its performance by eliminating those calls. But we can turn this observation on its head and observe that while we are developing a program we can *always* introduce calls to `=/2` whenever it is convenient, knowing that we can always later remove them with an extremely simple modification of the code. Often it is enough to know that you *can* improve a program's performance, because it may be easier to show that it is correct, for example, with the unoptimized version. Also, as we will see in a moment, it is often easier to apply other program transformations to clauses where all or most of the unifications are implemented in explicit calls to `=/2`.

## 10.2 Partial Execution

We have observed already that we can concatenate sequences in constant time if we represent them as difference lists and use `dl_append/6` instead of `append/3`. The definition of `dl_append/6` is repeated here.

```
dl_append( AsFr, AsBk, BsFr, BsBk, CsFr, CsBk ) :-
      AsFr = CsFr,
```

```
        AsBk = BsFr,
        BsBk = CsBk.
```

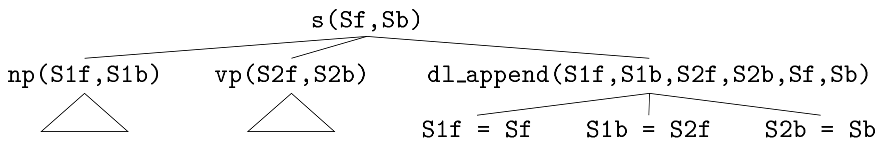
Suppose now that we have defined a predicate using append/3 and we want to take advantage of the efficiency gains that difference lists offer us. Let us take for an example the definite clause version of the phrase structure grammar rule $S \longrightarrow NP\ VP$.

```
    s( String ) :-
        np( String1 ),
        vp( String2 ),
        append( String1, String2, String ).
```

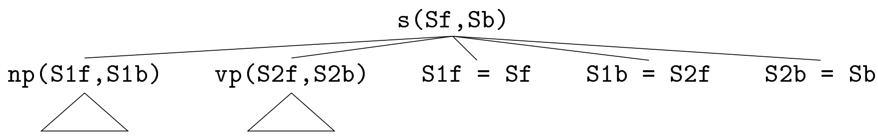
We know already that we can change the data structures to difference lists, and simply replace the call to append/3 with dl_append/6.

```
    s( StrFr, StrBk ) :-
        np( Str1Fr, Str1Bk ),
        vp( Str2Fr, Str2Bk ),
        dl_append( Str1Fr, Str1Bk, Str2Fr, Str2Bk, StrFr, StrBk ).
```

Now, dl_append/6 is defined with only one clause. So every time this clause for s/2 is used, it will always be solved by expanding the call to dl_append/6 to instances of the same three unifications. That is, every proof tree in which this clause for s/2 is used will contain a subtree of the form




The structure of the proof trees on the left (the calls to np/2 and vp/2) will vary with the string that is being recognized or generated by the grammar, but the structure of the right-hand tree will always be the same. So we can be sure that whenever this clause of s/2 is used, if it has a proof tree that looks like the one above, we know that it could also have a proof tree that looks like the one given below.




In this proof tree the call to dl_append/6 has been replaced by the body of its sole and only clause, under a suitable variable substitution. This proof tree would be generated by the following modified code.

```
    s( StrFr, StrBk ) :-
        np( Str1Fr, Str1Bk ),
```

```
        vp( Str2Fr, Str2Bk ),
        Str1Fr = StrFr,
        Str1Bk = Str2Fr,
        Str2Bk = StrBk.
```

Now the length of the shortest proof is shortened somewhat. But this transformation has also revealed a further possible optimization, with even greater potential savings, namely that we can now get rid of the calls to =/2, giving us the even more efficient implementation below.

```
s( Str1Fr, Str2Bk ) :-
        np( Str1Fr, Str1Bk ),
        vp( Str1Bk, Str2Bk ).
```

Notice that we have done something more than just move some expressions up into the head of the clause. We have also renamed the first argument of the call to `vp/2` *within the body of the clause.* The general idea is the same: we have simply traded *explicit* unifications, represented with a call to =/2, with *implicit* unifications, which are represented by simply replacing one expression with another in the actual code of the program.

Seen in this light, both transformations are very similar. In effect what the Prolog interpreter does *on-line* we have done *off-line*, by modifying the code to force certain unifications to take place automatically, or to reflect the fact that a certain proof step will always be executed in the same way. This family of techniques can be called *partial execution*, because they involve the programmer "running Prolog in her/his head" for a step or two, and replacing the code which was "executed" in this way with the results of that (partial) execution. With reference to the predicate `s/2`, the explicit unification `Str1Bk=Str2Fr` will always have the effect, whenever it is executed, of identifying whatever is the current value of `Str1Bk` with whatever is the current value of `Str2Fr`. We can effect the same identification by simply calling these two objects by the same name, as we do in the final implementation of `s/2`. Keep in mind that this is exactly what Prolog is doing when we call =/2, which is just defined as

```
X = X.
```

In the case of `dl_append/6`, we know that in any resolution-based proof which has used this clause of `s/2`, if we select the literal which is an instance of `dl_append( Str1Fr, Str1Bk, Str2Fr, Str2Bk, StrFr, StrBk )` for processing, then it will always be replaced by the corresponding instances of those three calls to =/2. So we can "partially execute" this expansion of `dl_append/6` by rewriting our code in exactly the same way. We "select" the literal `dl_append( Str1Fr, Str1Bk, Str2Fr, Str2Bk, StrFr, StrBk )` from the body of the clause for "processing" (that is, for transformation), and we replace it in the body of the clause with its own body, under a suitable substitution, namely, {`AsFr=Str1Fr, AsBk=Str1Bk, BsFr=Str2Fr, BsBk=Str2Bk, CsFr=StrFr, CsBk=StrBk`}.

Note that this transformation is not restricted to predicates which are defined using a single clause. However, if we partially execute a call to a predicate which is defined in more than one clause, then we must create as many new instances of the clause we are transforming as there are clauses in the definition of the predicate we are partially executing. For example, if we wanted to partially execute a call to (classical) append/3, which is defined with two clauses, then we would replace the clause we are transforming with two new clauses, one replacing append/3 with its base clause, the other with its recursive clause. So if we do this a lot, then the size of the program can increase exponentially. We never have to worry about this combinatorial explosion when we partially execute "1-clause predicates."

Now consider the following implementation of member/2.

```
member( X, L ) :- append( _Left, [X|_Right], L ).
```

Since this is logic programming that we're doing, we might expect to be able to prove the relation between our original definition and this new one. As is often the case, the proof is not of so much interest for its own sake as for the sake of the light it sheds on how one program can be transformed into another. The code for the original implementation of append/3 is given again below for reference.

```
append( As, Bs, Cs ) :-
      As = [],
      Bs = Cs.
append( As, Bs, Cs ) :-
      As = [E|AsTl],
      Cs = [E|CsTl],
      append( AsTl, Bs, CsTl ).
```

We begin by partially executing, or *unfolding*, the definition of member/2. What this means is that we replace the call to append/3 with the bodies of the clauses in the definition of append/3, just as if we were Prolog executing a resolution step.[1] There are two clauses in the definition of append/3, so we must transform the single clause for member/2 into two clauses.

We begin by partially executing the call to append/3 in the body of member/2 using the base clause. We proceed exactly as if we were trying to imitate the behavior of the Prolog interpreter. We match the selected literal

```
append( _Left, [X|_Right], L )
```

against the head of the selected program clause

```
append( As, Bs, Cs ).
```

They match, with mgu

$\sigma$ = { As/_Left, Bs/[X|_Right], Cs/L }.

[1] Or, more accurately, two Prologs running in parallel, each having selected a different clause in the definition of append/3.

We replace the selected literal with the body of the selected clause, and apply the substitution $\sigma$, yielding the new clause

```
member( X, L ) :-
      _Left = [],
      [X|_Right] = L.
```

The clause for `member/2` that we derive contains some singleton variables (marked with an underscore, but still named for reference in this text). The variable `_Left` is never used anywhere else, so the unification `_Left = []` is meaningless. It always succeeds, since `_Left` is always a variable (there is no way to bind it to anything previously), and it adds no information, that is, it does not act to constrain the values of variables elsewhere in the clause. So we can remove this subgoal without any problems. The result

```
member( X, L ) :-
      [X|_Right] = L.
```

is clearly an alphabetic variant of the base clause of the original explicitly recursive definition of the predicate `member/2`. In particular, it no longer mentions `append/3`; this is because we expanded the call to `append/3` using its base clause. In effect, we used the base clause (of `append/3`) to derive another base clause (of `member/2`.

That was the easy one! Now we do the same thing using the recursive clause of `append/3`. The "selected literal"

```
append( _Left, [X|_Right], L )
```

matches the head of the recursive clause

```
append( As, Bs, Cs )
```

under the same substitution

$\sigma$ `= { As/_Left, Bs/[X|_Right], Cs/L }.`

We replace the selected literal with the body of the selected clause under the substitution $\sigma$ and derive

```
member( X, L ) :-
      _Left = [E|AsTl],
      L = [E|CsTl],
      append( AsTl, [X|_Right], CsTl ).
```

Once again, since _Left is a singleton variable, unifying it with anything is (a) always possible and (b) always a waste of time, so we can eliminate the first subgoal.

```
member( X, L ) :-
      L = [E|CsTl],
      append( AsTl, [X|_Right], CsTl ).
```

But we are still left with another call to `append/3`; apparently we haven't done anything very significant at all.

Now the magic happens. We observe that the call to `append/3` in the second clause of the new `member/2` unifies with the *body* of the sole clause of the original definition, repeated below, with variables named apart.

$$\texttt{member( X}_1\texttt{, L}_1\texttt{ ) :- append( \_Left}_1\texttt{, [X}_1\texttt{|\_Right}_1\texttt{], L}_1\texttt{ ).}$$

The mgu here is

$$\theta = \{\ \texttt{\_Left}_1\texttt{/AsTl, X}_1\texttt{/X, \_Right}_1\texttt{/\_Right, L}_1\texttt{/CsTl}\ \}$$

Logically, this means that

$$\texttt{append( AsTl, [X|\_Right], CsTl )} \Rightarrow \texttt{member( X, CsTl )}$$

(the ':-' means implication, remember?). So we replace the call to `append/3` with the call to `member/2` that it implies.

```
member( X, L ) :-
     L = [E|CsTl],
     member( X, CsTl ).
```

(This procedure is often called *folding*, and the entire procedure, in which we first unfold a call and then subsequently fold it is calld the *fold/unfold* transformation.) This clause should look very familiar! It is essentially an alphabetic variant of the recursive clause of the original definition of `member/2`. In particular, there are no longer any calls to `append/3`.

As it happens, we developed the recursive definition of `member`/2 before we developed the simpler definition using `append`/3. If we had not done so, but had instead developed the `append`-based version first, we now see that we could have developed the explicitly recursive version of `member`/2 *mechanically* from that definition. The procedure requires some care, but very little creativity. In fact, it can be automated: there exist programs which apply this "fold/unfold" transformation automatically!

So we have our proof that the two implementations of the membership relation are equivalent. Along the way we have discovered a technique for transforming a program into a new program.

Sometimes fold/unfold can lead to a considerable gain in efficiency. The efficiency gains in this case are not particularly spectacular, but now consider the definition of `select`/3 in terms of `append`/3.

```
select( X, Source, Remainder ) :-
     append( Before, [X|After], Source ),
     append( Before, After, Remainder ).
```

We will proceed by unfolding the calls to `append/3`, as before. Note that this can be counter-productive: `append/3` has two clauses in its definition, and there are two calls to it in the definition of `select/3`, so we could wind up unfolding `select/3`'s single clause into *four* clauses. In general, the unrestricted application of this program transformation will lead to this sort of combinatorial explosion. In this case we are okay, however. The two clauses for `append/3` are distinguished by the value of the first argument: it acts as a switch to decide between clauses. In this case the first argument of both calls to `append/3` is `Before`, so only two of the four logically possible "unfoldings" are in fact consistent. The other two lead to binding `Before` simultaneously to the empty list and to a nonempty list. We are then concerned with the two cases where (1) `Before = []` and (2) `Before = [Elt|Rest]`, for some expression `Elt` and list `Rest`. In case (1), we derive the following code from the base clause of `append/3`.

```
select( X, Source, Remainder ) :-
      Before = [],
      [X|After] = Source,
      Before = [],
      After = Remainder.
```

Obviously we do not need both instances of `Before = []`. In fact, since `Before` is essentially a singleton variable, we do not need *either* of these calls. Removing them gives us

```
select( X, Source, Remainder ) :-
      Source = [X|After],
      Remainder = After.
```

the base clause of the standard recursive definition of `select/3`.

In case (2), where `Before = [Elt|Rest]`, the first call to `append/3` is unfolded under the substitution

$$\sigma_1 = \{ \mathtt{As}_1/\mathtt{Before},\ \mathtt{Bs}_1/\mathtt{[X|After]},\ \mathtt{Cs}_1/\mathtt{Source} \}$$

and the second under the substitution

$$\sigma_2 = \{ \mathtt{As}_2/\mathtt{Before},\ \mathtt{Bs}_2/\mathtt{After},\ \mathtt{Cs}_2/\mathtt{Remainder} \}.$$

This yields the unfolded clause:

```
select( X, Source, Remainder ) :-
      Before = [Head1|AsTl1],
      Source = [Head1|CsTl1],
      append( AsTl1, [X|After], CsTl1 ),
      Before = [Head2|RestAs2],
      Remainder = [Head2|CsTl2],
      append( AsTl2, After, CsTl2 ).
```

If, on considering this code, you find yourself wondering if this is all worth it, a moment's patience will reward you! But at the moment the code for `select/3` certainly does look ugly. We can, however, do some obvious things to transform it to a more useful form. First note that `Before` is equated with two different expressions, so we can unify them as follows.

```
[Head_1|RestAs_1] = [Head_2|RestAs_2]
```

$$\theta = \{\ \texttt{Head}_1/\texttt{Head}_2,\ \texttt{RestAs}_1/\texttt{RestAs}_2\ \}$$

Once we collapse the two now-identical unifications with `Before`, it becomes a singleton variable, and we can eliminate even the sole remaining equation. After applying $\theta$ we get the following code.

```
select( X, Source, Remainder ) :-
      Source = [Head_2|RestCs_1],
      append( RestAs_2, [X|After], RestCs_1 ),
      Remainder = [Head_2|RestCs_2],
      append( RestAs_2, After, RestCs_2 ).
```

Now things are looking better. Noting that conjunction is commutative, we now rearrange the subgoals of `select/3` to yield the code below.

```
select( X, Souce, Remainder ) :-
      Source = [Head_2|RestCs_1],
      Remainder = [Head_2|RestCs_2],
      append( RestAs_2, [X|After], RestCs_1 ),
      append( RestAs_2, After, RestCs_2 ).
```

We now show again the definition of `select/3`.

```
select( X_3, Source_3, Remainder_3 ) :-
      append( Before_3, [X_3|After_3], Source_3 ),
      append( Before_3, After_3, Remainder_3 ).
```

As before, we can see that the calls to `append/3` in the "unfolded" clause unify with the body of original clause. Under the derived substitution, the head of the (untransformed) program clause would become

```
select( X, RestCs_1, RestCs_2 )
```

Put another way, these two literals are exactly what we would get by issuing the goal

```
?- select( X, RestCs_1, RestCs_2 ).
```

We can now *fold*, then, and replace the two calls to `append/3` with the above goal, deriving the final code for `select/3`, given in full (with slightly cleaned-up variable names) below.

```
select( X, Source, Remainder ) :-
      Source = [X|After],
      Remainder = After.
select( X, Source, Remainder ) :-
      Source = [Head|RestSource],
      Remainder = [Head|RestRemainder],
      select( X, RestSource, RestRemainder ).
```

```
prefix( X, Y ) :- X = [], list( Y
      ).
prefix( X, Y ) :-
      list( X ),
      list( Y ),
      X = [Head|XTail],
      Y = [Head|YTail],
      prefix( XTail, YTail ).

list( []).
list( L ) :-
      L = [First|Rest],
      list( Rest ).
```

Figure 10.3: `Prefix/2`, with type checking, and `list/1`.

This actually represents a significant gain in efficiency. Under the old definition, owing to the nature of `append/3`, the `Source` list (up to `X`) had to be traversed twice: once to find `X` and once to build `Remainder`. In the new code these two distinct phases have been *combined*: we build up `Remainder` while we are searching for `X`.

This concern for efficiency is really anticipating material that we will focus on later. However, the goals of efficiency and logical simplicity are very tightly related in logic programming. In *The Craft of Prolog*, Richard O'Keefe actually elevates this observation to a slogan: "Elegance is not optional!" Our final example illustrates this point.

Up until now we have been very lax about type-checking. The careful reader will have noted, for example, that the following two goals would succeed given our interpretations, even though the arguments are not in fact all lists.

```
?- member(a,[a|b]).

?- append([a,b],c,[a,b|c]).
```

We have also been careful to give specifications in the form "`p(X)` is true if, whenever `X` is a list, ... ," rather than in the strict form "`p(X)` is true if `X` is a list and ... ." We turn our attention now to programs meeting these stricter specifications. Consider the code in figure 10.3 for `prefix/2`, with type-checking. (By the way, this code does not require its first argument to be a *proper* prefix of its second—the two can be equal.)

Here we are in the same position as with `select/3`. The calls to `list/1` will traverse each of the two lists, then `prefix/2`, which is a recursion on the list in its first argument (actually a simultaneous recursion on both arguments), will traverse the two lists again. Worse, `list(X)`, for example, will be called over and over again, once each time `prefix/3` calls itself. We can do better than this, and with much simpler code!

```
prefix( X, Y ) :-
      X = [First₁|Rest₁],      % list( X ),
      list( Rest₁ ),
      Y = [First₂|Rest₂],      % list( Y ),
      list( Rest₂ ),
      X = [Head|XTail],
      Y = [Head|YTail],
      list( XTail ),      % prefix(XTail,YTail).
      list( YTail ),
      XTail = [Head₃|XTail₃],
      YTail = [Head₃|YTail₃],
      prefix( XTail₃, YTail₃ ).
```

Figure 10.4: Unfolded code for `prefix/2`, before simplification.

```
prefix( X, Y ) :-
      X = [Head|XTail],
      Y = [Head|YTail],
      list( XTail ),      % ⇒ prefix(XTail,YTail)
      list( YTail ),
      XTail = [Head₃|XTail₃],
      YTail = [Head₃|YTail₃],
      prefix( XTail₃, YTail₃ ).
```

Figure 10.5: Unfolded code for `prefix/2` after simplification.

We cannot do anything to improve the base clause, so we will focus on the recursive step. We will unfold (once) each of the subgoals `list(X)`, `list(Y)` and `prefix(XTail,YTail)`. Note that there are two possible unfoldings of `prefix(XTail,YTail)`, since `prefix/2` is defined in two clauses. We tackle first the case where it is unfolded using its own recursive step. The unfolded code is given in figure 10.4.

We can simplify this code considerably before we continue. First note that

$$\begin{aligned} \mathtt{X} = [\mathtt{First}_1|\mathtt{Rest}_1] \wedge \mathtt{X} = [\mathtt{Head}|\mathtt{XTail}] \quad &\Rightarrow \quad \mathtt{First}_1 = \mathtt{Head} \wedge \mathtt{Rest}_1 = \mathtt{XTail} \\ \mathtt{Y} = [\mathtt{First}_2|\mathtt{Rest}_2] \wedge \mathtt{Y} = [\mathtt{Head}|\mathtt{YTail}] \quad &\Rightarrow \quad \mathtt{First}_2 = \mathtt{Head} \wedge \mathtt{Rest}_2 = \mathtt{YTail} \end{aligned}$$

so we can eliminate the extra equations, performing the obvious substitution at the same time. Rearranging the subgoals slightly, we get the simplified code in figure 10.5.

Now we can see that the last 5 subgoals of `prefix/2` unify with the body of the recursive clause in the original definition, so we can replace them with the single goal

```
prefix( XTail, YTail ).
```

```
prefix( X, Y ) :-
     X = [First1|Rest1],     % list(X),
     list( Rest1 ),
     Y = [First2|Rest2],     % list(Y),
     list( Rest2 ),
     X = [Head|XTail],
     Y = [Head|YTail],
     XTail = [],     % prefix(XTail,YTail).
     list( YTail ).
```

Figure 10.6: Unfolding `prefix/2` using its base clause (unsimplified).

```
prefix( X, Y ) :- X = [], list( Y ).
prefix( X, Y ) :-
     X = [Head|XTail],
     Y = [Head|YTail],
     prefix( XTail, YTail ).
```

Figure 10.7: Final code for `prefix/2` with type-checking.

But we are not done yet! There was another possible way to expand the call to `prefix( XTail, YTail )` in the original definition: we could have used the base clause. This unfolding is given in figure 10.6.

Once again we can do some simplifying.

$$\begin{array}{lcl} \mathtt{X} = [\mathtt{First}_1|\mathtt{Rest}_1] \wedge \mathtt{X} = [\mathtt{Head}|\mathtt{XTail}] & \Rightarrow & \mathtt{First}_1 = \mathtt{Head} \wedge \mathtt{Rest}_1 = \mathtt{Tail} \\ \mathtt{Y} = [\mathtt{First}_2|\mathtt{Rest}_2] \wedge \mathtt{Y} = [\mathtt{Head}|\mathtt{YTail}] & \Rightarrow & \mathtt{First}_2 = \mathtt{Head} \wedge \mathtt{Rest}_2 = \mathtt{YTail} \end{array}$$

This leaves us with

```
prefix( X, Y ) :-
     X = [Head|XTail],
     Y = [Head|YTail],
     XTail = [],
     list( YTail ).
```

Now the last two lines unify with the base clause of `prefix/2`, so we can replace them with the subgoal

```
prefix( XTail, YTail )
```

Fortunately, this yields exactly the same code as we got before, so both unfoldings, using the recursive clause and using the base clause, yield the code in figure 10.7. (The base clause has been included here for completeness.)

We arrive at a rather amazing result: we can simply remove all type-checking from the definition of `prefix/2`, except in the base clause. And there we only have to check the remainder of the input argument, that is, that part of it that follows the prefix `X`. The reason for this is simple enough, and indeed if I were not looking for ways of demonstrating partial execution I would probably just make this informal argument: every recursive step of the execution of `prefix/2` can only succeed if its arguments are lists. This is because it applies the dot construction to each argument anyway, as part of its core function. The only possible problems would come from goals like

```
?- prefix( [a|b], [a,b,c] ).

?- prefix( [a,b], [a,b|c] ).
```

where the arguments are just like lists, up to a point. The first case will eventually generate a subgoal of the form

```
?- prefix( b, [b,c] ).
```

which will fail, since `b` is neither the empty list nor an instance of a non-empty list. Similarly the second case will succeed, as far as `prefix/2` is concerned, but will eventually yield a subgoal of the form

```
?- list( c ).
```

which will fail.

This result will hold true in general when a predicate is defined by recursion on a structure: it will *automatically* check the well-formedness of the structure during every recursive step. We typically only need to worry when we reach the base case. In figure 10.8, we see the code for some familiar predicates on lists and natural numbers, with type-checking.

```
append( [], Bs, Cs ) :-
     Bs = Cs,
     list( Bs ).
append( As, Bs, AsBs ) :-
     As = [A|RestAs],
     AsBs = [A|RestAsBs],
     append( RestAs, Bs, RestAsBs ).

plus( 0, X, Y ) :-
     X = Y,
     nn( X ).
plus( X, Y, Z ) :-
     X = s(XLess1),
     Z = s(ZLess1),
     plus( XLess1, Y, ZLess1 ).

select( X, Source, Remainder ) :-
     Source = [X|Remainder],
     list( Remainder ).
select( X, Source, Remainder ) :-
     Source = [KeepMe|RestSource],
     Remainder = [KeepMe|RestRemainder],
     select( X, RestSource,
     RestRemainder ).
```

Figure 10.8: Some familiar predicates, with type-checking.

# Chapter 11

# Negation-as-Failure and Cut

## 11.1 Negation as Failure

We begin with a brief review. Consider for a moment the problem of removing all occurrences of a specified element from a list. We could try to implement it as follows, on the model of `member/2` or `select/3`.

```
remove( X, [], []).
remove( X, [X|Rest0], Rest ) :-
      remove( X, Rest0, Rest ).
remove( X, [Y|Rest0], [Y|Rest] ) :-
      remove( X, Rest0, Rest ).
```

Unfortunately, as we know, this doesn't work. For example, the following query succeeds.

```
?- remove( a, [a,b,c], [a,b,c] ).
```

Clearly, `a` has not been removed. What happened? Well, the goal unifies with either the second or third clause of the definition. If we resolve against the second clause, which is what we are clearly *meant* to do, then we will fail. But when we fail, we will backtrack, and so the third clause will be tried, causing `a` to remain in the output.

What we *wanted* to say was that `Y` should be kept in the output list, *if it does not equal X*. But without some sort of negation we cannot say this.

As we know, we can often get around the lack of explicit negation in Definite Clause Logic by using properties of specific problems. So, for example, if the input list is always going to be a list of *numbers*, then we can write the following.

```
remove( X, [], []).
```

```
remove( X, [X|Rest0], Rest ) :-
      remove( X, Rest0, Rest ).
remove( X, [Y|Rest0], [Y|Rest] ) :-
      Y < X, remove( X, Rest0, Rest ).
remove( X, [Y|Rest0], [Y|Rest] ) :-
      Y > X, remove( X, Rest0, Rest ).
```

Here we have taken advantage of the fact that for numbers, and indeed for any linearly ordered set, not-equals is the same thing as greater-than-or-less-than. To take another example from the domain of numbers, the property of being "not odd" is identical to the property of being even; often we can use similar reasoning to identify a positive property of a data item which is identical to the negative property which, intuitively, is what we are actually after. So, for example, in a tree structure every node is either above, below, before, after or equal to every other node. So to express the constraint that "node A is not above or equal to node B," we could say "node A is below, before or after node B." Finally, if we have a limited domain of data objects we are interested in, then we can sometimes actually implement our own theory of equality, or disequality, for example with a (maybe very large!) collection of unit clauses of the form `neq(a,b).`

Sometimes, however, there is no way around it. We really need some form of negation. Let us consider for example the problem of the "not member" relation, which holds between an expression `A` and a list `L` just in case `A` is not a member of `L`. First note that, because lists can contain arbitrary expressions as elements, we cannot hope to use any of the strategies mentioned above. We should be able to reason as follows: if we execute a call to `member(A,L)`, and it fails, then it is reasonable to suppose that `A` is in fact not a member of `L`. This form of negation, which is actually implemented in Prolog, is usually called *Negation As Failure*, and written '\+'. It is *very* important to keep in mind that

```
\+ member( A, L )
```

does *not* mean exactly the same thing as the logical formula

$$\neg member(A, L).$$

It means "I couldn't find a way to prove that `member(A,L)` holds." Fortunately, it is often the case that these two notions line up, and we can pretend that negation-as-failure is in fact negation. In particular, when there are no uninstantiated variables in the negated goal, then negation-as-failure is a sound implementation of "real" negation.

However, if there are variables in the negated goal, all bets are off!

To understand what just happened, we should look more closely at Negation-as-Failure. First, we will update our definitions of search trees to deal with this new language including the negation operator.

Recall that every branch of a search tree ends—if it ends at all—in either a success node or a failure node. From this we can define a *failed search tree* as one all of whose finite branches end in failure nodes; then a tree is *finitely failed* if *all* of its branches end in failure nodes (a fortiori, all of its branches are finite). The computation of a goal *fails* just in case that goal labels the root of a finitely failed search tree. This gives some formal content to the notion of a failed computation, and hence to negation-as-failure. Now the computation of a negated goal `\+G` *succeeds* if `G` has a finitely failed search tree. If `G` has a finite search tree with at least one success node, then `\+G` *fails.* If `G` has a search tree with an infinite branch, and at least one success node, then `G` *should* succeed, but whether or not Prolog will find the solution before it finds the infinite branch depends on a number of factors. However, if the search tree for `G` has only failure nodes and infinite branches we do not consider `G` to fail. We simply do not know what decision to make regarding `G`. This makes intuitive sense: in order to show that `G` is satisfiable, all we need to do is to find one successful branch of its search tree; but in order to show that `G` is not satisfiable, we need to show that no successful branch exists. In this latter case we actually have to visit all the branches, so there is no avoiding an infinite branch if one is present in the tree. And in that case no computational system can possibly ever arrive at a definite decision.

So now armed with these notions we can construct search trees for goals and programs with negation-as-failure. If the currently selected subgoal at a particular node of the search tree is a negated goal of the form `\+G`, then we construct a separate search tree with root ←`G`. If that "auxiliary" tree is finitely failed, then the original goal `\+G` succeeds. We treat negated goals almost exactly the same way as we treat unit clauses: if they succeed they are simply removed from the goal. They have no body that we should replace them with. There is one big difference, however: *negated goals never bind any variables!* This makes a certain amount of sense, since variable bindings are part of the definition of a solution to a problem, and we succeed on ←`\+G` just in case `G` has no solutions. But note that this was not true of all of our "home-made" implementations of negation. In particular, if we define a predicate `neq`/2 with a set of unit clauses, then it may well be generative, so that goals of the form `neq(A,B)` may succeed, binding `A` and/or `B` to some value, where the corresponding call to `\+ A=B` may succeed, but will never bind `A` or `B` to anything. One way to think about this in terms of search trees is to treat the procedure of solving a negative literal `\+G` as follows:

- Construct an auxiliary search tree for the goal ←`G`.
- If that tree is finitely failed, then replace the selected literal, `\+G`, with `true` (a zero-place goal that always succeeds). If that tree has a success node, then replace the selected literal with `fail` (a zero-place goal that always fails).

This procedure clearly just throws away any information about variable bindings, treating the negated goal as being equivalent to one or another zero-place predicate. So: *all variables in a negated goal are input variables.*

Note that the proof of a negated goal may itself involve proofs of other negated goals, and those proofs in turn may involve more negations. So our auxiliary trees may themselves spawn their own auxiliary trees.

## 11.2 The Cut Operator, and Prolog's Implementation of Negation as Failure

The above acount of negation as failure has skirted an issue that we have avoided for some time. Strictly, a goal `\+G` succeeds if the goal $\leftarrow$`G` has *a* finitely failed search tree. Now, particular search trees depend on the order in which literals are selected from the goal and clauses are selected from the program. That means that in our search for a finitely failed search tree we are free to assume any combination of selection strategies that gives us a such a search tree. And there are even deterministic strategies that always work, if any strategy will work. But Prolog's left-to-right, top-to-bottom strategy is *not* one of them. Consider, for example, the pair of goals

```
?- X=a, Y=b, \+ X=Y.
?- \+ X=Y, X=a, Y=b.
```

Both of these goals can be processed in such a way as to yield a finitely failed search tree for `\+ X=Y`. Simply select the other two subgoals first. But for Prolog, only the first goal succeeds. It's left-to-right selection function will take the second example, select the negated goal first, and construct a search tree for `X=Y`, instead of delaying this goal until it becomes instantiated to `a=b`. The former goal obviously has a success node in its search tree, and so the call to `\+ X=Y` fails. The remaining goals are not even tried, and there is no way to prove that there was in fact a solution. So Prolog will report as false a goal which is in fact true, a completely valid consequence of the underlying theory of equality of terms. So, in order to write correct Prolog programs incorporating negation, it is important to understand how Prolog has implemented its version of negation as failure.

The implementation of \+ in Prolog involves another more primitive operator, written '!' and pronounced "cut". The cut operator acts to actively prune the search tree under program control, actually removing choicepoints so that backtracking will ignore certain alternatives. This is a very powerful operation! We have talked about modifying search trees before, in connection with making programs operate more efficiently. But up until now those modifications were always made *off-line*. We simply changed the program or, in some cases, the goal. Once we decided which program to run and which goal to give it, the search tree was fixed. But this is not the case if a program contains cut! The shape of the search tree now depends on exactly how it is searched, and it *changes* during that search.

The cut operator is without a doubt one of the hardest features of Prolog to master.[1] I will give a definition of what cut does first—which you will almost certainly find to be quite dense—and then we will gradually unpack this definition and see what it really means.

Cut is a zero-place predicate. Logically, it is equivalent to `true`, that is, it always succeeds. But it succeeds *with side effects*. It succeeds and commits the Prolog interpreter to all of the choices it has made since the clause containing the cut was selected for resolving against the goal.

Here is an abstract program fragment we can use as an example. (More concrete examples follow.)

```
p( X ) :- q1(X),!,q2(X).
p( X ) :- ...

q1(X) :- X = a.
q1(X) :- X = b.
q1(X) :- X = c.
```

Note the cut between the calls to $\mathtt{q}_1$ and $\mathtt{q}_2$. Now suppose that we have a goal containing the literal `p(b)`, and that this literal has been selected by the Resolution Rule. Suppose further that clause 1 of the definition of `p/1` has been selected as well. This is the first point in time that we must keep track of. Every choice made from this point until the cut is executed will be committed to; alternatives will be erased. So we must keep track of what those choices are. Choice number one is the choice of clause 1. Prolog, as usual, must record the fact that clause 2 might also be used to help find a proof.

Now we select the first literal in the body of our clause, namely $\mathtt{q}_1\mathtt{(b)}$. It unifies with the heads of all three clauses in the definition of $\mathtt{q}_1/1$. We select the first one, and create choice-points recording that we still have the second and third clauses to consider. Our first choice here in fact fails: we cannot unify `b` with `a`. Backtracking is still working like normal (we haven't called the cut yet), so we check our stack of choice points and pop the choice corresponding to clause 2 of $\mathtt{q}_1/1$. This time we succeed, and the next goal to try is the cut. This succeeds, because it always succeeds (once), and it has the side effect of wiping out all of the choice points recorded since Prolog selected its surrounding clause for processing. So we erase the choice points corresponding to clause 3 of $\mathtt{q}_1/1$ and clause 2 of `p/1`. If $\mathtt{q}_2\mathtt{(b)}$ fails, we will have to backtrack all the way to the goal that originally called `p/1` in the first place.

The important thing to observe is that the choice points we have removed fall into two sorts. First, we have committed ourselves to all the choices we made prior to the cut *in the body of its clause*. I will refer to these as the *inner* choice points. Opposed to these we have the *outer* choice points, namely all of the clauses for `p/1` which appear after the clause containing the cut. So we prune away choices within both the definition of the predicate `p/1` and within

[1] Though some would say that you haven't mastered Prolog until you've learned to do without it entirely!

the body of the clause containing the cut. A common source of errors when programming with cut is to forget that it commits you in both places.

By erasing these choice points, cut has the effect of pruning away from the search tree all of the subtrees rooted in those lost choices. Various things can happen as a result, some good, some bad. For example, those trees might have contained only failure nodes, so that we would have wasted time searching them. Pruning them away is thus a good thing. Likewise, they might have contained success nodes, but success nodes which represent only solutions that we have already found. This too is a good thing: there is no point in finding the same solution twice. The pruned search trees might have contained infinite branches, something we can always do without. This too is a good thing. These sorts of cuts, which make a program run more efficiently without altering its consequences, are called *green* cuts.

Opposed to green cuts, we have red cuts. First there is the obvious case: the search trees we prune away might have contained solutions found nowhere else in the search tree. This is usually a bad thing, though there are some cases where even this is allowable. For example, if we are posing only purely existential queries, and we care only about the existence of some solution, then multiple solutions, even when they are distinct, are irrelevant. For example, suppose we have a finite state automaton and we wish to know only whether the language it accepts is empty or not. Then the fact that it accepts two, three or an infinite number of strings is irrelevant; we only need to find one solution, and we don't care which it is.[2]

There is a much more serious use of red cuts, though, and that is when they are used to prune away search trees which contain *incorrect* solutions. On the surface this seems like a good thing: we are after all not only improving the performance of a program, but also its correctness. But the fact that we have to do so means that the program was incorrect to start with. The only sensible logical semantics we can assign to cut is that it is equivalent to `true`. So that means that adding it does not in any way affect the logical semantics of a program. *Logically, the incorrect program is still incorrect!* Put another way, a program with this kind of red cut in it *has lost its declarative semantics.* It can only be interpreted procedurally.

Let us examine these various sorts of cuts in the context of a program for applying a substitution to a term. Assume that a substitution is a list of `VarName-Term` pairs, and that terms are given in our usual slash-notation, except that leaves are Prolog constant expressions, which represent either variable names or constant names of the term-language we are using. Finally, assume that we have defined the term-language using the predicates `variable/1`, `constant/1` and `gen/3`, for the generator functions. Then we have

```
apply( Theta, TermIn, TermOut ) :-
```

[2]Note though that even here, we could have used an anonymous variable to indicate that we didn't care what the solution was. This would have the same effect and be logically pure, as well. That is, a goal like `?-p(_).` should succeed only once, even if there are multiple solutions.

```
     constant( TermIn ),
     TermOut = TermIn.
apply( Theta, TermIn, TermOut ) :-
     variable( TermIn ),
     member( TermIn-Value, Theta ),
     TermOut = Value.
apply( Theta, TermIn, TermOut ) :-
     variable( TermIn ),
     \+member( TermIn-_, Theta ),
     TermOut = TermIn.
apply( Theta, TermIn, TermOut ) :-
     gen( TermIn, F, Args0 ),
     apply_star( Theta, Args0, Args ),
     gen( TermOut, F, Args ).
```

Assuming that we have implemented our language properly, and no expression is both a variable name and a constant name, or a function and a constant, or what have you, then these four clauses are distinct. Only one of them should apply (in the intended standard mode!) in any one case. The "mildest" use of cuts is just to express this intended determinism. So, for example, we might write

```
apply( Theta, TermIn, TermOut ) :-
     constant( TermIn ),
     !,
     TermOut = TermIn.
apply( Theta, TermIn, TermOut ) :-
     variable( TermIn ),
     member( TermIn-Value, Theta ),
     !,
     TermOut = Value.
apply( Theta, TermIn, TermOut ) :-
     variable( TermIn ),
     \+member( TermIn-_, Theta ),
     !,
     TermOut = TermIn.
apply( Theta, TermIn, TermOut ) :-
     gen( TermIn, F, Args0 ),
     apply_star( Theta, Args0, Args ),
     gen( TermOut, F, Args ).
```

Note that no cut is necessary in the final clause: by the time Prolog has backtracked its way to the fourth clause, no alternatives remain in any case. Note also the placement of the cuts. They follow the tests which "guard" the clauses. So, for example, we only mean to use clause 2 if `TermIn` is both a variable and has some binding in `Theta`. Once we know that both of these things are true, we know that only clause 2 can be used. And until we know that both these things are true, we cannot know that only clause 2 can be used. So we put in a cut as soon as we know we are in the right clause and *not earlier and not later.*

That was an example of a green cut: we pruned away search trees which were known to contain only failure nodes. A red cut appears in the following version.

```
apply( Theta, TermIn, TermOut ) :-
      constant( TermIn ),
      !,
      TermOut = TermIn.
apply( Theta, TermIn, TermOut ) :-
      variable( TermIn ),
      member( TermIn-Value, Theta ),
      !,
      TermOut = Value.
apply( Theta, TermIn, TermOut ) :-
      variable( TermIn ),
      !,
      TermOut = TermIn.
apply( Theta, TermIn, TermOut ) :-
      gen( TermIn, F, Args0 ),
      apply_star( Theta, Args0, Args ),
      gen( TermOut, F, Args ).
```

This code is constructed based on the following reasoning. If we ever get to clause 3, then in trying clause 2 either `variable(TermIn)` failed or else `member(TermIn-Value,Theta)` failed. So if `variable(TermIn)` succeeds in clause 3, then we know that it must have been `member(TermIn-Value,Theta)` that failed. So there is no reason to call `\+member(TermIn-_,Theta)`, since we know already that by the definition of negation as failure it must succeed. So we simply leave it out. (Note that we can't apply this reasoning to the last clause since the call to `gen/3` is used not only to check the type of the input term but also to extract its arguments.)

This is a red cut of the second type. Without the cut, variables which had instantiations in `Theta` would be left untouched whenever we used clause 3 to process them. The cut is absolutely necessary because the program is logically incorrect, and so we need the cut to remove the incorrect success nodes from the search tree. This sort of red cut probably provokes the bitterest debate in the Prolog programming community, because, though we have lost logical clarity (the program is now harder to read and much less modular) we have certainly gained efficiency. Imagine that `Theta` was a very long list. We would have to search it twice whenever the search failed (once to fail back out of clause 2 and once to get past the guard on clause 3). With the cut we only have to search it once.

There are several ways to mitigate the problems caused by red cuts of this sort (they really ought to have their own name, like "scarlet cuts" or something).

The readability problem can be gotten around by including comments that indicate what has been removed. For example:

```
apply( Theta, TermIn, TermOut ) :-
      variable( TermIn ),
      % \+member( TermIn-_, Theta ),
      !,
      TermOut = TermIn.
```

The lost declarative interpretation can be gotten around to some degree by retaining a working version of the program code which has the goals in it that have been removed from the optimized version. Then one can argue at least that the optimized version is procedurally equivalent to the unoptimized version, and in some (rather ill-defined) sense "inherits" the declarative semantics of its source. Nonetheless, this sort of optimization should be done only as a last resort.

That said, I have unfortunately to report that that is exactly the way in which the negation as failure operator is implemented in Prolog. It's behavior can be described by the following Prolog program.

```
\+ P :- P, !, fail.
\+ P.
```

First note that this program makes use of Prolog's so-called "meta-variable facility". This allows you to write predicates which take *goals*—not terms—as their arguments, making them, officially, *operators* and not true predicates. So, to take the first clause for \+/1 as an example, its sole argument `P` is assumed to be bound to a legal Prolog goal when \+/1 is called. Then the call to `P` behaves just like a call to whatever goal actually instantiates `P` at the time. Note that if `P` is unbound when called, then the program will crash, not fail.

As a result of the "scarlet cut" in this program, the search tree for goals of the form `?-\+G.` will always contain a success node, because the second clause always succeeds. So logically this predicate is equivalent to `true`! It works, of course, because this permanent success node is pruned away in every case in which we can actually reach the cut. And we can reach the cut whenever `P` succeeds. So we are committed to the left branch of the search tree whenever `P` succeeds, and only reach the right branch if `P` fails. This means that, since whenever `P` succeeds, we call `fail`, then whenever `P` succeeds, `\+P` fails, and `\+P` only succeeds in the case where `P` fails. Which gives us the semantics of negation as failure, but only if we interpret this program using Prolog's left-to-right, top-to-bottom strategy! If we try and apply other definite clause proof procedures to programs with cut in them, we cannot be assured of correct results!

# Chapter 12

# Metalanguage, Metalogic and Metaprogramming

In this section we examine the distinction between *object language* and *metalanguage*. The distinction arises originally in logic. Part of the business of logic is the construction of proof systems, formal systems for deriving expressions from expressions in a truth-preserving manner. We talk about *theorems* as expressions which can be derived from fixed sets of other expressions called *axioms* by the application of a fixed set of *inference rules*. Now we have a formal system for the construction of *proofs*. But we also want to know something about such a formal system, and in particular to prove that it has or has not certain interesting properties. So we spend our time not only constructing proofs *in* the system, but also constructing proofs *about* the system. The formal language in which we write our axioms and theorems is the *object language*, and the (usually) semi-formal language in which we prove mathematical properties of the system as a whole is the *metalanguage*.

The same kinds of distinction appear also in computing. For example, suppose I write a Prolog program. It contains a number of clauses defining relations over terms; one can look at the program and extract from it a certain language with a given set of function symbols, relation symbols and constants, plus the logical connectives (',', ':-', etc.) which Prolog provides to every program. Let this be our object language. Then there is another language in which our entire program becomes one very large term, on whose subterms substitutions can be applied, unification operations executed and the 'resolution relation' defined which holds between a goal and a goal-clause pair just in case the first goal is the resolvent of the goal-clause pair. This language, the language in which the Prolog interpreter is defined, is a metalanguage. But wait! How do you get the Prolog interpreter running in the first place? You sit down in front of a computer running some sort of command-processor (XTerm, for example), and type "prolog" or "sicstus" or some such. This operating system thus defines its own language, in which expressions like "sicstus" are defined, and as far as it is concerned, logically speaking, the entire Prolog interpreter is one big term. And so it goes. In computing there are so many levels of metalanguage, metametalanguage, metametametalanguage, etc., that one can easily become quite confused!

Well, that is an exaggeration. Normally it is not hard to keep the various levels separate. However, from time to time, and indeed quite often in computational linguistics, we need to write programs in which various levels of language are present at the same time. Consider for example the problem of writing a sophisticated expert system with some sort of linguistic knowledge built in. This sytem needs a grammar, in particular a grammar which it can read and understand. Such a "machine-readable grammar" is in many ways like a computer program. In particular it itself must be written in a formal language (a "grammar formalism") so that it represents the rules and constraints of the grammar in an explicit and unambiguous form. Now this program-like grammar is the input to various possible applications. First and foremost some sort of application must exist which allows ordinary linguists to enter and maintain the grammatical knowledge base. As far as this grammar maintenance program is concerned, the grammar is just an object, a database of some sort. But then we need to check that the grammar is correct, that it derives just the grammatical sentences of the language of interest. The program that puts the grammar to work, the grammar interpreter, treats the grammar like a program, in effect "running" it.

So now you, the programmer, must write one of these applications. Your program is written in Prolog. But certain expressions which appear in your program are meant to represent expressions of the grammar description language. So some expressions which appear in your program are special: they obey the syntax and have the interpretation of expressions in the grammar description language. So suddenly you are programming in an environment where two languages exist at once, and your program, which is the object program as far as the Prolog interpreter is concerned, has become the metaprogram as far as the grammar is concerned.

The same situation appears throughout computer science, every time you sit down to write a program which takes another program as its input or produces a program as its output (or both). Similarly in logic programming: every time you write a theorem prover or a new logic programming language you are engaged in *metaprogramming*. In this section we will examine general techniques which can be used to develop metaprograms within a pure (or semi-pure, since we will be using negation) logic programming environment. We will also consider some facilities which are built into Prolog which make metaprogramming much easier, but which take Prolog itself well beyond the bounds of pure logic programming systems. So from now on Prolog will wear two faces: as a logic programming system for definite clause programs using SLD (or SLDNF, NF for negation-as-failure) resolution, and as a sort of "assembler language" for writing other logic programming systems.

## 12.1 Reification: Halfway to Metalogic

Throughout the course I have been acting as if our definitions of lists and successor-terms were fully parallel. In fact this is not quite accurate. The most mathematically pure definition of a sequence, and the one most like the definition of successor terms, would define a sequence as the application of a number of different monadic function symbols to a single argument, the empty sequence. That is, the sequence which we might write as *abcd* is interpreted as the term $a(b(c(d(\epsilon))))$.

More formally, given a set $\Sigma$ of monadic function symbols, i.e., function symbols which are only ever allowed to take one argument,

- $Basic = \{\epsilon\}$,
- $Gen = \{\mathcal{F}_\sigma | \sigma \in \Sigma\}$

where for any expression $t$, $\mathcal{F}_\sigma : t \mapsto \sigma(t)$. Then the set of sequences is as usual the closure of the basic elements under the application of the generators. Compare this to the standard definition of successor terms.

- $Basic = \{0\}$
- $Gen = \{\mathcal{F}_s\}$

where for any expression $t$, $\mathcal{F}_s : t \mapsto s(t)$. In this way the similarity between lists and successor terms, and the way in which lists generalize successor-terms, becomes more obvious. However, this notation with sequence elements as monadic functions is computationally much less useful than the list-notation that we have been using. Suppose for example that $\Sigma = \{a, b, c\}$. Then the code for `member/2`, restricted to lists of $a$'s, $b$'s and $c$'s, looks like the following:

```
member( a, a(T) ).
member( b, b(T) ).
member( c, c(T) ).
member( X, Term ) :-
      Term = a(SubTerm),
      member( X, SubTerm ).
member( X, Term ) :-
      Term = b(SubTerm),
      member( X, SubTerm ).
member( X, Term ) :-
      Term = c(SubTerm),
      member( X, SubTerm ).
```

Since we are now being extra careful about the distinction between object language and metalanguage, I have been much more careful about "overloading" symbols. There are much simpler ways to do this, when we allow ourselves to be sloppy about the distinction between object and metalanguages. Here it is our main purpose to identify exactly where that sloppiness appears and what it is useful for.

First, here is the definition of the language of sequence terms.

```
constant( a ).    constant( b ).    constant( c ).

gen( f_a, T, a(T) ).
gen( f_b, T, b(T) ).
gen( f_c, T, c(T) ).
```

So: in the definition of `gen/3`, we have three Prolog expressions. Taking the first clause as an example, `f_a` represents the generator function $\mathcal{F}_a$; `T` represents the term which serves as argument to the generator function ($t$ in the mathematical version) and `a(T)` represents the expression which is the value of $\mathcal{F}_a$ when it is applied to $t$. Now: generator functions belong to the *metalanguage*. But they *apply* to expressions of the *object language* (to create more expressions of the object language). Essentially, the object language knows nothing of how it was created. This is important because there are many different ways of defining any given formal language. So the construction of object language expressions could be different, but the object languages so constructed could still be the same! For example, consider the following context free grammar. (Quotes around the parentheses indicate that they are terminal symbols, just like $a$, $b$ and $c$.)

$$\begin{array}{l} S \longrightarrow a \text{ `('}\ S \text{ `)'} \\ S \longrightarrow b \text{ `('}\ S \text{ `)'} \\ S \longrightarrow c \text{ `('}\ S \text{ `)'} \\ S \longrightarrow \epsilon \end{array}$$

This generates the same language as our generator functions do, under the standard rewriting interpretation of phrase structure grammars. So, as usual in mathematics, the distinctions we make are based at least in part on our pretheoretic notions of what things should be the same and what things should be different.

In light of this discussion, consider the following much more compact definition of `member/2` for sequence terms.

```
member( X, T ) :- gen( X, _, T ).
member( X, T ) :-
      gen( _, SubT, T ),
      member( X, SubT ).
```

This version of `member/2` is much more like the version we are used to. It encodes directly our intuition that `X` is a member of `T` if `T` was constructed with the generator function $\mathcal{F}_{\mathcal{X}}$, or else if it is a member of the single argument of `T`. But we see immediately that the definition of `member/2` is formally very odd. It refers to metalanguage relations, namely the relation of functional application implemented in the definition of `gen/3`, and so it itself becomes a metalanguage relation. We can't really consider it to be a relation on sequence terms at all. Membership is supposed to be a relation which holds between certain expressions—here the constants—and certain other expressions—here the sequence terms. But it holds, between `a` and `a(b([]))`, for example, only in virtue of the way in which the term `a(b([]))` was constructed.

Note that this is not true of the first version. There we explicitly stated that the membership relation was the relation which held between `a` and `a(_)`, between `b` and `b(_)`, and so forth. We could change the way we generate our terms, for example, without changing the membership relation in the least. That is a sure sign of a genuinely well-defined object language relation. But this code will obviously become quite large as soon as we try to work with lists of really interesting data items. Furthermore there is *no* way, even with a very great amount of typing, that we can handle sequences where $\Sigma$ is infinite. But this is exactly what we want to be able to do; we want to write code that can handle lists of arbitrary expressions, and there are indeed an infinite number of Prolog expressions (well, up to the limits of computer memory, anyway).

Logically, what's going on is that we can only define truly general relations over things that we can refer to with *variables*. It is precisely this ability to name unknown things that gives logic its power and expressiveness. But the rules say that in first order logic, and so *a fortiori* in definite clause logic, variables can only range over *individuals*. They cannot range over functions or relations (you only get to do that in second-order or higher). So we cannot construct Prolog expressions like

```
member( X, X(_) )
```

We need to take all of the entities that we want to make generalizations about (logically: "quantify over") and make them *individuals*.

In our case, we want to be able to quantify over things which appear as function symbols in our sequence terms. The following device can always be applied in this situation, where we want to quantify over function symbols. We "reify" (lit. "make into a Thing") the function symbols, and construct our terms with some more abstract function symbol, which stands for the raw abstract concept of function application. In symbols:

$$\texttt{f(a,b)} \Longrightarrow \texttt{apply(f,a,b).}$$

We have, of course, lots of different ways available to us for representing this notion of functional application. With sequences, we use the *cons* function—the list-constructor `[·|·]`—to represent the application of the monadic function symbol in the head to the single argument in the tail. So

$$\texttt{a(b([]))} \Longrightarrow \texttt{[a|[b|[]]] (= [a,b]).}$$

Now we can write programs which are as expressive as the "sloppy" version of `member/2` which called `gen/3`, and which avoid this slopiness; that is, we make no appeal in the standard definition of `member/2` to any sort of generator function. There is no need to quantify over function symbols here, since there is only one of them, namely *cons*.

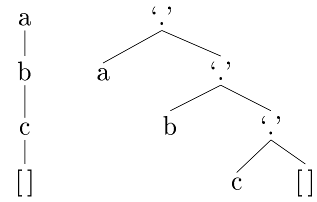



Figure 12.1: Two ways of representing 3.

So to be strictly accurate, we should have represented the (object) language of lists in the following way.

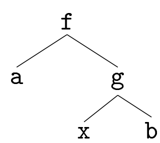
- *Basic* $= \{[]\}$,
- *Gen* $= \{\mathcal{F}_{cons}\}$,

where $\mathcal{F}_{cons} : (\sqcup_\infty, \sqcup_\in) \mapsto [\sqcup_\infty|\sqcup_\in]$.

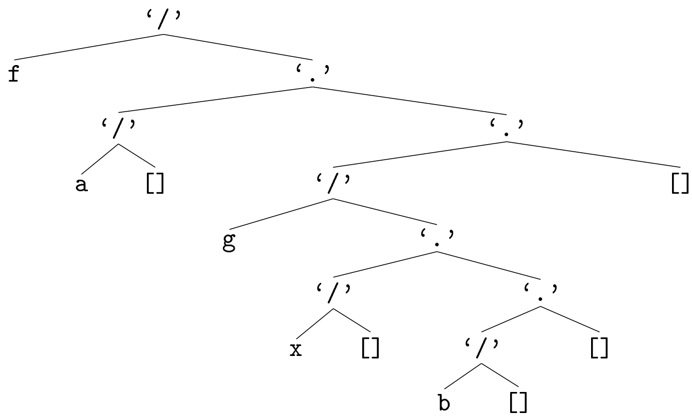
We could equally well have done the same thing with the natural numbers. That is, we could represent zero as `[]`, and the successor of `N` as `[s|N]`. (Recall your homework assignment using "list-of-1's" notation.) However, here again there is only one constructor function, and so no need to quantify over it. Put another way, it seems silly to treat a function symbol as an unknown when it is in fact very well known!

Consider the pair of expression trees in figure 12.1. Here we have a graphical representation of what is going on in reification. Intuitively, we gain computational leverage over the expressions of our language when we "push them down" from being labels of the interior nodes of an expression tree to labels of its leaves. With natural numbers we have no use for this extra leverage: we have only one function symbol to deal with. There are no generalizations to make about the function symbols in the language of successor terms! But with lists of arbitrary expressions, where we now have potentially an infinite number of lists even just of length 1, we need the extra leverage. More formally, the rules defining first order logic say that we can only talk about *terms*, that is, we can only talk about entire subtrees. So we can only talk about the functor of a term (i.e., the root of a subtree) by talking about the entire set of terms which share that functor (e.g., when we use `a(T)` to describe all of the sequences starting with `a`). If we want to store data items of some kind in a structure, we usually try to make these data items into leaves of that structure, because leaves are the only terms which are at once terms and single items.

We did the same thing with tree structures, in fact. That is, we intend the Prolog expression `f/[a/[],g/[x/[],b/[]]]` to represent the tree



but in fact this expression corresponds literally to the Prolog expression tree




Observe how the interior nodes of this expression tree are all labeled with one of the two general-purpose constructor functions '.' and '/'. All the real *content* of the original expression has been pushed down to the leaves.

Mathematically, trees are just terms, and terms are just trees. The functor of a term is the label on the root of the corresponding tree, and the label on the root of a tree is the functor of the corresponding term. So we are given, again, an alphabet $\Sigma$, but now with a difference. Whereas with sequences we treated the elements of the alphabet as being uniformly *monadic*, i.e., one-place functions, now we allow our symbols to take any number of arguments: they are *polyadic*. $\Sigma$ is thus what is often called a *ranked* alphabet, where the rank of a symbol is an indication of how many arguments it combines with. We might define trees, then as follows.

1. $Basic$ = the set of *primitive expressions*

2. $Gen = \{\mathcal{F}_{\sigma/n}\}$, where $\sigma$ is a primitive expression of arity $n$, and, given $n$ trees $t_1, \ldots, t_n$,

$$\mathcal{F}_{\sigma/n}(t_1, \ldots, t_n) = \sigma(t_1, \ldots, t_n).$$

Once again, however, if our main *computational* goal is to *store data in a tree structure*, we have much better access to and control over our data if it is reified and pushed down to the leaves of the tree. So we replace our infinite set of generator functions with a single function representing functional application.

- $Basic$ = the set of *primitive expressions*

- $Gen = \{\mathcal{F}_{`/'}\}$,

where, given an expression $t$ and $n$ expressions $t_1, \ldots, t_n$,

$$\mathcal{F}_{`/'}(t_1, \ldots, t_n) = t/[t_1, \ldots, t_n].$$

Reification is our introduction to dealing with the enforced separation of object-language expressions from metalanguage expressions. Assuming, as we will do from now on, that the metalanguage is also a logical language, in particular a first order language, then we have the following basic rule: The formulas of the object language are treated as terms in the metalanguage. This means that, from the point of view of the metalanguage, the separation in the object language between terms and formulas disappears. It has to be enforced *explicitly*; we have to build into our metaprogram the knowledge of what metalanguage terms count as object language terms and what metalanguage terms count as object language formulas.

Reification avoids some of the problems of dealing with two levels of language simultaneously, by pushing all interesting information into the object language. There is always a quasi-philosophical cost to be paid for this dirty trick, however. Reification means "making into a *thing*," and what we are doing with trees and lists, for example, is making *operations* into elementary *things*. In particular, we are no longer talking about the same structures. Nonetheless, there is usually a straightforward mapping from the structures which interpret our reified languages and the structures which interpret the languages we are really thinking about. (Consider, for example, the boolean-expression evaluator you built in your homework assignments: it assigned exactly the same value to the reified expression `and/[or/[true,false],not/[false]]` that a mathematician would assign to the 'pure' expression `and(or(true,false),not(false))`. Without really thinking about it, you were constructing just such a mapping between '/'-trees and the intended trees.)

/*

## 12.2 A Prolog-like Interpreter in the Ground Representation

We describe here a simple interpreter for solving definite goals against a definite clause program. The interpreter is essentially like Prolog. It implements a left-to-right scan of the goal and a top-to-bottom scan of the program. So like Prolog it implements a depth-first, left-to-right traversal of the search tree and, like Prolog, it is subject to termination problems and logical incompleteness when the search tree contains infinite branches. Also like Prolog its unification algorithm does not include an occurs check, so can give unsound solutions.

The treatment of substitutions implemented here is a little different from the way we have been handling them up til now. Up until now, we insisted that substitutions be in *solved form*, that is, if a variable appeared anywhere on the left of the '/', it would never appear anywhere on the right. Concretely this means that if we have a binding for `#3` as in `Theta = [ ... ,#3/f(a,b), ... ]`, then `Theta` would not also contain a binding like `#2/g(#3)`. The intuition is that, if this situation ever arose, we ought to have replaced `g(#3)` with `g(f(a,b))`, and if we did this systematically everywhere that `#3` appeared, then we could simply eliminate the `#3/f(a,b)` binding completely. In fact this procedure is known as *variable elimination*.

But variable elimination is a lot of work. Here we use instead a technique called *lazy evaluation*. The idea is this. Given the two bindings `#2/g(#3)` and `#3/f(a,b)`, we know all we need to know to construct the value of `#2` *when we absolutely need to do so*. We simply traverse the term that `#2` is bound to and, whenever we encounter a variable, we search the substitution for *its* binding, and continue our traversal. The assumption is that we will not actually need to know the full values of our variables very often, so we do not compute them until we have to. For example, we can tell that the unification problem `eq/[#2,h(#4)]` has no solution without knowing anything about the value of `#3`. Our implementation of lazy evaluation is not very sophisticated, but it is extremely simple, as you will see.

### 12.2.1 Preliminaries: The Object Language

The program begins with some syntactic sugaring. We define our usual variable marker `#` in the usual way, and we define an analogous constant marker `$` as well. We also define our own neck symbol, `<-`, which Prolog will treat as an infix predicate name with two arguments.

```
*/
:- ensure_loaded(library(lists)).

:- op(200,fx,#).
:- op(200,fx,$).
:- op(1200,xfx,<-).
/*
```

The object language is very simple. Expression trees are all of the form

`Label/SubExpressions`,

or else they are variables or constants. In particular this means that there is no support for infix operators. There is no special treatment for numerical constants, and no facility for using builtins. In particular this means that we cannot use Prolog list structures in the object language. We have to use an explicit *cons*-function. (See the definition of `list/1` in the sample program below.) Goals are either expression trees or lists of expression trees (or lists of lists of expression trees: goals can be arbitrarily nested).

```
*/
%==============================================
%
% Object Language Definitions

variable(#_).
```

```
constant($_).

gen(F/Args,F,Args).

goal([]).
goal([_|_]).
goal(G):- gen(G,_,_).
/*
```

The following is a sample program implementing some familiar relations in our new language. As you can see, the simplicity of our language makes writing programs rather awkward. However, it means that the number of cases we must consider in constructing the interpreter is vastly reduced! So we will begin here and consider later how to make our little programming language more "user friendly". Clauses are defined using the `<-` symbol, which takes an expression tree on its left (the head) and a Prolog list of expression trees on the right (the body). "Unit clauses" are written with an empty body.

Once again, keep in mind that the Prolog list structures use a functor ('.', alias [·|·]) which is *not available* in our object language. So lists here form metalanguage objects (either tuples of arguments or conjunctions of goals) which cannot be referred to in the object language.

```
*/
%==================================================
%
% A sample program

eq/[#1,#1] <- [].

list/[$0] <- [].
list/[c/[#1,#2]] <- [list/[#2]].

member/[#1,c/[#1,#2]] <- [].
member/[#1,c/[#2,#3]] <- [member/[#1,#3]].

append/[$0,#1,#1] <- [].
append/[c/[#1,#2],#3,c/[#1,#4]] <- [append/[#2,#3,#4]].
/*
```

Note that, while many pieces of Prolog syntax and Prolog builtin predicates are not available to us, we can still write any definite clause program! Any relation that can be defined with definite clauses can be defined in this langauge. This is all the syntax that we need; everything else is just to make programs easier to read and write.

### 12.2.2 The Interpreter Itself

Our main predicate, solve/2, is an interface to the true interpreter, solve/5. It actually contains a feature which is a little like a hack, namely it returns the entire substitution that the solver computed. We need it to return something, or else we will not be able to get Prolog to look for more than one solution. I chose to return State because it may be useful for debugging purposes, and there is a sense in which solve/2 is a relation between a goal and a state. But what we are really interested in is the "projection" of the state onto the variables which actually appeared in the original goal. Most of the work which solve/2 does is related to this problem, the problem of returning a decent-looking presentation of the solution we are really interested in.

```
*/
%===============================================\
%===============================================
%
% Solver Main Predicate


solve(Goal,State):-
        goal_vars(Goal,Vars,Max),
        solve(Goal,[],State,Max,_End),
        apply_tuple(State,Vars,Solutions),
        nl, print_solution(Vars,Solutions).


%
%===============================================
%==============================================/
/*
```

In pursuit of this goal, we first call goal_vars/3, which traverses the input goal and extracts all of the variables which appear in it, binding Vars to a list of them. It also computes the largest integer appearing in a variable name in the original goal, returning it in Max. We will need this later when we have to do "naming apart" of program clauses. Then we call solve/5, which we will describe in detail in a moment. It returns in State all of the variable bindings which have been made during the solution of the goal. We start off with the state being empty, []. The variable _End contains the largest variable name used in the solution. It is needed in recursive calls to solve/5, but when we are done we can just throw it away. It is just used for "book-keeping". Then we call apply_tuple/3 to actually compute the values of the variables appearing in the original goal. Finally, we call print_solutions/2 to print out, for each goal variable in Vars, its computed value in Solutions.

```
*/
```

```
%==============================================
%
% solve/5
%
% solve(
%    Goal,  /*the current goal (see goal/1);*/
%    State0,/*the state (i.e., substitution) coming in;*/
%    State, /*the state going out;*/
%    VName0,/*the next unused variable name (for naming apart);*/
%    VName  /*the last v-name we used, plus 1;*/
%    ):-
% true if Goal can be solved in State. The VName
% parameters are just carried around for use in
% renaming program clauses apart from the goal
% (see is_clause/4).

solve([],State,State,VNm,VNm).
solve([G1|Gs],State0,State,VNm0,VNm):-
        solve(G1,State0,State1,VNm0,VNm1),
%       be lazy,
        solve(Gs,State1,State,VNm1,VNm).
solve(G,State0,State,VNm0,VNm):-
        gen(G,_,_),
        is_clause(Head,Body,VNm0,VNm1),
        unify_term(G,Head,State0,State1),
%       be lazy,
        solve(Body,State1,State,VNm1,VNm).
/*
```

The main feature of the solver itself—`solve/5`, that is—is that there appears to be a step missing. If you recall the definition of the Resolution Rule from the beginning of the course, you may remember that when a subgoal is solved, the resulting substitution is supposed to be applied to the entire resulting goal. That step is missing here. (It would be inserted where the "`%be lazy`" comments appear.) Once again, if we ever need to know the current value of a given variable, we can trace through the network of variable bindings in the current state to find out. We will see an example of this when we examine the implementation of `unify_term/5` in a moment.

As you might expect by now, the structure of `solve/5` reflects directly the structure of goals, as defined in `goal/1`. The empty goal is always true and has no effect on the state. The non-empty conjunction of goals is true just in case the selected goal is true and the resulting resolvent is also true. Here, as with Prolog, the selected goal is always the leftmost (i.e., the head of the goal-list). Our depth-first strategy means that we will solve all of the subgoals derived from `G1` before we examine any goals in `Gs`. Solving `G1` extends the input state `State0` to the intermediate state `State1`, which is then input to the second call to `solve/5`. Similarly, we keep track of the variable names we have used in naming clauses apart so that the second call to `solve/5` does not use any variable names that have already been used.

The third clause handles atomic goals. It calls `is_clause/4` to search the program for a candidate clause. (It would be more natural to call it `clause`, but that is a Prolog builtin!) Naming apart is done in `is_clause/4`, so `Head` and `Body` come back as *variants* of the actual program clauses. Then `Head` is unified against the goal `G`. If this fails, Prolog will cause our interpreter to backtrack into `is_clause/4`. Maybe a better way to describe this is "We nondeterministically select a program clause whose head unifies with the selected goal `G`." Note that just because we are writing an interpreter for logic programs, it does not mean that our interpreter is somehow no longer itself a logic program. On the contrary! Our interpreter has a very nice clean declarative semantics, though we are describing it here in a rather procedural manner.

Once we have found a program clause whose head unifies with `G` (under the substitution which is the difference between `State1` and `State0`), we solve the body of the clause, under the new, extended substitution `State1`. Once again, we do not explicitly apply the mgu of `G` and `Head` to the `Body`, but wait until we are really sure we need to do so.

Now we examine the unification algorithm. This is where we must atone for our laziness. There are several cases to consider.

1. `T1` is a constant. Then either

   (a) `T2` is a constant identical to T1 or

   (b) `T2` is a variable.

   Clause 1 handles the former case, clause 3 the latter. (Clause 3 handles all cases where `T2` is a variable and `T1` is not.) Note that, because we are using the ground representation of programs, we can check the identity of `T1` and `T2` safely using unification. Even if `T2` is a variable—an *object language variable*, that is—it is still a *metalanguage* (i.e., Prolog) ground term. So, in particular, even though the goal `eq/[#1,$a]` always succeeds in our interpreter, because `#1` is a variable as far as the interpreter is concerned, the goal `#1 = $a` does not succeed anywhere. It does not succeed in our interpreter because there are no `<-` clauses defining `=`, and it does not succeed in Prolog because, well, those two terms just don't unify. (Their functors are different, in this case.)

2. `T1` is a variable. Be careful! Here's where we have to deal with our lazy evaluation strategy. In the standard, "eager" evaluation strategy, if `T1` had been bound to something already, it wouldn't be a variable any more; it would have been replaced by what it got bound to. But with our lazy strategy, where we do not explicitly apply the substitutions that we derive, it may be that there is already a binding for `T1` in `State0`! So, for example, `#1` cannot be unified with `f/[$a]` if it had already been bound to `f/[$b]`. So we have the following subcases to consider.

(a) T1 is identical to T2. We want to avoid uninformative bindings like #1/#1. So in this case we do nothing. Unification succeeds, but no new bindings are created.

(b) T1 is already bound to something, let's call it Term. Then our original problem of unifying T1 with T2 becomes a problem of unifying Term with T2.

(c) T1 is not bound to anything. So we are free to bind it to T2; we create the binding T1/T2 and add it to the state.

3. T1 is a compound term. As with constants there are two ways unification can succeed here.

   (a) T2 is also a compound term, with the same functor as T1. In this case we extract the argument-tuples of each term and attempt to unify their arguments.

   (b) T2 is a variable. As with constants this is handled by clause 3.

4. Finally we have the case where T2 is a variable. Here again there are two subcases to consider.

   (a) T1 is also a variable. Then this case has already been handled by clause 2. Unification should be deterministic, so we want to avoid this case. Therefore we put an extra guard on clause 3 which requires that T1 not be a variable. Note by the way that here again the ground representation of programs comes to our aid. Whatever T1 is in the object language, be it constant, variable or compound term, as far as Prolog is concerned it is a ground term. And so variable(T1) is a ground goal, and, as we know, negation-as-failure is sound as long as it is applied to ground goals. So no problems arise in our use of negation here. In particular, we are free to reorder the subgoals of this clause however we like; conjunction is commutative, just like it ought to be.

   (b) T1 is not a variable. Then we need to do exactly the same things we needed to do in the case where T1 was the variable. That is, we need to find out what, if anything, T2 is bound to currently. Because so much of the processing is the same in these two cases, we define a separate predicate, unify_var/4, and call it from both clause 2 and clause 3 of unify_term/4. The only difference is which of T1 and T2 gets to be the first argument.

That took a lot longer to say in English than in Prolog! Here's the code for our unification procedure.

```
*/
%================================================
%
% Unify:
```

```
unify_term(T1,T1,State,State):-
        constant(T1).
unify_term(T1,T2,State0,State):-
        variable(T1),
        unify_var(T1,T2,State0,State).
unify_term(T1,T2,State0,State):-
        variable(T2),
        \+variable(T1),
        unify_var(T2,T1,State0,State).
unify_term(T1,T2,State0,State):-
        gen(T1,F,Args1),
        gen(T2,F,Args2),
        unify_tuple(Args1,Args2,State0,State).

unify_tuple([],[],L,L).
unify_tuple([A1|A1s],[A2|A2s],L0,L):-
        unify_term(A1,A2,L0,L1),
        unify_tuple(A1s,A2s,L1,L).
/*
```

The predicate `unify_tuple`/4 is a predicate of a sort which should by now be extremely familiar: it just applies the `unify_term`/4 predicate to each pair of elements from the argument-tuples of `T1` and `T2` in turn.

The predicate `unify_var`/4 handles unification of an (apparent) variable `V` with another term `T` (possibly also a variable). Clause 1 handles the case where `T` is not only a variable but a variable identical to `V`. In this case we succeed without doing anything to the state. Note once again that we can use simple (Prolog!) unification to test for identity of object-language variables, because these are ground terms in Prolog, and there is thus no danger of accidentally instantiating a Prolog variable. Obviously if we were dealing with Prolog variables here, we could not test for true identity with unification: `X=Y` would succeed even though `X` and `Y` are different Prolog variables. Even worse, it would bind `Y` to `X`, even though this was not our intention. This accidental binding of (Prolog) variables is the chief problem with using a non-ground representation for object-language programs.

```
*/
%--------------------------------------------
% unify_var/4 is used because so much processing
% of variables is symmetrical: it is the same no
% matter which of the two terms to be unified
% is the variable.

unify_var(V,T,State0,State):- % x=x has no effect
        V = T,
        State = State0.
```

```
unify_var(V,T,State0,State):-
        \+ V = T, % this is not actually needed
        member(V/Val,State0),
        unify_term(Val,T,State0,State).
unify_var(V,T,State0,State):-
        \+(V = T),
        \+member(V/_,State0),
        State = [V/T|State0].
/*
```

Clause 3 of `unify_var`/4 handles the normal case of unification with a variable. This is the case where `V` is not already bound to anything, and we are free to create the binding `V/T` and add it to the state. Once again, we want to avoid uninformative reflexive bindings, so we add as a guard the condition that `V` and `T` be distinct terms.

These two cases are independent of our use of lazy evaluation; even if we were writing an eager system we would still need to avoid reflexive bindings. Clause 2 handles the case which is peculiar to our use of lazy evaluation techniques. In this case we find that there is already a binding for `V` in `State0`, namely `Val`. In that case we continue with our unification algorithm but this time with `Val` in place of `V`. That's all there is to it.

That completes our tour of the unification algorithm. The only remaining major subroutine of our interpreter is the predicate which fetches clauses from the program and names them apart.

```
*/
%================================================
%
% Renaming Apart:

is_clause(Head,Body,VNm0,VNm):-
        (Head0 <- Body0),
        rename(Head0,Head,VNm0,VNm1,[],Map1),
        rename_tuple(Body0,Body,VNm1,VNm,Map1,_).
/*
```

The predicate `is_clause`/4 first finds an instance of a program clause, that is, a clause in the (Prolog) definition of `<-`/2. Then it renames the head and body of the clause and passes the results back to the interpreter.

Here is where we need the `VNm0` and `VNm` arguments that the interpreter has been carefully passing around all this time. Recall that `VNm0` starts out being bound to the highest-numbered variable name appearing in the original goal. So variable names built from numbers larger than `VNm0` must be variables that have not yet been used in solving the goal. As long as we keep track of the numbers we have used, and are careful to output the largest such number, then we can be sure that whenever we need a brand-new, never-before-used variable name, we can construct it out of the current value of `VNm0` (plus one). `VNm0` and VNm are integers, but essentially they are just another accumulator pair. In particular they have the difference VNm-VNm0 as their semantic interpretation. That is, the difference VNm-VNm0 represents the set of variable names that were used in this call to is_clause/4. And VNm1-VNm0 represents the set of variable names that were used in the call to rename/6, while VNm-VNm1 represents the set of variable names that were used in the call to rename_tuple/6. In this way we can assure that every call to rename/6 uses a disjoint set of variable names.

The actual renaming works as follows. We construct a *renaming substitution*, that is, a substitution which only maps variables to variables, in the accumulator pair MapIn, MapOut. We do this in the following way. We simply traverse an object expression looking for variables. When we find one, we check to see if we already have a binding for it in the substitution we are building up. If so, we replace it with that value. If not, we create a new variable, by adding one to the current value of VNm0 (and sticking a '#' on the front), and add a binding to the state indicating that the variable we have found is to be replaced by the variable we just created.

So, we have the usual four cases to consider: TmIn is a constant (clause 1), or it is a compound term (clause 4) or it is a variable, in which case we need to check whether it is a variable we have seen before (clause 2) or not (clause 3). Only clause 3 does any real processing; that is where new variables are created. We take the number which VNm0 is bound to, add 1 to it, making the result the value of VNm and binding TmOut to #VNm.

```
*/
%----------------------------------------------
% rename/6:
%

rename(TmIn,TmOut,VNm0,VNm,MapIn,MapOut):-
        constant(TmIn),
        TmOut = TmIn,
        VNm = VNm0,
        MapOut = MapIn.
rename(TmIn,TmOut,VNm0,VNm,MapIn,MapOut):-
        variable(TmIn),
        member(TmIn/Trans,MapIn),
        TmOut = Trans,
        VNm = VNm0,
        MapOut = MapIn.
rename(TmIn,TmOut,VNm0,VNm,MapIn,MapOut):-
        variable(TmIn),
        \+member(TmIn/_,MapIn),
        VNm is VNm0+1,
        TmOut = #VNm,
        MapOut = [TmIn/TmOut|MapIn].
rename(TmIn,TmOut,VNm0,VNm,MapIn,MapOut):-
```

```
        gen(TmIn,Func,ArgsIn),
        gen(TmOut,Func,ArgsOut),
        rename_tuple(ArgsIn,ArgsOut,VNm0,VNm,MapIn,MapOut).

rename_tuple([],[],Name,Name,Map,Map).
rename_tuple([A1In|AsIn],[A1Out|AsOut],VNm0,VNm,Map0,Map):-
        rename(A1In,A1Out,VNm0,VNm1,Map0,Map1),
        rename_tuple(AsIn,AsOut,VNm1,VNm,Map1,Map).
/*
```

The rest of the program is essentially just utilities useful in making a nice presentation of the solution. The predicate `goal_vars`/3 extracts the variables appearing in the goal in `Vs`, and also returns the highest number that appeared in a goal variable in `Max`. It has the same three clauses that we find in the basic definition of goals in `goal`/1. In case the goal is an atom, `goal_vars`/3 calls `term_vars`/5. This in turn has the familiar four-clause structure, with two cases for variables depending on whether they are familiar (clause 2) or unfamiliar (clause 3). In the latter case we pull out the number from the variable name and test it against the current value of `M0`, returning the larger in `M`. Note the use of the Prolog builtin arithmetic function `max`. In this way, if we start with `M0` equal to 0, then we are guaranteed that, on exit from `goal_vars`/3, `Max` will contain the largest integer that we encountered.

```
*/
%===============================================
%
% Interface Utility Stuff


%-------------------------------------------
% goal_vars/3 extracts the variables appearing
% in the goal in Vs, and also returns the
% highest number that appeared in a goal variable
% in Max. This is used in the construction
% of fresh variable names in renaming apart.

goal_vars(G,Vs,Max):-
        goal_vars(G,[],Vs,0,Max).


goal_vars([],L,L,M,M).
goal_vars([G1|Gs],L0,L,M0,M):-
        goal_vars(G1,L0,L1,M0,M1),
        goal_vars(Gs,L1,L,M1,M).
goal_vars(G,L0,L,M0,M):-
        gen(G,_,Args),
        term_vars_star(Args,L0,L,M0,M).
```

```
term_vars(C,L,L,M,M):- constant(C).
term_vars(V,L,L,M,M):-
        variable(V),
        member(V,L).
term_vars(V,L,[V|L],M0,M):-
        variable(V),
        \+member(V,L),
        V = #N,
        M is max(N,M0).
term_vars(T,L0,L,M0,M):-
        gen(T,_,Args),
        term_vars_star(Args,L0,L,M0,M).

term_vars_star([],L,L,M,M).
term_vars_star([T1|Ts],L0,L,M0,M):-
        term_vars(T1,L0,L1,M0,M1),
        term_vars_star(Ts,L1,L,M1,M).
/*
```

By using lazy evaluation we have avoided the necessity of ever actually applying the substitutions that we have built up. But we don't want to be lazy about reporting our solution! So when we are done with `solve/5`, we call `apply_term/3` (actually `apply_tuple/3`) to apply the substitution which is the final value of `State` to just those variables that actually appeared in the original goal. (We assume that those are the only variables of interest to the user.) Once again, `apply_term/3` has the same old four-clause structure. Because we have been using lazy evaluation, when we find a variable `X` for which we have some binding in `State`, the problem of applying `State` to `X` is transformed into the problem of applying `State` to `Val`, the value of `X` in the current state. This is the same idea that we have already seen in the unification algorithm.

```
*/
%-------------------------------------------
% apply_term/3: apply a substitution to a term

apply_term(_State,X,X):-
        constant(X).
apply_term(State,X,XState):-
        variable(X),
        member(X/Val,State),
        apply_term(State,Val,XState).
apply_term(State,X,X):-
        variable(X),
        \+member(X/_,State).
apply_term(State,X,XState):-
        gen(X,F,Args),
```

```
        gen(XState,F,ArgsState),
        apply_tuple(State,Args,ArgsState).

apply_tuple(_,[],[]).
apply_tuple(State,[A1|As],[A1State|AsState]):-
        apply_term(State,A1,A1State),
        apply_tuple(State,As,AsState).
/*
```

Finally, `print_solution/2` handles the actual formatting and printing of "goal-variable = computed-value" pairs. This is just a simple simultaneous recursion on two lists.

```
*/
%-------------------------------------------
% print_solution/2 prints the goal variable names
% together with their final values.

print_solution([],[]).
print_solution([V|Vs],[T|Ts]):-
        write(V),write(' = '),write(T),nl,
        print_solution(Vs,Ts).

/*
```

That is all there is to it. There is a lot of code here, but by now you should have seen that the same ideas and the same program structures keep appearing over and over again, with slight variations. Though it was probably not so easy to work through the definitions of the first couple of predicate definitions, by now things should be starting to look familiar, and it should be easier to look at a predicate definition and grasp its overall structure pretty quickly.

We have not considered the non-ground representation of programs yet, in which we are allowed to use Prolog variables as object-language variables, so it is too soon to compare the two approaches to metaprogramming, and in any case many of the issues that to this day keep the logic programming community in constant debate fall outside the scope of this course. Nonetheless let me stress at this point the advantages of the ground representation which we can see already. First and foremost, the interpreter is clean. Though it uses a fair number of Prolog builtins, and the negation-as-failure operator, it does not step outside the bounds of pure SLDNF resolution. We used none of the truly nonlogical Prolog builtins nor did we use the cut operator, and we only used negation where we could be sure that the negated goal was ground. The fact that our interpreter has a clean declarative semantics means that it is much easier to apply things like automatic optimizers to it, for example. So: we have clean declarative semantics, we can use negation safely, and we can also use unification to test for identical variables, for example. The disadvantages will become clearer when we consider interpreters for non-ground object programs. But I should warn you right away: the community of logic programming researchers has not yet, after all these years, reached an agreement on which approach is better, so you should not permit yourselves any over-hasty judgments! */

# Chapter 13

# Prolog as an Assembler for Logic Programming Systems

## 13.1 Manipulating Prolog Terms

In this section we explore ways in which Prolog programs can directly manipulate Prolog terms, i.e., expression-trees. Up until now we have only been able to use unification to affect the form and content of terms. But consider the following simple problem: Write a predicate which takes an arbitrary term as input and returns the number of arguments of that term. At this point the only way we can come close to doing this is by defining a predicate like `arity`/2, below.

```
arity(f,0).
arity(f(_),1).
arity(f(_,_),2).
/* etc.  */
```

Clearly, we will have to stop after some definite number of such clauses have been written; but to solve the problem, even restricted to terms with functor `f`, we need an infinite number of such clauses. And it gets worse: there are an infinite number of possible functors for each of which we must write an infinite number of clauses! Clearly, we are asking too much of our fragment of first order logic.

Note that we can solve this problem for object languages other than Prolog itself. For example, if we define a language in which the functors have been reified to Prolog terms, as for example when we represent a logical term like $f(a, b, c)$ as the Prolog term `f/[a,b,c]`, then we can define `arity`/2 as

```
arity(_/Args,N):- length(Args,N).
```

The problem only arises when we want our object language and metalanguage to be the same, that is, when we want to use Prolog itself as our metalanguage.

Direct manipulation of Prolog terms is made possible by the use of three built-in predicates:

- `functor(Term,Functor,Arity)`,
- `arg(ArgNumber,Term,Argument)`, which returns in `Argument` the argument of term `Term` found in position `ArgNumber`,
- `Term =.. List`, which binds `List` to `[Functor,Arg1, ... , ArgN]`.

For the reasons noted above, these predicates do not have definitions in Prolog. Note, however, that "`=../2`" is interdefinable with the pair `functor/3` and `arg/3`. The code that implements each in terms of the other is a good example of their respective uses. Defining `functor`/3 and `arg`/3 in terms of `=..`/2 is simpler, so we begin there.

The predicate `functor`/3 is meant to operate in either of two modes: in the first mode, we are given a term and we wish to compute its functor and arity; in the second mode we are given a functor and an arity, and we wish to compute the term. The "univ" operator `=..`/2 also has two modes: it can be given the term so as to compute the list, or it can be given the list so as to compute the term. This introduces a little complexity into the definition of `functor`/3, as you can see:

```
functor(T,F,N):-
        (   atomic(F),
            integer(N) ->
            synthesize(F,N,T)
        ;
            nonvar(T) ->
            analyze(T,F,N)
        ).

analyze(T,F,N):-
        T =.. [F|Args],
        length(Args,N).

synthesize(F,N,T):-
        length(Args,N),
        T =.. [F|Args].
```

Recall that `->` (if-then-else) is defined with an implicit cut:

```
(A -> B ; C) :- A, !, B.
(A -> B ; C) :- C.
```

As a result, even when all three arguments are properly instantiated, only `synthesize`/3 will apply. The two sub-predicates `analyze`/3 and `synthesize`/3 differ only in the ordering of their subgoals. As you can see, once we are outside the bounds of definite clause logic multimodal behavior does not come so naturally anymore!

The predicate `arg/3` is also used in two modes, but they are not so different. That is, it only functions when the term argument is instantiated, but if a certain argument position contains a variable, it can be used to bind that variable to something else, in effect inserting a subterm in that position. For example:

```
?- T = f(A,B,C), arg(2,T,g(X)).
```

succeeds, binding `T` to `f(A,g(X),C)`. Since the term argument is bound to be instantiated to something definite, we do not have to struggle so hard to get this mulitmodal behavior out of our implementation of `arg/3`.

```
arg(N,T,X):-
        integer(N),
        nonvar(T),
        T =.. [_|Args],
        nth(N,Args,X).
```

Defining `=../2` in terms of `functor/3` and `arg/3` is more tedious, mainly because we have to explicitly transfer each argument from the term to the list or vice versa; there is no such thing as the “tail” of a term! Also, once again, we must have two different “front ends” to the predicate, depending on which mode it is called in.

```
(T =.. L):-
        (   nonvar(L) ->
            list2term(L,T)
        ;
            nonvar(T) ->
            term2list(T,L)
        ).

term2list(T,L):-
        functor(T,F,N),
        length(Args,N),
        L=[F|Args],
        transfer_args(N,T,Args).

list2term(L,T):-
        L = [F|Args],
        length(Args,N),
        functor(T,F,N),
        transfer_args(N,T,Args).

transfer_args(0,_,_).
transfer_args(N,T,L):-
```

```
        arg(N,T,X),
        nth(N,L,X),
        M is N-1,
        transfer_args(M,T,L).
```

Note however that code which uses functor/3 and arg/3 is usually more efficient, in spite of being more tedious to write. (Also the difficulty of writing such code is reduced when it is not required to run in more than one mode!)

Consider now the problem of implementing arrays in Prolog. There are essentially two approaches to this problem. One uses trees with fixed branching factor (e.g., binary trees), in which case modulus arithmetic can be used to convert an array index into a tree address. The other, which we will investigate here, uses Prolog terms as arrays, with the argument position corresponding to the index; that is, the $i$th element of the "array" `array(a1, ... ,aN)` will be the $i$th argument of the term. Clearly, in the simplest case, accessing an array element $A[37]$ is simply a matter of calling `arg/3` to find the 37th argument of the array. It is not so clear how to go about setting or resetting the contents of array cell number 37. We need some way of constructing a new term which is exactly like the old except that it has something different in its 37th argument position.

```
*/
empty_array(Size,Array):-
        functor(Array,$,Size).

get(Array,Index,Contents):-
        arg(Index,Array,Contents).

put(ArrayIn,Index,Contents,ArrayOut):-
        ArrayIn =.. [$|Cells],
        replace_nth(Index,Contents,Cells,CellsOut),
        ArrayOut =.. [$|CellsOut].

replace_nth(0,X,[_|Rest],[X|Rest]).
replace_nth(N,X,[Y|L1],[Y|L2]):-
        N>0,
        M is N-1,
        replace_nth(M,X,L1,L2).
/*
```

Note that updating these arrays is not spectacularly efficient with either version. Modifying array cell number `N` requires scanning `N` list positions, and copying them to the output one by one. On the other hand, making a copy of the remaining positions is much more efficient. It requires only the single unification binding the tail of the input list to the tail of the output list (clause 1 of `replace_nth/4`). In many modern Prologs, no new copy of `Rest` will be made, so the operation is particularly inexpensive. So we have this asymmetry between the list elements preceding the target cell and those following it. It would be nice if we could treat the initial cells in the same way as the later cells. The problem is based on the fact that lists are uniformly right-branching trees. They have very short left branches! So copying an entire left hand subtree doesn't do much for you: it is just copying a single list element. Whereas copying an entire right hand subtree does a lot: it copies the entire remainder of the list. So if we used trees which were more balanced than lists, we might be able to copy larger chunks of the array elements preceding the target cell as well as following it. What we give away then is the ability to get the contents of an array cell in a single, very simple operation, since now we will certainly have to do some tree walking to find it. So exactly how one should implement arrays in Prolog depends very much on the expected ratio of array accesses (calls to `get`) to array updates (calls to `put`).

A more sophisticated example of term manipulation is the following, which expands on the techniques that we used in the above example of array processing. Often we want to make multiple copies of a term which have the same pattern of variable-sharing *within* themselves, but which do not share any variables *among* themselves. This is required in naming apart, for example. A related—and fairly common—example is a breadth-first deductive engine, where each branch of the proof may instantiate a given goal differently; we want to explore all the branches at once, so all the different instantiations must coexist. In such a case what we actually need to make is a *fresh* copy, that is, a copy with the right pattern of variable-sharing, but with completely new variables, so that instantiating a variable in the copy leaves the original intact.

This is tricky to do! We have seen already plenty examples in which we construct or apply a variable substitution, but until now the "variables" have always been variables in some object language different from Prolog. As far as Prolog itself was concerned they were ground terms, and not (Prolog) variables at all. Here we concern ourselves with the "non-ground" representation of terms and programs, so now our substitutions are lists pairing—in this case—Prolog variables with Prolog variables. Searching such a list is an extremely delicate operation, because we cannot always use unification. For example, suppose we have the substitution `[X/X1,Y/Y1,Z/Z1]`, and we wish to find out what to replace an occurrence of `Y` with. If we just call something like

```
member(Y/YVal,[X/X1,Y/Y1,Z/Z1])
```

we will have the disastrous result of succeeding immediately, binding `X` to `Y`! Prolog provides a built-in predicate `==/2` for testing "meta-identity". It is a much stronger test than unifiability (i.e., `=/2`). It succeeds just in case its two arguments really are the same *syntactic* object. In the case of variables, it succeeds just in case they have the same name. For example:

```
?- X == X. % succeeds
?- X == Y. % fails
?- X == Y, X = Y. % still fails
?- X = Y, X == Y. % succeeds
```

By now it should not be surprising that the order of goals matters here. It is not normally possible in a logic to describe relations which refer to syntactic facts like variable names, so clearly ==/2 may show behavior that is incompatible with the expected behavior of classical logical conjunction.

Using these techniques, we can now construct a predicate `vmember/2`, for searching substitutions in the non-ground representation, and using `vmember/2` we can construct our term-copying program.

```
copy(X,Y):-
      Substitution0 = [],
      copy(X,Y,Substitution0,_Substitution).

copy(X,Y,Subst,Subst):-
      var(X),
      vmember(X/Z,Subst),
      Y = Z.
copy(X,Y,Subst0,Subst):-
      var(X),
      \+ vmember(X/_,Subst0),
      Subst = [X/_|Subst0].
copy(X,Y,Subst0,Subst) :-
      nonvar(X),
      functor(X,F,Arity), % Atoms and numbers have arity=0.
      functor(Y,F,Arity),
      transfer_args(Arity,X,Y,Subst0,Subst).

transfer_args(0,_,_,Subst,Subst).
transfer_args(N,X,Y,Subst0,Subst) :-
      N > 0,
      arg(N,X,Arg_N),
      copy(Arg_N,Arg_N_Copy,Subst0,Subst1),
      arg(N,Y,Arg_N_Copy),
      M is N-1,
      transfer_args(M,X,Y,Subst1,Subst).

vmember(X/Y,[X1/Y1|_]):-
      X == X1,
      Y = Y1.
vmember(X/Y,[_|L]):-
      vmember(X/Y,L).
```

/*

## 13.2 A meta-circular interpreter for programs in the non-ground representation

The example of making a fresh copy of a term illustrates the complexities of working with a non-ground object language. However, it does have its advantages. The following is an interpreter for programs written in an object language which is identical to Prolog (as usual, the metalanguage). Conceptually, this is a rather complex and tricky situation, and mistakes are easy to make. However, one sees immediately that if one is willing to invest the required care, the effort can result in metaprograms which are considerably more compact than their cousins in ground representation.

```
*/
%
% solve.pl
%

:- dynamic eq/2,member/2,append/3,solve/1,solve/2.

solve((G1,Gs)):-
        solve(G1),
        solve(Gs).
solve(\+(G)):-
        cut_fail(G).
solve(A):-
        \+ A=(_,_),
        \+ A=(\+_),
        predicate_property(A,built_in),
        A.
solve(A):-
        predicate_property(A,dynamic),
        clause(A,Body),
        solve(Body).

cut_fail(G):- solve(G),!,fail.
cut_fail(_).
/*
```

That's all the code there is in this implementation!

Note first the expression

```
:- dynamic member/2,append/3,solve/1,solve/2.
```

The symbol `dynamic` is defined as a prefix operator, so there are no parentheses needed here. This ability to declare predicates either *dynamic* or *static* is built into both Quintus and SICStus Prolog, but is not otherwise a general feature of Prologs. It is required here because the builtin predicate `clause/2` only works for predicates which have been declared to be `dynamic`. Normally, no such declaration is required; the builtin `clause`/2 (which *is* a general feature of Prolog implementations) should work with any user-defined clause. A call to `clause(Goal,Body)` succeeds when `Goal` is bound to an atomic goal and Body is the body of a program clause whose head matches `Goal`, under the resulting substitution. (The "body" of a unit clause is by definition the 0-place predicate `true`/0, which always succeeds.) So a single call to `clause`/2 takes the place of almost all of our ground-representation interpreter! All the problems of naming apart, unification and application (or lazy evaluation) are taken care of internal to Prolog.

The SICStus-specific builtin predicate `predicate_property`/2 is used to distinguish predicates of various types. I believe that all Prologs have some equivalent predicate, but it is not always called by the same name. The reason we use it here is that `clause`/2 only works for user-defined clauses. Builtin predicates do not (usually) have definite-clause definitions. It may be possible to describe them using definite clauses, but, as part of the Prolog interpreter, they have generally been implemented at some much lower level.

The interpreter works as follows. Goals are defined as being either Prolog terms (i.e., atomic goals), or else conjunctions of goals. Here conjunction is represented by a ',' operator. Recall that this is the representation we have always been using for conjunctions. So the bodies of your program clauses have always been legitimate Prolog expression trees whose roots are labeled with ',', an infix operator. For example, the body of the recursive clause of `append`/3 might look like

```
                           ','
               ____________|____________
              |                         |
As = [X|AsTl]                          ','
                              __________|__________
                             |                     |
                     Cs = [X|CsTl]     append(AsTl,Bs,CsTl)
```

Note that, unlike ordinary lists, these "round lists" do not have a distinguished terminating element like the empty list. Note also that this comma, which is a legitimate infix operator (with binding strength 1000 and associativity `xfy`, if you really want to know), is to be distinguished from the comma which separates the arguments of a term. That comma is just punctuation, like the parentheses or the period.

So, goals can either be atomic or they can be of the form `(G1,G2)` where `G1` and `G2` are also goals. There is no reason that non-atomic goals cannot be nested, as in

```
?- (X=Y,Y=a),Z=f(X).
```

But without parentheses a goal like

```
?- X=Y, Y=a, Z=f(X).
```

will always be treated as

```
?- (X=Y, (Y=a, Z=f(X))).
```

So `solve/1` has three clauses. The first handles conjunctions, the second handles Prolog builtins, by calling them directly, using Prolog's metavariable facility, and the third handles user-defined predicates.

The non-ground representation of programs, that is, the writing of object level programs in Prolog itself leads to a very particular and rather confusing kind of metaprogramming. The version of `solve/1` given here can solve itself, for example. That is, it is perfectly capable of correctly handling goals like

```
?- solve( solve(member(b,[a,b])) ).
```

*/

## 13.3 A Very Simple Compiler

In this section we examine another sort of metaprogram, namely one which takes a program as input and translates it into another program. In this case, we will take a "program" written in the format of a context-free phrase structure grammar and translate it into a Prolog program which can then be used to recognize the strings which are in the language generated by the input grammar.

We have already seen that there is a very close correspondence between the production rules of a context-free grammar and definite clauses. For example, a rule like

$$S \longrightarrow NP\ \ VP$$

corresponds to a definite clause of the form

```
s(L0,L):-
      np(L0,L1),
      vp(L1,L).
```

This clause states that the string which is the difference between `L0` and `L` belongs to category `s` if its first part can be categorized as an `np` and its second part can be categorized as a `vp`. Likewise, a rule expanding a nonterminal symbol into a terminal symbol, like

$$NP \longrightarrow John$$

corresponds to a definite clause of the form

```
np(L0,L):- L0 = [john|L].
```

Then a string `L` is generated by a grammar just in case the goal

```
?- s(L,[]).
```

succeeds.

```
*/
%
% cfdcg.pl
%

:- op(1200,xfx,==>).
/*
```

We begin by defining an operator '==>'/2, which we will use as our production arrow (the more obvious choice '-->' is already taken by Prolog). Grammar rules will then look like

```
s ==> np,vp.
```

in the case of expansion into nonterminal categories, and, in the case of expansion into terminal elements,

```
np ==> [john].
```

Note that anything which marks terminal elements as different from nonterminals would work here. We use this strategy because that is how Prolog's own compiler for "Definite Clause Grammars" represents terminal elements. Here is a complete grammar in this format. /*

```
*/
%
% g1.pl
%

s ==> np,vp.
s ==> np,aux,vp.

np ==> det,n.
np ==> [john].
np ==> [mary].
np ==> [he].
np ==> [him].
np ==> [she].
np ==> [her].

det ==> [the].
det ==> [a].
det ==> [every].
det ==> [some].

n ==> [lawyer].
n ==> [politician].
n ==> [dog].
```

```
aux ==> [will].
aux ==> [can].
aux ==> [wont].

vp ==> v.
vp ==> v,np.
vp ==> v,s.

v ==> [thinks].
v ==> [see].
v ==> [sees].
v ==> [think].
v ==> [sneeze].
v ==> [sneezes].

/*
```

*/

This grammar generates strings like

*john sneezes*

and

*some dog thinks every politician will see mary sneeze*

among (too many) others.

Note that, once we define '`==>`' as an operator of the appropriate sort, this grammar is a legitimate Prolog program, and can be loaded into Prolog with the usual commands for consulting or compiling programs. However, it cannot be "run" as is. One thing we could do would be to write an interpreter which could call a subgoal like

```
(LeftSide ==> RightSide)
```

to nondeterministically select instances of the '`==>`'/2 predicate for further processing.

Here, however, we will do something different. We will tell Prolog to consult the grammar file in a special way, so that when it reads in an expression whose primary functor is `==>`/2, it will translate that expression into a normal Prolog clause (i.e., an expression whose primary functor is `:-`/2), so that we can then "run ther grammar" directly, without an interpreter.

SICStus Prolog provides a special predicate called `expand_term`/2, which is called on every term in a file when that file is either consulted or compiled. The first thing that `expand_term`/2 does is to look and see whether the user has defined a special "hook" predicate called `term_expansion`/2. If any clauses for `term_expansion`/2 exist, then these are called and their output becomes the output of expand_term/2. Otherwise, expand_term/2 does nothing. So the simplest way to construct a compiler for programs in an object language of our own design is to provide a definition for term_expansion/2 which does the right thing.

```
/*
term_expansion((A ==> Bs),Clause):-
        cat2atom(A,L0,L,Head),
        Clause = (Head:- Body),
        build_body(Bs,Body,L0,L).
/*
```

Here we have defined term_expansion/2 so that it will recognize grammar rules as they are read in from a file. (Note that the file can contain ordinary Prolog clauses as well: if expand_term/2 calls term_expansion/2 and it fails, then expand_term/2 just defaults to its normal behavior.) Each such rule is translated to a definite clause `Clause` with head `Head` and body `Body`. The head is constructed by a call to cat2atom/4 and the body is constructed by build_body/4, both of which we will define below.

The essential problem is twofold. We must construct Prolog atoms out of category symbols, by making the category symbol the main functor and adding two argument positions, and then we must fill the argument positions with the appropriate difference-list arguments. That is, we have to make sure that, whenever two categories are adjacent in the right hand side of a rule, then the last argument of the first category must be identical to the first argument of the second category. In this way we enforce the condition that the second category begin where the first one ends.

The predicate cat2atom/4 expects to be given the proper list arguments when it is called. All it does is attach them to the category symbol to make a Prolog term.

```
*/
cat2atom(Category,L0,L,Atom):-
        functor(Atom,Category,2),
        arg(1,Atom,L0),
        arg(2,Atom,L).
% or
%       Head =.. [A,L0,L].
/*
```

Then build_body/4 just calls cat2atom/4 for every category in the right hand side of the grammar rule, making sure to keep proper track of the difference lists.

```
*/
build_body([Word],BAtom,L0,L):-
        BAtom = (L0=[Word|L]).
build_body(B,BAtom,L0,L):-
```

```
        \+ B = (_,_),
        cat2atom(B,L0,L,BAtom).
build_body((B1,Bs),(B1Atom,Rest),L0,L):-
        cat2atom(B1,L0,L1,B1Atom),
        build_body(Bs,Rest,L1,L).
/*
```

The first clause handles terminal productions. If the right hand side of a rule is a terminal element, that is, a word enclosed in square brackets, then it is converted into an atom with functor =/2 which will act to consume a word from the input string.

The second clause is the base clause for the case where the right hand side of the rule is a sequence of nonterminal symbols. Note that the right hand sides of rules are "round lists" like the bodies of Prolog clauses and goals in general. There is no way to recognize the end of a round list except by the negated goal `\+B=(_,_)`, so we put this in as a guard on clause 2. In this case we simply call `cat2atom`/4 one last time to return the last subgoal of the body we are building. Finally, in the recursive clause we call `cat2atom`/4 on the first category in the rule's right hand side, passing the output difference list argument `L1` to the recursive call to `build_body`/4 so that it will be used as the *input* difference list argument when the next body-atom is built.

Once we consult this code (which I have named `cfdcg.pl`), all subsequent calls to `consult` or `compile` will make use of the newly defined clauses for `term_expansion`/2. So, given the grammar `g1.pl` defined above, the following goal

```
?- [cfdcg],[g1].
```

(or just "`?-[cfdcg,g1].`" for short) will cause `g1.pl` to be translated into Prolog clauses. For example, after issuing the above goal, if you then issue the goal

```
?- listing.
```

you will see something like the following.

```
v(A, B) :-
        A=[thinks|B].
v(A, B) :-
        A=[see|B].
v(A, B) :-
        A=[sees|B].
v(A, B) :-
        A=[think|B].
v(A, B) :-
        A=[sneeze|B].
v(A, B) :-
        A=[sneezes|B].
```

```
s(A, B) :-
        np(A, C),
        vp(C, B).
s(A, B) :-
        np(A, D),
        aux(D, C),
        vp(C, B).

n(A, B) :-
        A=[lawyer|B].
n(A, B) :-
        A=[politician|B].
n(A, B) :-
        A=[dog|B].

np(A, B) :-
        det(A, C),
        n(C, B).
np(A, B) :-
        A=[john|B].
np(A, B) :-
        A=[mary|B].
np(A, B) :-
        A=[he|B].
np(A, B) :-
        A=[him|B].
np(A, B) :-
        A=[she|B].
np(A, B) :-
        A=[her|B].

det(A, B) :-
        A=[the|B].
det(A, B) :-
        A=[a|B].
det(A, B) :-
        A=[every|B].
det(A, B) :-
        A=[some|B].

aux(A, B) :-
        A=[will|B].
aux(A, B) :-
        A=[can|B].
aux(A, B) :-
```

```
        A=[wont|B].

vp(A, B) :-
        v(A, B).
vp(A, B) :-
        v(A, C),
        np(C, B).
vp(A, B) :-
        v(A, C),
        s(C, B).

   */
```

# Bibliography

[1] Ivan Bratko (1986). *Prolog: Programming for Artificial Intelligence.* Addison-Wesley.

[2] Ivan Bratko (1986). *Prolog: Programmierung für Künstliche Intelligenz.* Addison-Wesley.

[3] William F. Clocksin and Christopher S. Mellish (1987). *Programming in Prolog.* 3rd edition. Springer-Verlag.

[4] Esther König and Roland Seiffert (1989). *Grundkurs PROLOG für Linguisten.* Tübingen: Francke Verlag.

[5] Fernando C. N. Pereira and Stuart M. Shieber (1987). *Prolog and Natural-Language Analysis.* CSLI/Univ. of Chicago Press.

[6] Leon Sterling and Ehud Shapiro (1986). *The Art of Prolog.* London: MIT Press.

[7] Robert Kowalski (1979). *Logic for Problem Solving.* Amsterdam: North-Holland.

[8] John W. Lloyd (1993). *Foundations of Logic Programming.* 2nd ed. Heidelberg: Springer-Verlag.

[9] Richard A. O'Keefe (1990). *The Craft of Prolog.* London: MIT Press.