# Programming in Tabled Prolog (very) DRAFT [1]

*David S. Warren*

Department of Computer Science
SUNY @ Stony Brook
Stony Brook, NY 11794-4400, U.S.A.

July 21, 1999

[1]This is a very early draft made available privately for those who might find it of interest. 

# Contents

# Chapter 1

# Background and Motivation

There is a flaw in the very foundations of Logic Programming: Prolog is nondeclarative. Of course, everyone knows that real Prolog as it is used is nondeclarative. Prolog programs abound with cuts, var tests, asserts, and all kinds of ugly warts. Everyone agrees that Prolog should be more declarative and much research time has been spent, and spent productively, trying to remedy these problems by providing more declarative alternatives. But I believe that there is a more serious flaw closer to the heart of Prolog, right at its very center, and this flaw has caused a number of problems in logic programming. I'm convinced that we must address this flaw, before we can proceed together productively with logic programming. Of course, whether this is a "flaw" is in the eye of the beholder; we all have arguments over whether a particular behavior is a bug or a feature. As is clear from my presentation, I think I've found a bug; some of you will think it's a feature. Wherever we stand on this issue, however, I believe we must be clear about its implications. Only then will we be able to communicate effectively about where LP is and where it should go.

Let me also mention that I am by no means pointing out something new, unknown to the community; rather I'm trying to raise our consciousness about an inconsistency and its implications.

The problem is the very semantics of logic programming. The foundational results of our field show that for pure horn clauses, the following are equivalent: logical implication, SLD resolution, and the least fixpoint of the $T_p$ operator. And logic programming basically "works" because SLD corresponds to computation (i.e. Prolog) *and* to logical implication (truth in models).

But there is a fly in the ointment, the dirty secret that we Prolog aficionados try to keep in the closet but Prolog programmers are all too aware of. Let's look more carefully at the full SLD tree, the foundation for us Prolog hackers. There are three kinds of paths in an SLD tree: 1) those ending in success, 2) those ending in failure, and 3) those not ending. The foundational results tell us that those ending in success correspond exactly to those instances of the query that logically follow from the program. This is fine and dandy and certainly something we want and need. But as programmers, that's not enough. We programmers also want, need actually, to distinguish between paths that end in failure and paths that don't end at all. When we run a program, and it responds 'no', we may accept that response as a good answer to our query. But if it goes into an infinite loop never coming back with an answer at all, we decide we have a bug in our program and set out to fix it.

So the problem with SLD is not that it doesn't do the right things for success, but that it doesn't do the right thing with the other two possible outcomes. Note that this has nothing to do with how we search the SLD tree, e.g. unfairly with depth first search or fairly with breadth-first search; it's a property of the tree itself. And it has nothing to do with the fact that our theory deals best with ground programs and Prolog programs deal with variables. The problem concerns which paths in the SLD tree are finite and which are infinite.

In Prolog, as compared with other languages, it seems easier to write partially correct programs, programs that, if they halt, they give the right answers. This comes from the power of declarative programming. But it seems much harder to write and reason about totally correct programs, programs that halt in all (the right) cases. So it may be the case that what we gain from declarativeness in our ability to reason about the partial correctness of Prolog programs, we lose in the difficulty of reasoning about total correctness. Theoreticians may accept partial correctness as good enough; but we users have to know precisely how partial.

And sadly, the problems show up for very simple programs. The issue is clearly seen in transitive closure. Consider the definition:

```
tca(X,Y) :- a(X,Y).
tca(X,Y) :- a(X,Z), tca(Z,Y).
```

Prolog programmers know that this definition is fine when the predicate "a" stands for "parent", and then "tca" is "ancestor". But what happens if "a" is defined by the following facts?

```
a(1,1).
a(2,1).
```

Consider what happens when we ask the query: :- a(1,2). Prolog (that is SLD resolution) goes into an infinite loop. Are we as programmers happy? Do we say that since it hasn't returned, it must mean that you can't get to 2 from 1 in this graph? Not at all. We turn back to fix the program, probably modifying it by adding a loop-check. That is, we add a list of nodes to the predicate "tca", which contains the nodes already visited. Then before continuing to search from a node, we check that it has not already been visited. We're not happy until our program responds 'no' to our query. So clearly the semantics of our programming language is going to have to account for this. And it does. We can't depend on the previously referenced foundational results to give a semantics for pure Prolog as a programming language, where we want to know what both 'yes' and 'no' answers mean. The theory must somehow be extended. This was done by Clark who defined the completion of a program. The semantics of (pure) Prolog programs is normally taken to be the logical implications of the completion (plus some standard axioms such as the unique names axiom.) To find the completion of our tca/2 program, we first convert the two implication rules for tca/2 into the equivalent one implication with a disjunction in the antecedent, and then we turn the implication into a biconditional. We do a similar transformation for a/2. Then we take the meaning of the program to be the logical implications of these biconditionals (including the standard axioms). And it is the case that $\sim a(1,2)$ is not logically implied by this theory (and neither is a(1,2), of course.) So the completion semantics correctly tells us that SLD will not terminate for this query. And indeed Prolog theoreticians (as well as the programmers) know that tca/2 as defined above is NOT a general definition for transitive closure. (But note that it *is* transitive closure if the graph is acyclic. Maybe this is OK, since it is often the case that algorithms for constrained data are simpler, but here it could be rather confusing, I think.)

But in any case the Prolog community was (mostly) happy with this situation, until the database people showed up and asked the obvious question: Why doesn't the obvious two-line definition of tca/2 indeed mean transitive closure? (To some of us, it felt somewhat as though they were pointing out that the emperor had no clothes.) They pointed out that if one uses the fixed point characterization from the original foundation trilogy, instead of the SLD characterization, it does. Indeed, transitive closure is only the beginning; there are many advantages in using this semantics. *All* programs without function symbols are terminating, and many simple programs now have a natural programmers' semantics. Indeed DCG's now *do* mean the corresponding context free languages, whereas under the completion semantics they don't always. But even more important is what this fixpoint semantics does to the theory of programs. The theory is much simpler, and accords with theory already developed. E.g., automata theory (e.g. CFL's) is now correctly reflected in LP. And also interestingly, this semantics leads the way for a theory of default negation, which has blossomed. This is because if we have a better idea of when things are known false, the theory describing such negative knowledge is simpler, more interesting, and perhaps even more useful.

Actually, the situation for the completion is somewhat worse than implied by the transitive closure example. There are programs for which Prolog goes into an infinite loop even though the completion of the program determines all goals. And this is *not* because Prolog is depth-first; it's because its selection rule is left-to-right. To get Prolog actually to display the behavior predicted by the completion, one would have to interleave computations of all goal orderings, and fail a goal if any of the interleavings failed. This is hardly an attractive alternative for Prolog implementors or programmers.

So I personally think minimal model semantics (i.e., the fixpoint semantics) is better than the completion semantics for logic programming. I think the completion semantics is a bug, which should be fixed, and it should be fixed by taking the fixpoint semantics. There are those who either don't think this is a bug, or don't think that this is a reasonable fix. I certainly accept that the point is arguable. Going to the fixpoint semantics is a big step for at least two reasons:

Objection 1: It violates a principle dear to the hearts of some theoretically inclined logic programmers, that the meaning of a program consists of the logical implications of the theory of the formulas making up the program (or a simple transformation of them).

Objection 2: It violates a principle dear to the hearts of some practically inclined logic programmers (like me), that a program should have the well-known procedural semantics, i.e., be executable by SLD resolution (and WAM engines).

The first objection is real and in some sense is not satisfactorily solvable, I believe. We know that there is no first-order theory that characterizes transitive closure (the usual Prolog loop-check program requires default negation.) I will discuss some implications of this later, and try to claim that things aren't so bad, but I can't say they aren't true. Basically, the purpose of this entire book is to claim that the second objection can be overcome. It is overcome by using OLDT resolution in place of SLD resolution in the foundational theorems. OLDT is a resolution strategy very similar to SLD, but one that avoids redundant computation by remembering subcomputations and reusing their results to respond to later requests. This remembering and reuse has been called memoization, tabling, lemmatization, caching, well-formed substring tables, and probably a number of other names. With OLDT instead of SLD, the foundational theorem can be even stronger, including a theorem relating OLDT failure leaves and the fixpoint definition.

OLDT terminates (i.e., has only success and failure leaf nodes) for all queries to programs that have finite minimal models, i.e. that have a least fixpoint of finite size. This guarantees total correctness for the transitive closure definition (over finite graphs). Actually, even more can be said: OLDT terminates for all queries to programs for which only a finite portion of the minimal model is looked at; that is, for all programs with the bounded term size property (BTS). BTS is slightly circular, and undecidable, but to me intuitively says that any finitary, constructive definition will terminate. (Admittedly, this is also somewhat circular.)

One thing that happens when we do not have first-order theories to explain our semantics, i.e. the first objection above, is that we may leave the realm of r.e. sets. That is, if we did have a first-order theory to explain the semantics, then we'd know that a theorem-prover could enumerate the theorems and the semantics would be r.e. However, by going to the minimal model semantics, we may (and do, when we add negation) get non-r.e. sets. That is, we leave the realm of the computable sets. This might give pause to someone interested in computation, because then we can write programs that aren't in principle computable. Note that this isn't a problem for totally correct programs (which will be recursive programs), which are those we usually want to write. I guess that just as we rely on the programmer to write programs that are terminating, we must rely on the programmer to write programs that are computable.

SLDNF was the obvious (and perhaps only reasonable) extension of SLD to include default negation. If we are to base our new logic programming on OLDT, the question arises as to how OLDT should be extended to handle negation. Given the theory of default negation developed over the last several years, it seems that the well-founded semantics is the appropriate semantics to compute for logic programs with unconstrained negation. (The other possible semantics is the partial stable model semantics, but this is NP complete for propositional programs, so may be not particularly appropriate for a basic computational mechanism.) For OLDT there have been several proposals for computing the well-founded semantics, in particular WELL! and XOLDTNF, which are quite similar. The one I'm proposing here is SLG, which I claim is the (or a) right one. The simplest reason that SLG is better than the previous two is that they are exponential for general propositional normal programs, whereas SLG is polynomial. Also SLG produces a residual program when there are undefined atoms in the well-founded models, and this residual program can be further processed to find certain partial stable models. So in many cases SLG could be used as a preprocessor for a partial stable model evaluator.

SLG implements the procedural interpretation of Horn clauses, with the addition of tabling. This is a more complex procedural interpretation than SLDNF (and the procedural interpretation of negation makes it even more complex), but I think it is indeed a procedural interpretation and I think it is acceptable. But this is not enough to respond fully to objection 2 above. The Prolog programmers want to know if SLG can be implemented so that it is efficient enough to compete with Prolog, (or actually efficient enough to compete with C++, perhaps our *true* competitor.) I think the answer is 'yes'. Actually, even more than that, I think it will be the case that we will be able to construct an engine that for many useful problems will be significantly faster than Prolog. And these problems will be in what has historically been logic programming's core applications: natural language, AI search, and databases. The examples in this book will indicate in detail where some of the problems lie. There is much work to do, but I will claim that results we already have in the XSB project indicate that it will definitely be possible.

The approach is to take SLG as the basic computation strategy, and to use SLDNF as an optimization. So when SLDNF can be proved to have the same behavior as SLG, then we can use it and get current Prolog speeds. To do this we need an implementation that fully and closely integrates SLG and SLD, allowing them to be used interchangably and intermixedly. And that is what SLG resolution supports and its implementation in XSB achieves.

# Chapter 2

# Introduction to Prolog

This chapter introduces the Prolog programming language. Here we will explain a bit of how Prolog works. It is not intended to be a full description of how to become an expert Prolog programmer. For that, after reading this chapter, you should refer to another book, such as *Programming in Prolog* by Clocksin and Mellish, or if you are already very familiar with Prolog, you might look at *The Craft of Prolog* by Richard O'Keefe. While this is an introduction to Prolog, even experts may find something of interest in this chapter, since I explain Prolog in a somewhat unusual way, that may give new insights to old Prolog programmers.

Prolog's name is short for "Programming in Logic" (or really for Programmation Logique?") As its name suggests, Prolog is firmly based on logic, and Prolog programs can be understood as statements in a formal logic. I.e., a Prolog program can be thought of as a set of statements in first-order logic, and the meaning of the program is the set of true implications of those logical statements. This is the approach that is usually taken to describe Prolog programming to novices. However, the amazing thing about logic programming to me is not that it is logic, but that it is programming. These Prolog programs are not only statements in a logic but they are also statements in a programming language. This is referred to in the biz by saying that Prolog programs have a procedural interpretation (i.e., as programs) as well as a declarative interpretation (i.e., as statements in a logic.) The introduction to Prolog that I give here will emphasize its procedurality. This may be anathema to some Prolog purists (and it certainly would have been to me a while ago) but I now feel that this is the best way to introduce logic programming to computer scientists who already know about programming. We will build on your understanding of programming, and use that to lead to logic.

## 2.1 Prolog as a Procedural Programming Language

Prolog, as a programming language, is a little unusual. It can be understood as a standard procedural language with two unusual properties. It is a procedural language like Pascal or Algol. One programs in a procedural language by writing procedures that carry out particular operations. One specifies what a procedure is to do by using primitive statements and by invoking other procedures. Prolog procedures have only local variables and all the information that a procedure can use or produce must be passed through its arguments.

C can be viewed as a procedural language by thinking of using it without functions; i.e., all functions return void, and information is passed to and from functions through their arguments only.

In Prolog, procedures are called predicates. The two unusual aspects of Prolog are:

1. Prolog has assign-once variables, and
2. Prolog is nondeterministic.

### 2.1.1 Assign-once Variables

By saying that Prolog has assign-once variables, I mean that any particular variable in a Prolog procedure can only ever get one value assigned to it. A Prolog variable at any point in execution either has a value, which can thereafter never be changed, or it has not yet been given a value. This may seem to be an incredibly strong restriction. In Pascal, for instance, one usually programs by setting a variable's value and then changing it during the execution of the program. For example, to sum up an array of numbers, one sets the accumulator variable to 0 and the loop variable to 1, and then increments the loop variable to step through the array modifying the accumulator at each step. How in the world could this be done with assign-once variables? The secret is that it can be done easily through the use of recursion.

So let's write a simple Prolog program to see how we can do something interesting with only assign-once variables. Let's consider the problem of adding up the numbers in a list. Prolog is a list-processing language, similar to Lisp. Its primary data structure is the tree (called a term), a very common form of which is the list. There are three basic data types in Prolog: 1) integers, 2) floating point numbers, and 3) atoms. The first two should be self-explanatory. Atoms are simply symbols that represent themselves. Prolog terms (trees) are constituted of integers, floats, atoms and other terms.

A list in Prolog is written with square brackets and with its elements separated by commas. For example the following are lists:

```
[1,2,3]   [aa,bbb,d]   []   [[2,b],or,not,[2,b]]
```

The first is a list of integers, the second a list of atoms, the third is the empty list consisting of no elements, and the fourth is a list containing four elements, the first and last being themselves lists.

So let's now write a program in a made-up procedural language (that is *not* Prolog, but somewhat similar) and see if we can sum up the elements of an integer list with assign-once variables.

```
sum(List,Sum) :-
    if List = []
      then Sum := 0
      else Head := head(List)
           Tail := tail(List)
           sum(Tail,TailSum)
           Sum := TailSum + Head
```

The first line (preceding the `:-`) declares the name of the procedure and its formal parameters. The remaining lines (following the `:-`) make up the body of the procedure definition. We have assumed the existence of two functions, head and tail, to extract the first element of a list, and the remainder of the list after the head is removed, respectively. Sum is a recursive procedure. It takes a list of numbers as its first argument and returns the sum of the numbers in its second. It first checks to see if the list is empty. If so, it sets the sum to 0 and returns directly. If not, it saves the first element of the list in a local variable, `Head`, and then calls `sum` recursively on the tail of the list, getting the sum of the rest of the list in the local variable `TailSum`. It then adds `Head` to `TailSum` to get the value for `Sum`, which it sets and returns. Notice that no single variable gets two different values. The variable `Sum` is not set and then changed; each recursive invocation has a different `Sum` variable and each gets set only once, to its appropriate partial sum. Note also that the loop variable in the iterative version of summing an array is here replaced by the variables containing each sublist. So here too there is no need to have multiply assigned variables. Instead of one variable getting many values, we can instead uses many variables, each getting one value. (Let me point out for those of you who may be worried about efficiency that this is a conceptual point; it may well be that the underlying implementation of such a program would actually use just one location for all the `Sum` variables.)

So we see that we are able get by in this case with assign-once variables. It turns out that this idea of using recursion and the multiple variables at the different recursion levels is very general. This is not just a trick that works in this case only, but is an example of a very general technique.

Now having assign-once variables gives rise to a very interesting phenomenon: assignment can be symmetrical in Prolog. That is, Prolog doesn't have to treat the left and the right sides of an assignment differently, as must be done in normal procedural languages such as Pascal or C. As a matter of fact, Prolog doesn't have to treat tests and assignments differently either. I.e., Prolog doesn't need two operators, say == for testing and = for assignment as C does; it needs only one.

Let's first consider assignment. Consider the following assignments:

```
X := 5
Y := X
```

We'll assume that neither X nor Y have been assigned a value before this sequence is executed. So X gets the value 5 by the first statement, and then Y is assigned the value of X, so Y gets the value 5 as well. Now consider the following statements:

```
X := 5
X := Y
```

The first statement again assigns 5 to X. Now consider the second. X has the value 5 and Y has no value. Since Prolog is an assign-once language, X can get only one value and it already has it, so we know we can't change it. But Y doesn't yet have a value. So the only reasonable thing to do is to set Y to be 5, i.e., to X's value. Note that this sequence of assignments has the same net effect that the previous sequence had.

This suggests how we can treat both sides of an assignment in the same way. If one of the variables has a value and the other doesn't, then assign the value that the one has to the other. If neither variable has a value yet, then make it so that whenever one of them gets a value, the other gets that same value. If they both have a value, then if it's the same value, then the assignment is a no-op. If the two values are different, then there is a problem since neither can get a new value. In this case we say the computation fails. (We will talk more about failure later.)

Notice that this definition of "assignment" means that any ordering of the same (or symmetric) assignments gives the same result. For example, consider the different ordering of our assignments above:

```
X := Y
X := 5
```

Again assuming that X and Y start with no values, the first statement causes Prolog to bind X and Y together, so that whenever one of them gets a value, the other will also get that same value. Then the second statement causes X to get the value 5, and so Y gets that value, too. So after these two assignments are executed, both X and Y have the value 5, exactly as they do after the previous two versions of these assignments. This is also a very general phenonemon: with this meaning of assignment, any ordering of any set of assignments gives the same result.

So let's rewrite our sum program with these ideas in mind. We will use = for our symmetric assignment statement. (From now on, all our programs will be syntactically correct Prolog, and XSB, programs, so you can type them into XSB and try them out. [sidebar] to explain how to create files, consult them, and run defined predicates).

```
sum(List,Sum) :-
    List = []
     ->    Sum = 0
     ;     List = [Head|Tail],
```

```
            sum(Tail,TailSum),
            Sum is TailSum + Head.
```

I've changed the syntax for if-then-else to Prolog's syntax, using `->` and `;`. Here we've said that `Sum = 0`; using the properties of symmetric assignment, we could just as well have said that `0 = Sum`. Consider the symmetric assignment: `List = [Head|Tail]`. The structure on the right is how one constructs a list from a head element and a tail list. (In Lisp it is known as `cons`.) So our symmetric assignment here is even more powerful. We know that the variable `List` has a list as its value. So this assignment assigns both variables `Head` and `Tail` so that they get the values of the first element of `List` and the tail of `List`, respectively. We can see that symmetric assignment is here extended to matching. We match the value in the variable `List`, which is a list, to the structure `[Head|Tail]`. `Head` and `Tail` are the only variables without values, so the symmetric assignment will fill them in with the appropriate values to make the `List` and the `[Head|Tail]` structure the same. This matching process, which we have been referring to as "symmetric assignment", is called *unification*.

Notice that we've used the same operation of unification, `List = []`, in the test of the if-then-else. Here we see a use of failure. Recall that we said that if a symmetric assignment cannot be made because the two sides have values that are different, then the assignment (or unification) fails. The if-then-else construct does the unification and if it succeeds, then it executes the then statement (which follows the `->`); if it fails, it executes the else statement (which follows the `;`.) So even the boolean test of an if-then-else can use our universal unification operation.

Notice, however, that we have not used unification for the last statement that adds Head to the partial sum. This is because here we don't want to match the two sides, since Prolog considers the right side to be a tree (with + at the root and with two leaves of TailSum and Head.) So here we must use an operation that explicitly asks for the tree on the right to be evaluated as a numeric expression. In Prolog that operation is named `is`.

As another example, consider the append procedure:

```
append(L1,L2,L3) :-
    L1 = []
     ->  L3 = L2
     ;   L1 = [X|L1t],
         append(L1t,L2,L3t),
         L3 = [X|L3t].
```

This is a procedure that takes two lists and concatenates them together, returning the resulting list in its third argument. This definition says that if the first list is empty, then the result of concatenating `L1` and `L2` is just `L2`. Otherwise, we let `X` be the head of the first list and `L1t` be its tail. Then we concatenate `L1t` and `L2`, using append recursively, to get `L3t`. Finally we add `X` to the beginning of `L3t` to construct the final result, `L3`.

Consider the following version of append:

```
append(L1,L2,L3) :-
    L1 = [X|L1t]
     ->  L3 = [X|L3t],
         append(L1t,L2,L3t)
     ;   L3 = L2
```

This one looks rather strange, but it also works. We've used the boolean test unification also to deconstruct the list. (This is probably a poor idea in real Prolog programming.) The other perhaps stranger (but less bad) difference is that we've moved the construction of the output list `L3` to before the recursive call to `append`. You might wonder how we can construct a list before we have its components. But with unification, that works just fine. The intuition is that if a variable can get only one value, it doesn't really matter when it gets it. So it is often the case that unifications can be moved earlier. What happens here is that the list cell is constructed before the call to `append`, and then the recursive call to `append` will fill in the tail of that cell with the appropriate value.

We've looked at assign-once variables and seen how they lead to symmetric assignment, which leads to unification. Next let's consider the other unusal property of Prolog programs, the fact that they can be nondeterministic.

### 2.1.2 Nondeterminism

Pascal is a deterministic programming language (as are C and Algol); at any point in the execution of a Pascal program there is exactly one next step. Prolog, however, is nondeterministic. There are points in the execution of a Prolog program when there are multiple legal next steps. The way this is specified in Prolog is to give multiple definitions of the same procedure. For example, we could write a procedure to find both square roots of a positive real number by:

```
a_sqrt(X,Y) :-
    X > 0,
    Y is sqrt(X).
a_sqrt(X,Y) :-
    X > 0,
    Y is -sqrt(X).
```

`A_sqrt` takes a number and returns its square root. Here we want it to return both square roots, one positive and one negative. We can do that by giving two definitions of a procedure `a_sqrt`. A Pascal compiler would complain that the procedure is multiply defined, but Prolog accepts such multiple procedure definitions happily. The first definition checks that the input argument is greater than 0, and if so uses a Prolog primitive builtin to calculate the positive square root. The second definition does the same, but returns the negation of the positive square root. In Prolog terminology, each definition is called a "clause", so `a_sqrt` is defined by two clauses.

### Prolog execution as the execution of multiple machines

The way to understand how Prolog handles multiple procedure definitions is first to think of how a deterministic procedural machine executes a procedural program. It maintains a state (usually a stack of activation records) and executes instructions which update that state, calling subprocedures, performing the indicated operations, and returning from subprocedures. To introduce nondeterminism into this picture, we consider what happens when a machine encounters a procedure that has multiple definitions. At this point it duplicates itself, creating a copy for each definition, and each copy continues by executing the definition it is assigned to. Recall that an execution may fail when it does a unification that discloses an inconsistency. When this happens, we can think of the machine as simply disappearing. So we can think of a Prolog execution as a set of executing deterministic machines; whenever any one of them encounters a procedure call of a multiply defined procedure, it forks into multiple machines; whenever a machine fails, it disappears out of existence. The answer to a Prolog program is the set of answers returned by all the individual machines that make it to the final instruction of the program, i.e., that return successfully from the initial procedure call.

So if we invoke the `a_sqrt` procedure defined above with the following procedure call statement:

```
:- a_sqrt(13,Y).
```

we will get two answers:

```
X = 3.6055;
X = -3.6055;
```

Let's revisit the append program we wrote above. Instead of using an if-then-else construct there, we can now use nondeterminism. Consider:

```
append(L1,L2,L3) :-
    L1 = [],
    L3 = L2.
append(L1,L2,L3) :-
    L1 = [X|L1t],
    append(L1t,L2,L3t),
    L3 = [X|L3t].
```

Notice that for whatever the list that is assigned to the variable `L1`, exactly one of the two procedure definitions will fail, and the other will succeed. The first will succeed only if `L1` is the empty list, and the second will succeed only if `L1` is a nonempty list. This program is essentially equivalent to the one above with the if-then-else.

Actually we can now improve this new append program. Consider how a normal procedural programming language passes parameters to procedures. One way is to do it by assignment: local variables are allocated for each formal parameter and the actual parameters are assigned to local variables on invocation. So assignment is used for passing parameters. Prolog can pass parameters this way as well, but instead of using assignment, it uses its symmetric assignment operation, unification (or matching.) So rather than doing a unification in the body of a procedure definition, we can simply put the values to be unified in the place of the formal parameters. So, for example, in the first procedure definition for `append`, rather than assigning the actual parameter to a local variable `L1` and then checking it against the empty list, we can directly check the first argument against the empty list as follows:

```
append([],L2,L3) :-
    L3 = L2.
```

This looks very odd for a conventional procedural language, having a constant in the place of a formal parameter, but with Prolog's symmetric assignment, it works fine. It simply unifies the empty list with the first actual parameter at the time of invocation.

As a matter of fact, whenever we have an explicit unification of a variable with another term, we can replace all occurrences of the variable with the term and eliminate the explicit unification. So we can replace L3 by `L2` in the above clause and we get simply:

```
append([],L2,L2).
```

(When a definition has no body operations, we don't even write the :-.) This procedure definition has no operations in its body. In a traditional procedural language, it would be a no-op, but in Prolog it actually does some work through the unification of its arguments when it is invoked.

The same idea for eliminating explicit unifications can be used on the second clause for `append`, and we obtain the usual Prolog definition of `append`:

```
append([],L2,L2).
append([X|L1t],L2,[X|L3t]) :-
    append(L1t,L2,L3t).
```

### 2.1.3 Executing Programs in XSB

Now, we can load this definition into XSB and then call it in various ways to experiment with how it works. So we put this definition into a file called, say, appendfile.P. (The '.P' suffix indicates to XSB that this is a file containing source code.) We run XSB and then compile the file and load it into XSB by:

```
% xsb
```

```
XSB Version 1.4.1 (94/11/21)
[sequential, single word, optimal mode]
| ?- [appendfile].
[Compiling ./appendfile]
[appendfile compiled, cpu time used: 0.901 seconds]
[appendfile loaded]

yes
| ?-
```

The XSB system top-level prompt is '`| ?- `', which is printed when XSB is waiting for the user to enter something. Here we've entered the filename in a list. This requests the system to compile and load the indicated file, which the system then does. The compilation creates an object file, in this case named appendfile.O. Then the XSB loader is called to load that file into XSB's space. (If the last-change date of the object file is more recent than the last-change date of the source file, then the compiler is not called, but the object file is loaded.) So now we have the append program in XSB memory and we can ask XSB to execute it. We do this by entering a call at the top-level prompt, as follows:

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

X = [a,b,c,d,e]
```

XSB calls the append procedure and executes it, passing the two lists in and when append returns, X has been assigned the answer, which XSB prints. It's possible that there is more than one answer (as would be the case with `a_sqrt` above), so XSB waits to let the user ask for another answer, if desired. To request another answer, the user enters a ';', to which XSB responds with the next answer, if any. Here the result is as follows:

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

X = [a,b,c,d,e];

no
| ?-
```

XSB has responded with 'no' and then returned with the top level prompt. This is because, in this case, there is just one answer so asking for another results in the response of 'no'.

We could, of course, ask for different invocations of append, giving it different lists, but we can also give different forms of invocations. The unification of Prolog allows us to call some procedures in perhaps a surprising variety of different ways.

For example, we can enter the following query (i.e., procedure invocation) and will get the indicated result from XSB:

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

yes
| ?-
```

Here we've given the answer. XSB simply verifies that the answer is correct, and indicates it is by responding 'yes'. In this execution, unifications that set variable values in the previous execution simply verify that the variables already have the correct values. If the values don't check out, as they won't in this following case:

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

no
| ?-
```

XSB gives a response of 'no' indicating that the first two arguments do *not* concatenate to form the third.

Actually, Prolog can respond to even stranger invocations of our append procedure. Consider the following invocation:

```
| ?- append(X,Y,[a,b,c]).
```

Here we are asking for what two values will concatenate together to form the list [a,b,c]. The tokens beginning with capital letters, `X` and `Y`, are variables, and we are asking the system to fill them in with correct values. (A variable starts with an upper-case letter, and an atom starts with a lower-case letter. We've been using this convention all along, and it is important to know.)

Prolog can answer this query reasonably. There are four possible pairs of lists that do concatenate together to produce `[a,b,c]`, and Prolog will produce them:

```
| ?- append(X,Y,[a,b,c]).

X = []
Y = [a,b,c];

X = [a]
Y = [b,c];

X = [a,b]
Y = [c];

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

```
Y = [];

no
| ?-
```

Here XSB produced the first answer and then waited for my response. I responded with a `;`, and it responded by producing the next answer. We continued until all the answers were produced. Since Prolog is nondeterministic, queries that have multiple correct answers are reasonable to ask. In this case Prolog answers the query correctly and reasonably.

Let's consider another simple (and well known) Prolog program known as `member`. `Member` is a binary predicate, i.e., it is a procedure with two arguments. It is given an element and a list, and it checks to see whether the element is a member of the list:

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

The first clause says that `X` is a member of a list whose head is `X`, an unexceptional statement. The second clause `X` is a member of a list if `X` is a member of the tail of the list.

Example executions of `member` are:

```
| ?- member(2,[1,2,3]).

yes
| ?- member(2,[1,3,4]).

no
| ?- member(X,[1,2,3]).

X = 1;

X = 2;

X = 3;

no
| ?-
```

Notice that we can use `member` to generate all the elements of a list.

(Aside: If you tried to compile this `member` program exactly as it is written here, you noticed that the XSB compiler issued some warning messages. The first message says that the variable `L` in the first clause appears only once in that clause. Such a variable is called an *anonymous variable*. An anonymous variable is just a placeholder, since the value that it might get is never used anywhere else, because the variable appears nowhere else. In such cases you are encouraged to use a slightly different notation: instead of starting anonymous variables with upper-case letters, start them with underscores (_), or simply use an underscore alone. Each occurrence of the underscore symbol is a distinct (unnamed) variable. The compiler will not complain if you begin the name of an anonymous variable with an underscore. I strongly suggest following this convention; it will save untold grief that you'll otherwise suffer when you mistype variable names. So an equivalent, but syntactically improved, version of `member` is:

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

End of aside.)

As a final example of a simple Prolog list-processing predicate, consider the procedure `reverse`, which is given a list and returns a list that contains the elements of the input list, but in the reverse order.

```
reverse([],[]).
reverse([X|L],R) :-
    reverse(L,RL),
    append(RL,[X],R).
```

The first clause says that if the input list is empty, then the resulting list is also the empty list. The second clause says that if the input list has head `X` and tail `L`, then first reverse the tail `L` of the input list obtaining `RL`, and then add `X` to the end of `RL`. The predicate `append` is used to add this element to the end of the reversed sublist. Notice that we must use `[X]` as the second argument of append, not just `X`, because `append` requires a list there, not an element.

An example of executing `reverse` is:

```
| ?- reverse([1,2,3],R).

R = [3,2,1];

no
| ?-
```

exactly as expected. You might reasonably think that we should also be able to ask the following query:

```
| ?- reverse(X,[1,2,3]).

X = [3,2,1];
```

And it looks as though everything works fine. However, what has really happened is that after the system produced the expected answer, I asked it for another answer. It should have simply said "no", but instead it went into an infinite loop and didn't respond at all. To understand why Prolog behaves like this, we have to understand more clearly exactly how it goes about evaluating queries.

[add an example to introduce ; and disjunction.]

### 2.1.4 The Scheduling of Machine Execution in Prolog

Recall that above we described a conceptual model of how Prolog executes nondeterministic programs by talking of a growing and shrinking set of deterministic procedural machines. In order to completely understand how Prolog executes these nondeterministic programs, we must look more carefully at how this growing and shrinking set of machines is actually executed. Clearly Prolog is not executed by actually have a set of hardware machines that grows and shrinks. (While it would be nice, the physics of such machines has not yet been worked out.) Instead these multiple machines must somehow be simulated, or emulated, by a single hardware machine having just one processor.

The Prolog engine keeps track of the states of these multiple machines and uses them to simulate the machine executions. Let's first consider the way Prolog keeps track of the state of a single machine. We can model execution of a single machine by "expanding" procedures. When a procedure is called, the actual parameters are matched with the formal parameters. All variables that get values in the matching process, occurrences in the body of the procedure being called and variables in the calling procedure, are replaced by those values. And the procedure call is replaced by the body of the procedure. The explanation is complex but the idea is simple: procedure calls are replaced by procedure bodies, with the variables appropriately set. For example.....

Now that we have seen how a single machine executes, the real question is in what order does it emulate the multiple machines. Clearly when a query first starts, there is just one machine to execute it. What happens when that machine encounters a procedure defined by multiple clauses? At that point there are several machines to be executed. In what order does Prolog execute them? I.e., how are they scheduled?

====================

The formal counterpart of Prolog execution is the SLD search tree. Each node in the search tree corresponds to a state of one of the machines. Each path through the search tree corresponds to the execution sequence of one of the machines. A branching node in the tree corresponds to a choice point, when a machine is duplicated to create instances of itself that will explore the various alternatives.

Let's look at a simple example to see how this works for the append program when it is called with a final list and it is asked to find all pairs of lists that concatenate to form the given list.

Consider the query:

```
| ?- append(X,Y,[a,b]).
```

First we have to determine a way to represent the states of the individual procedural machines. The state of execution of a procedural program is normally kept by a current instruction pointer and a stack of activation records associated with active procedures, which indicate values of local variables and also where to return when the procedure exits. For our Prolog programs we will use a very abstract representation that will enable us to understand the machine's operations without getting lost in encoding details. We will keep an instance of the variables of the original query and the sequence of procedure calls that remain to be done to complete the machine's computation. So the state of the initial machine is:

```
answer(X,Y) :- append(X,Y,[a,b]).
```

We use the `:-` to separate the query variables, which are grouped by `answer`, from the sequence of procedure calls that remain to be done. Initially the only thing to be done is the initial procedure call. We move a machine to its next state by taking the first procedure call after the `:-` in the machine state, matching it against the head of the chosen clause that defines the procedure, and replacing that call by the body of the clause, with the variables appropriately updated. Thus one step of computation replaces a procedure call by the sequence of procedure calls that make of the body of its definition, with the variables appropriately updated according to the parameter passing method.

So let's now consider the execution of the above query. The execution will be a tree of machine states, with the above machine state at the root. This tree is shown in Figure 2.1.

Prolog explores this tree in a top-down left-to-right order. The left-to-right order of children of a node corresponds to the order in which clauses for the called procedure appear in the text of the program. This is implemented in the Prolog engine by maintaining a stack of alternatives; whenever new alternative computation states are generated, it pushes them onto the stack, and whenever it needs another alternative, it takes the top one from the stack. So the Prolog engine begins by taking the first (and only) state off the stack and matching the first procedure, `append(X,Y,[a,b])`, with the heads of the appropriate procedure definitions (i.e., clauses). For each one that matches, it pushes a new state on the stack, replacing the procedure call by the body of the procedure, updating the variables accordingly. Now this first procedure call matches both clauses, so we generate two new states as children of the root state:

```
answer([],[a,b]) :- .
answer([a|L1ta],L2a) :- append(L1ta,L2a,[b]).
```

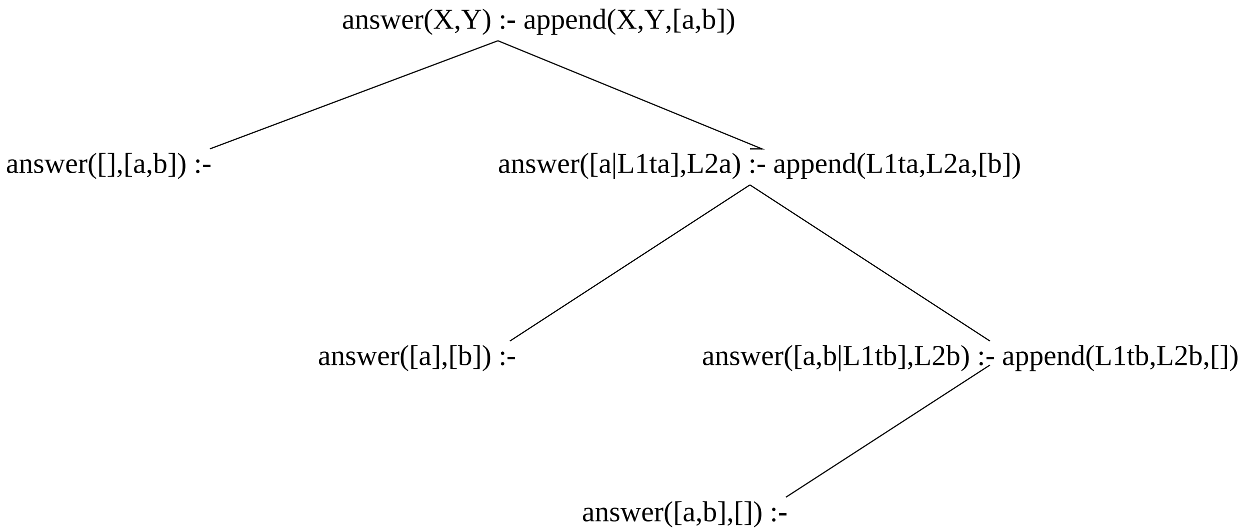


Figure 2.1: SLD tree for the query: append(X,Y,[a,b]).

The second state comes from the second clause, and the procedure call is replaced by the single procedure call in the body of the second clause defining `append`. The first state comes from the first clause, which has no procedure call in its body, so this state has no procedure call to do, and thus is a successful *final state* of the Prolog program. The arguments of the answer template contain the values of the variables of the original query, and so constitute the final answer of this machine. Here they are `[]` and `[a,b]`, which do indeed concatenate to generate `[a,b]`, as expected. Prolog will print this answer out, remove this state from the stack and continue expanding the next state on the top of the stack, here the second child of the root node.

Now consider that state:

```
answer([a|L1ta],L2a) :- append(L1ta,L2a,[b]).
```

It was generated by matching the original procedure call with the second clause for `append`. In a procedural language, whenever a procedure is called, the procedure gets a *new* set of local variables, and in Prolog it is the same. I've indicated that here by giving the variables in the clause new names, by adding 'a' to the end of their original names. Each time I take a clause, I'll have to rename the variables to new ones, so we don't have unintended collisions, and I'll do this by adding a new letter suffix.

Again Prolog expands this state by replacing the first procedure call by the bodies of matching clauses. Again both clauses for `append` match this one so we get two new states on the stack:

```
answer([a],[b]) :- .
answer([a,b|L1tb],L2b) :- append(L1tb,L2b,[]).
```

The top one is again an answer, since it has no procedures left to call, and its values for the result variables are: [a] and [b], which again do concatenate to form the desired [a,b].

After the answer is printed and the state removed from the stack, the next state:

```
answer([a,b|L1tb],L2b) :- append(L1tb,L2b,[]).
```

is expanded. Now this procedure call matches only the first of the append clauses; the second fails to match because the third argument in the call is [] but in the procedure head is [X|L3t]. So the new stack is just:

```
answer([a,b],[]) :- .
```

The top state is an answer, which gets popped and displayed, and then the stack is empty, indicating that Prolog has completely finished traversing the SLD tree and thus with evaluating the query. It has simulated all the deterministic procedural machines to completion.

Stepping back a bit and thinking about the SLD tree, we can quite easily describe the tree by giving an operation that can be applied to a subtree to extend it. Then we can define the SLD tree as the result of applying this operation to an initial (trivial) tree, and all resulting trees until no operation is applicable. This operation is called Program Clause Resolution.

**Definition 2.1.1 (Program Clause Resolution)** Given a tree with a node labeled $A : -A_1, A_2, \ldots, A_n$, and a rule in the program of the form $H : -B_1, B_2, \ldots, B_k$, and given that $H$ and $B_1$ match with matching variable assignment $\theta$, then add a new node as a child of this one and label it with $(A : -B_1, B_2, \ldots, B_k, A_2, \ldots, A_n)\theta$, if it does not already have a child so labeled. Note that the matching variable assignment is applied to all the goals in the new label. □

Notice that the entire tree of Figure 2.1 is developed by applying this rule to the trivial tree consisting of the single node answer(X,Y) :- append(X,Y,[a,b]). So we can think of Prolog as applying this Program Clause Resolution rule over and over again (in a top-down backtracking manner) to the initial query to trace out the SLD tree.

The example we have used here has a relatively simple execution and Prolog executions can get considerably more complex, but all the basics have been illustrated.

## 2.2 Grammars in Prolog

Next we turn to more complex examples of Prolog programming. Prolog was originally invented as a programming language in which to write natural language applications, and thus Prolog is a very elegant language for expressing grammars. Prolog even has a builtin syntax especially created for writing grammars. It is often said that with Prolog one gets a builtin parser for free. In this section we will see why this claim is made (and sometimes contested).

Consider the following simple context-free grammar for a small fragment of English.

$S \longrightarrow NP\, VP$
$NP \longrightarrow Det\ N$
$VP \longrightarrow TV\ NP$
$VP \longrightarrow V$
$Det \longrightarrow the$
$Det \longrightarrow a$
$Det \longrightarrow every$
$N \longrightarrow man$
$N \longrightarrow woman$
$N \longrightarrow park$
$TV \longrightarrow loves$
$TV \longrightarrow likes$
$V \longrightarrow walks$

In this grammar we can derive such simple sentences as:

a man loves the woman
every woman walks
a woman likes the park

We can write a simple Prolog program to recognize this language, by writing a recursive descent parser. We first must decide how to handle the input string. We will use a list in Prolog. For each nonterminal we will construct a Prolog procedure to recognize strings generated by that nonterminal. Each procedure will have two arguments. The first will be an input parameter consisting of the list representing the input string. The second will be an output argument, and will be set by the procedure to the remainder of the input string after an initial segment matched by the nonterminal has been removed. An example will help clarify how this works. The procedure for `np` would, for example, take as its first argument a list `[a,woman,loves,a,man]` and would return in its second argument the list `[loves,a,man]`. The segment removed by the procedure, `[a,woman]`, is an NP. The Prolog program is:

```
s(S0,S) :- np(S0,S1), vp(S1,S).
np(S0,S) :- det(S0,S1), n(S1,S).
vp(S0,S) :- tv(S0,S1), np(S1,S).
vp(S0,S) :- v(S0,S).
det(S0,S) :- S0=[the|S].
det(S0,S) :- S0=[a|S].
det(S0,S) :- S0=[every|S].
```

```
n(S0,S) :- S0=[man|S].
n(S0,S) :- S0=[woman|S].
n(S0,S) :- S0=[park|S].
tv(S0,S) :- S0=[loves|S].
tv(S0,S) :- S0=[likes|S].
v(S0,S) :- S0=[walks|S].
```

The first clause defines procedure `s`, for recognizing sentences. An input list `S0` is passed into procedure `s`, and it must set `S` to be the remainder of the list `S` after a sentence has been removed from the beginning. To do that, it uses two subprocedures: it first calls `np` to remove an NP, and then it calls `vp` to remove a VP from that. Since the grammar says that an S is an NP followed by a VP, this will do the right thing. The other rules are exactly analogous.

Having put this program in a file called `grammar.P`, we can load and execute it on our example sentences as follows:

```
% xsb
XSB Version 1.4.1 (94/11/21)
[sequential, single word, optimal mode]
| ?- [grammar].
[Compiling ./grammar]
[grammar compiled, cpu time used: 1.14 seconds]
[grammar loaded]

yes
| ?- s([a,man,loves,the,woman],[]).

yes
| ?- s([every,woman,walks],[]).

yes
| ?- s([a,woman,likes,the,park],[]).

yes
| ?- s([a,woman,likes,the,prak],[]).

no
| ?-
```

When the string is in the language of the grammar, the program will execute successfully through to the end, and the system produces 'yes'. If the string is not in the language, as in the final example where 'park' was misspelled, the system answers 'no'. We called the `s` procedure with the input string as first argument and we gave it the empty list as second argument, because we want it to match the entire input string, wiht nothing left over after seeing an `s`.

The grammar above is called a *Definite Clause Grammar* (DCG) and Prolog supports a special rule syntax for writing DCGs. The syntax is simpler, much closer to the syntax one uses in writing context-free grammar rules. When using the DCG syntax, the programmer doesn't have to write all the string variables threaded through the nonterminal procedure calls; the compiler will do it. Here following is the same Prolog program as above, but witten as a DCG:

```
s --> np, vp.
np --> det, n.
vp --> tv, np.
vp --> v.
det --> [the].
det --> [a].
det --> [every].
n --> [man].
n --> [woman].
n --> [park].
tv --> [loves].
tv --> [likes].
v --> [walks].
```

Notice that these "procedure definitions" use the symbol `-->` instead of `:-` to separate the procedure head from the procedure body. The Prolog compiler converts such rules to (almost) exactly the program above, by adding the extra arguments to the predicate symbols and treating the lists as terminals. The "almost" is because it really translates, for example, a single word list `[loves]` above to the procedure call `'C'(S0,loves,S)`, and includes the definition of this new predicate as:

```
'C'([Word|String],Word,String).
```

This gives exactly the same effect as the Prolog program for the grammar given above.

Consider another example grammar, this one for simple arithmetic expressions over integers with operators `+` and `*`:

```
expr --> term, addterm.
addterm --> [].
addterm --> [+], expr.
term --> factor, multfactor.
multfactor --> [].
multfactor --> [*], term.
factor --> [I], {integer(I)}.
factor --> ['('], expr, [')'].
```

There are several things to note about this DCG. Notice that the list entries, representing terminals, need not appear alone on right-hand-sides of DCG rules, but may accompany nonterminals. Also notice the first rule for `factor`; it has a variable (`I`) in a list, which will cause it to be matched with, and thus set to, the next input symbol. The following procedure call is enclosed in braces. This means that it matches no input symbols and so its translation to Prolog does NOT result in the string variables being added. It remains just a call to the Prolog procedure with one argument: `integer(I)`. The `integer` procedure is a Prolog builtin which tests whether its argument is an integer. Note also that we must quote the parentheses in the final rule. Otherwise, Prolog's reader would not be able to parse them correctly as atoms.

Consider some example executions of this grammar:

```
% xsb
XSB Version 1.4.1 (94/11/21)
[sequential, single word, optimal mode]
| ?- [grammar].
[Compiling ./grammar]
[grammar compiled, cpu time used: 1.309 seconds]
[grammar loaded]

yes
| ?- expr([4,*,5,+,1],[]).

yes
| ?- expr([1,+,3,*,'(',2,+,4,')'],[]).

yes
| ?- expr([4,5,*],[]).

no
| ?-
```

This grammar is not the most obvious one to write down for this expression language. It is specially constructed to avoid being left recursive. We mentioned above that we were writing a recursive descent parser for the grammar, and that is what one gets for a DCG from Prolog's execution strategy. Prolog execution of the underlying deterministic machines and its use of a stack to schedule them naturally yields a recursive descent parser. And it is well known that a recursive descent parser cannot handle left-recursive grammars; it will go into an infinite loop on them. So in Prolog we must avoid left-recursive grammars.

Also a recursive descent parser can be quite inefficient on some grammars, because it may re-parse the same substring many times. In fact, there are grammars for which recursive descent parsers take time exponential in the length of the input string. When using DCGs in Prolog, the programmer must be aware of these limitations and program around them. It is for this reason that some people respond to the claim that "You get a parser for free with Prolog" with "Maybe, but it's not a parser I want to use."

(Another example for adding arguments and using word/3 instead of strings?).

## 2.3 Prolog as a Database Query Langauge

Prolog is an elegant language for database queries. In fact if one constrains Prolog programs to use only atoms, integers and reals (no lists or complex terms) and disallows recursive definitions, one gets a database language that is equivalent to a powerful subset of SQL. In this section we will see how this is so.

A relation in relational database theory is a set of tuples. A common example of a database relation is the `employee` relation, which contains a tuple for each employee in a company. Each tuple might contain fields for: employee number, last name, first name, street address, city, state, zipcode, department number, hire date, and salary. It could, of course, contain many more. We can represent a set of tuples in Prolog as a highly nondeterministic procedure, the procedure returning every one of the tuples in the relation.

```
employee(193,'Jones','John','173 Elm St.','Hoboken','NJ',
                                        12345,1,'25 Jun 93',25500).
employee(181,'Doe','Betty','11 Spring St.','Paterson','NJ',
                                        12354,3,'12 May 91',28500).
employee(198,'Smith','Al','2 Ace Ave.','Paterson','NJ',
                                        12354,3,'12 Sep 93',27000).

(and more...)
```

And we might have a department relation which contains for each department, a tuple that gives its number, name, and employee number of its manager.

```
department(1,'Grocery',181).
department(3,'Deli',193).
department(5,'Produce',199).
...
```

Given these basic relations (also called extensional relations), we can define other relations using Prolog procedure definitions to give us answers to questions we might have about the data. For example, we can define a new relation containing the names of all employees making more than $28,000:

```
well_paid_emp(First,Last) :-
    employee(_Num,Last,First,_Addr,_City,_St,_Zip,_Dept,_Date,Sal),
    Sal > 28000.
```

As another example, we could ask for the name of the manager of the Deli department:

```
deli_manager(First,Last) :-
    department(_Deptno,'Deli',MgrID),
    employee(MgrID,Last,First,_Addr,_City,_St,_Zip,_Dept,_Date,_Sal).
```

Here we first call the department relation to find the employee number of the manager of the Deli department; then we call the employee relation to find the first and last names of the employee with that number.

(Should we introduce negation here? Without recursion, it is pretty easy. Would have to talk about safety.)

## 2.4 Deductive Databases

By staying with the simple data types, but adding recursion to this database language, one gets a language called (positive?) Datalog, which is the language underlying deductive databases. Deductive databases are an extension of relational databases which support more complex data modeling. In this section we will see how simple examples of deductive databases can be represented in Prolog, and we will see more of the limitations of Prolog.

A standard example in Prolog is a geneology database. An extensional relation stores the `parent` relation: `parent(X,Y)` succeeds with `X` and `Y` if `X` has parent `Y`. (Maybe do an example consisting of some English monarchs?) Given this `parent` relation, we can define the ancestor relation as follows:

```
    ancestor(X,Y) :- parent(X,Y).
    ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

This says that X has ancestor Y if X has parent Y; and X has ancestor Y if there is a Z such that X has parent Z and Z has ancestor Y. Given a definition for the parent relation as follows:

```
parent(elizabeth_II, charles_??).
etc.
```

we can query about ancestors, such as:

```
:- ancestor(elizabeth_II,X).
```

and find all of Queen Elizabeth's ancestors.

This works very nicely in Prolog, since the parent graph is (essentially) a tree. However, if we try the same definition of transitive closure for a graph that is not acyclic like a tree, we can be in trouble. Say we have a relation `owes(X,Y)` which indicates that `X` owes money to `Y`, and we want to define a predicate `avoids(X,Y)`, meaning that `X` tries to avoid running into `Y`. The definition is that people avoid someone to whom they owe money, and they avoid anyone that someone to whom they owe money avoids:

```
avoids(X,Y) :- owes(X,Y).
avoids(X,Y) :- owes(X,Z), avoids(Z,Y).
```

This definition has the same form as the ancestor definition. The problem here is that the `owes` relation may be cyclic. It is possible for Andy to owe money to Bill, Bill to own money to Carl and Carl to owe money to Bill:

```
owes(andy,bill).
owes(bill,carl).
owes(carl,bill).
```

and if we ask who Andy avoids:

```
| ?- avoids(andy,X).
```

we get:

```
| ?- avoids(andy,X).

X = bill;

X = carl;

X = bill;

X = carl;

X = bill;

X = carl;

X = bill;
....
```

an infinite loop.

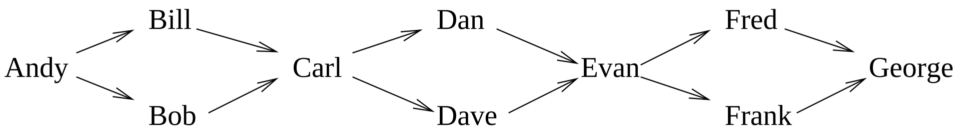


Figure 2.2: Graph on which Prolog is exponential

If we would like to use Prolog as an engine for deductive databases, this shows up a serious problem: that a user can write a simple specification (using only atoms and variablse) and yet Prolog won't give an answer, but will wander off to (or toward) infinity. One couldn't afford to give such a system to a naive database user. There it is important that any query come back with some answer. Think of an SQL system that sometimes went into an infinite loop. It wouldn't be much used, and certainly not by naive users.

This problem of infinite looping is a well-known problem in Prolog and Prolog programmers learn to program around it. The usual fix is to add an extra argument to the `avoids/2` predicate that keeps the list of people encountered in the process of finding the avoidees. Then if one of them is encountered again, the search is made to fail at that point, since it is known that all avoidees from that one have already been found. I.e.,

```
avoids(X,Y,L) :- owes(X,Y), \+ member(Y,L).
avoids(X,Y,L) :- owes(X,Z), \+ member(Z,L), avoids(Z,Y,[Z|L]).
```

Here we've used the Prolog predicate `member/2`, and the Prolog builtin, `\+`, which implements *not.* `\+ member(Y,L)` succeeds just in case `member(Y,L)` fails, and fails if it succeeds. Now with this program and the corresponding query, we get:

```
| ?- avoids(andy,X,[]).

X = bill;

X = carl;

no
| ?-
```

This fix works to avoid the infinite loop, but it sometimes has some undesirable properties. There are graphs for which the computation will be exponential in the number of arcs. Consider the case in which Andy owes money to Bill and Bob, and Bill and Bob owe money to Carl; Carl owes money to Dan and Dave, and Dan and Dave owe money to Evan; Evan owes money to Fred and Frank, and Fred and Frank owe money to George; and so on. The graph of the owes relation can be pictured as in Figure 2.2. On this graph, this query will be exponential. This graph is acyclic, so the original, simpler Prolog program for transitive closure would terminate. But it (and the `member`-enhanced version, too) would recompute the same answers again and again. It would in effect turn this (essentially) linear list into a tree.

## 2.5 Summary

So we have seen that Prolog is an interesting language that combines logic and computation. Some programs are very simple and elegant and execute very reasonably, such as the `append` program or the `member` program. Other programs, such as those derived from context-free grammars, are very simple and elegant, but sometimes have undesirable computational properties. For example, some context-free grammars have recognition times that are exponential in the length of the input string. This is clearly undesirable, since we know that there are recognition algorithms that work for all context-free grammars in at worst cubic time. And even worse, for some grammars, such as left-recursive grammars, Prolog will go into an infinite loop, not returning an answer at all. We also saw the same problem with the Datalog language. Perfectly good specifications might end up with (very) bad computational behavior. In the next chapter we will see how Prolog might be modified to deal with some of these problems.

# Chapter 3

# Tabling and Datalog Programming

In the previous chapter we saw several limitations of Prolog. When we considered grammars in Prolog, we found that the parser provided by Prolog "for free" is a recursive descent parser and not one of the better ones that we'd really like to have. When looking at deductive databases, we found that some perfectly reasonable programs go into an infinite loop, for example transitive closure on a cyclic graph. We had to go to some lengths to program around these limitations, and even then the results were not completely satisfying.

XSB implements a feature not (yet) found in any other Prolog system. It is the notion of tabling, also sometimes called memoization or lemmatization. The idea is very simple: never make the same procedure call twice: the first time a call is made, remember all the answers it returns, and if it's ever made again, use those previously computed answers to satisfy the later request. In XSB the programmer indicates what calls should be tabled by using a compiler directive, such as:

```
:- table np/2.
```

This example requests that all calls to the procedure `np` that has two arguments should be tabled. Predicates that have such declarations in a given program are called tabled predicates.

A simple example of a use of tabling is in the case of a definition of transitive closure in a graph. Assume that we have a set of facts that define a predicate `owes`. The fact `owes(andy,bill)` means that Andy owes Bill some money. Then we use `owes` to define a predicate `avoids` as we did in the previous chapter. A person avoids anyone he or she owes money to, as well as avoiding anyone they avoid.

```
:- table avoids/2.
avoids(Source,Target) :- owes(Source,Target).
avoids(Source,Target) :-
    owes(Source,Intermediate),
    avoids(Intermediate,Target).
```

Here we are assuming that the edges of a directed graph are stored in a predicate `owes/2`. The rules in this program are the same as those used in Prolog to define ancestor. The difference is that in XSB we can make the table declaration, and this declaration guarantees that this predicate will be correctly computed, even if the graph in `owes/2` *is* cyclic. Intuitively it's clear that any call to `avoids` will terminate because there are only finitely many possible calls for any finite graph, and since tabling guarantees that no call is ever evaluated more than once, eventually all the necessary calls will be made and the computation will terminate. The problem with Prolog was that in a cyclic graph the same call was made and evaluated infinitely many times.

Indeed, executing this program on the graph:

```
owes(andy,bill).
owes(bill,carl).
owes(carl,bill).
```

for the query `avoids(andy,X)`, which we saw go into an infinite loop without the table declaration, yields the following under XSB:

```
warren% xsb
XSB Version 1.4.2 (95/4/6)
[sequential, single word, optimal mode]
| ?- [graph].
[Compiling ./graph]
[graph compiled, cpu time used: 0.589 seconds]
[graph loaded]

yes
| ?- avoids(andy,Y).

Y = bill;

Y = carl;

no
| ?-
```

### XSB tabled execution as the execution of concurrent machines

We understood a Prolog evaluation as a set of executing deterministic procedural machines, increasing in number as one of them executes a multiply-defined procedure, and decreasing in number as one of them encounters failure. Then we saw how it was implemented by means of a depth-first backtracking search through the tree of SLD computations, or procedure evaluations. To add the

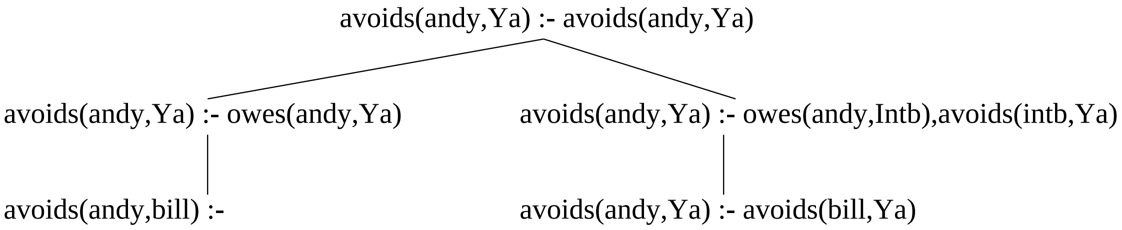


Figure 3.1: Tree for avoid(andy,Ya) goal server

concept of tabling, we have to extend our computational model. Tabling execution is best understood as computation in a concurrent programming language. Nontabled predicates are evaluated exactly as in SLD, with the intuition of a procedure call. Evaluating a tabled predicate is understood as sending the goal to a cacheing goal server and then waiting for it to send back the answers. If no server for the goal exists, then one is created and begins (concurrently) to compute and save answers. If a server for this goal already exists, none needs to be started. When answers become available, they can be (and eventually will be) sent back to all waiting requesters. So tabled predicates are processed "asynchronously", by servers that are created on demand and then stay around forever. On creation, they compute and save their answers (eliminating duplicates), which they then will send to anyone who requests them (including, of course, the initiating requester.)

Now we can see tabled execution as organized around a set of servers. Each server evaluates a nondeterministic procedural program (by a depth-first backtracking search through the alternatives) and interacts with other servers asynchronously by requesting answers and waiting for them to be returned. For each answer returned from a server, computation continues with that alternative.

The abstraction of Prolog computation was the SLD tree, a tree that showed the alternative procedural machines. We can extend that abstraction to tabled Prolog execution by using multiple SLD trees, one for each goal server.

Let's trace the execution of the program for reachability on the simple graph in `owes/2`, given the query `:- avoids(andy,Ya)`. Again, we start with a query and develop the SLD tree for it until we hit a call to a tabled predicate. This is shown in Figure 3.1. Whereas for SLD trees, we used a pseudo predicate `ans` to collect the answers, for SLD trees for goal servers, we will use the entire goal to save the answer, so the left-hand-side of the root of the SLD tree is the same as the right-hand-side. Computation traces this tree in a left-to-right depth-first manner.

So the initial global machine state, or configuration, is:

```
avoids(andy,Ya) :- avoids(andy,Ya).
```

Then rules are found which match the first goal on the right-hand-side of the rule. In this case there are two, which when used to replace the expanded literal, yield two children nodes:

```
avoids(andy,Ya) :- owes(andy,Ya).
```

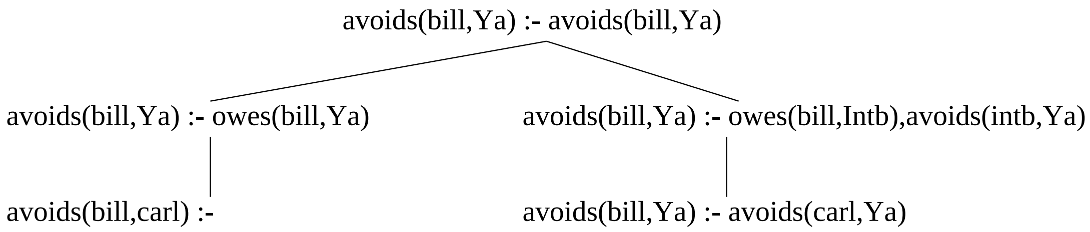


Figure 3.2: Tree for avoid(bill,Ya) goal server

```
avoids(andy,Ya) :- owes(andy,Intb),avoids(Intb,Ya).
```

Computation continues by taking the first one and expanding it. Its selected goal (the first on the right-hand side) matches one rule (in this case a fact) which, after replacing the selected goal with the (empty) rule body, yields:

```
avoids(andy,bill) :-
```

And since the body is empty, this is an answer to the original query, and the system could print out `Y=bill` as an answer.

Then computation continues by using the stack to find that the second child of the root node:

```
avoids(andy,Ya) :- owes(andy,Intb),avoids(Intb,Ya).
```

should be expanded next. The selected goal matches with the fact `owes(andy,bill)` and expanding with this results in:

```
avoids(andy,Ya) :- avoids(bill,Ya).
```

Now the selected goal for this node is `avoids(bill,Ya)`, and `avoids` is a tabled predicate. Therefore this goal is to be solved by communicating with its server. Since the server for that goal does not exist, the system creates it, and schedules it to compute its answers. This computation is shown in Figure 3.2.

This computation sequence for the goal `avoid(bill,Y)` is very similar to the previous one for `avoid(andy,Y)`. The first clause for `avoids` is matched, followed by the one fact for `owes` that has `bill` as its first field, which generates the left-most leaf of the tree of the figure. This is an answer which this (concurrently executing) server could immediately send back to the requesting node in Figure 3.1. Alternatively, computation could continue in this server to finish the tree pictured in Figure 3.2, finally generating the right-most leaf:

```
avoids(bill,Ya) :- avoids(carl,Ya)
```

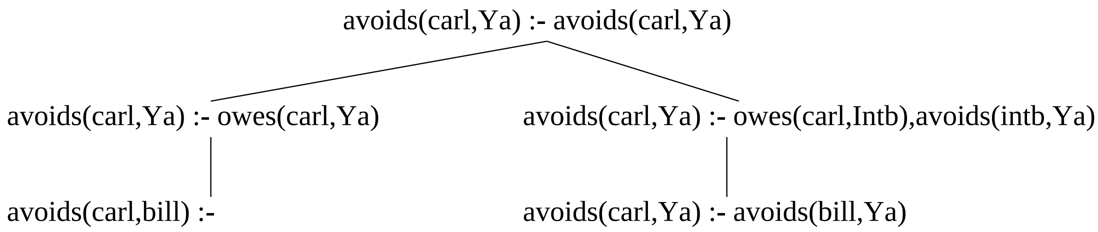


Figure 3.3: Tree for avoids(carl,Ya) goal server

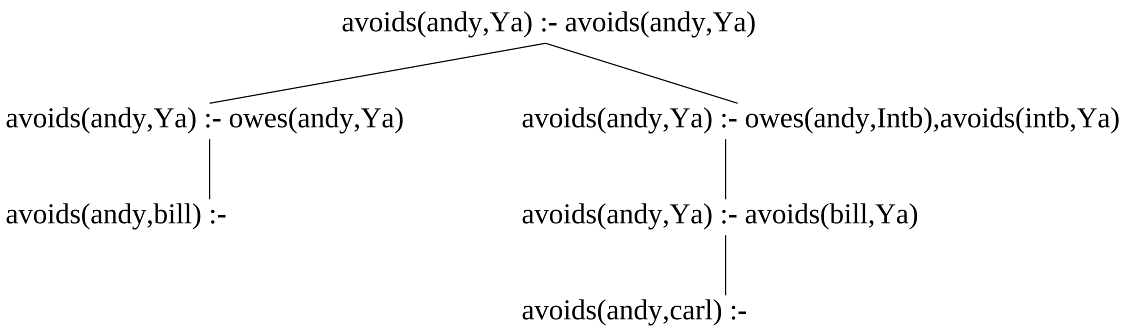


Figure 3.4: Updated tree for avoid(andy,Ya) goal server

Now since the selected goal here is tabled, the server for it is queried and its response is awaited. Again, there is no server for this goal yet, so the system creates one and has it compute its answers. This computation is shown in Figure 3.3.

This computaton is beginning to look familiar; again the form of the computation tree is the same (because only one clause matches `owes(carl,Y)`). Again an answer, `avoids(carl,bill)`, is produced (and is scheduled to be returned to its requester) and computation continues to the right-most leaf of the tree with the selected goal of `avoids(bill,Ya)`. This is a tabled goal and so will be processed by its server. But now the server *does* exist; it is the one Figure 3.2. Now we can continue and see what happens when answers are returned from servers to requesters. Note that exactly *when* these answers are returned is determined by the scheduling strategy of our underlying concurrent language. We have thus far assumed that the scheduler schedules work for new servers before scheduling the returning of answers. Other alternatives are certainly possible.

Now in our computation there are answers computed by servers that need to be sent back to their requesters. The server for `avoids(bill,Ya)` (in Figure 3.2) has computed an answer `avoids(bill,carl)`, which it sends back to the server for `avoids(andy,Ya)` (in Figure 3.1). That adds a child to the rightmost leaf of the server's tree, producing the new tree shown in Figure 3.4. Here the answer (`avoids(bill,carl)`) has been matched with the selected goal (`avoids(bill,Ya)`) giving a value to `Ya`, and generating the child `avoids(andy,carl) :-`. Note that this child is a new answer for this server.

Computation continues with answers being returned from servers to requesters until all answers have been sent back. Then there is nothing left to do, and computation terminates. The trees of the three servers in the final state are shown in Figure 3.5. Duplicate answers may be generated (as we see in each server) but each answer is sent only once to each requester. So duplicate answers are eliminated by the servers.

Let's be more precise and look at the operations that are used to construct these server trees. We saw that the SLD trees of Prolog execution could be described by giving a single rule, PROGRAM CLAUSE RESOLUTION, and applying it over and over again to an initial node derived from the query. A similar thing can be done to generate sets of server trees that represent the computation of tabled evaluation. For this we need three rules:

**Definition 3.0.1 (Program Clause Resolution)** Given a tree with a node labeled $A:-A_1, A_2, \ldots, A_n$, which is either a root node of a server tree or $A_1$ is not indicated as tabled. Also given a rule in the program of the form $H:-B_1, B_2, \ldots, B_k$, (with all new variables) and given that $H$ and $B_1$ match with matching variable assignment $\theta$, then add a new node as a child of this one and label it with $(A:-B_1, B_2, \ldots, B_k, A_2, \ldots, A_n)\theta$, if it does not already have a child so labeled. Note that the matching variable assignment is applied to all the goals in the new label. □

**Definition 3.0.2 (Subgoal Call)** Given a nonroot node with label $A:-A_1, A_2, \ldots, A_n$, where $A_1$ is indicated as tabled, and there is no tree with root $A_1:-A_1$, create a new tree with root $A_1:-A_1$. □

**Definition 3.0.3 (Answer Clause Resolution)** Given a non-root node with label $A:-A_1, A_2, \ldots, A_n$, and an answer of the form $B:-$ in the tree for $A_1$, then add a new node as child of this node labeled by $(A:-A_2, \ldots, A_n)\theta$, where $\theta$ is the variable assignments obtained from matching $B$ and $A_1$ (if there is not already a child with that label.) □

So for example the trees in Figure 3.5 are constructed by applying these rules to the initial tree (root) for the starting goal. XSB can be understood as efficiently constructing this forest of trees. We have seen that XSB with tabling will terminate on a query and program for which Prolog will loop infinitely. It turns out that this is not just an accident, but happens for many, many programs. For example, here we've written the transitive closure of owes using a right recursive rule, i.e., the recursive call to `avoids` follows the call to `owes` in the second rule defining `avoids`. We could also define `avoids` with a rule that has a call to `avoids` before a call to `owes`. That definition would not terminate in Prolog for any graph, but with tabling, it is easily evaluated correctly.

## 3.1 More on Transitive Closure

We saw in the previous section how XSB with tabling will correctly and finitely execute a transitive closure definition even in the presence of cyclic data. Actually, this is only a simple example of the power of tabled evaluation.

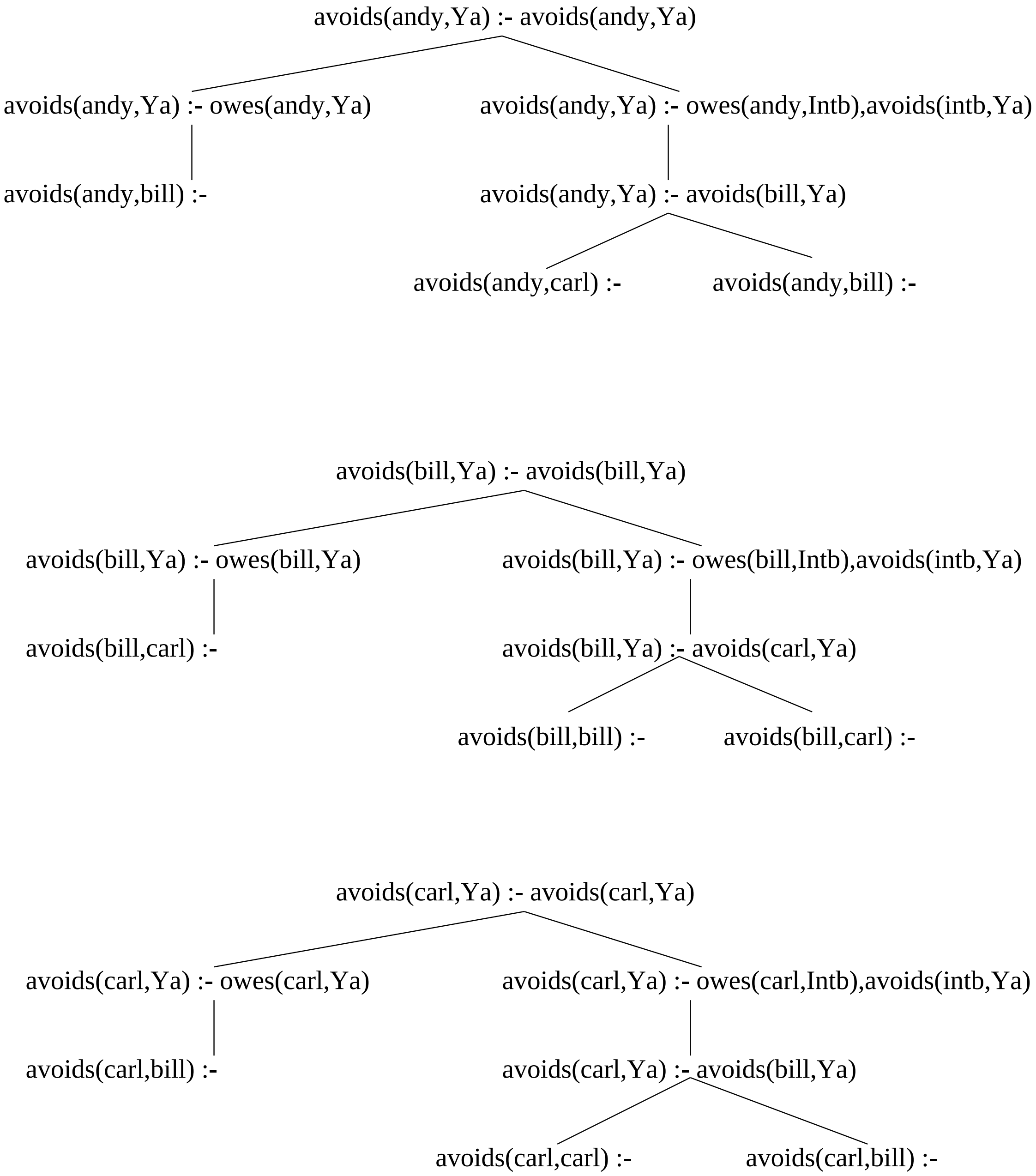


Figure 3.5: Final state for all goal servers for query avoids(andy,Ya)

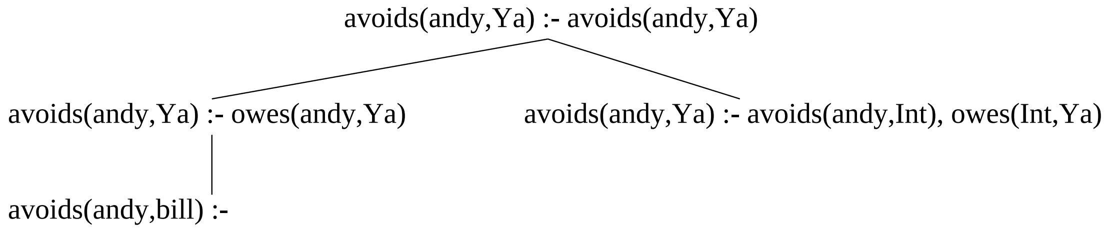


Figure 3.6: Beginning of evaluation of avoids(andy,Ya) for left-recursive transitive closure definition

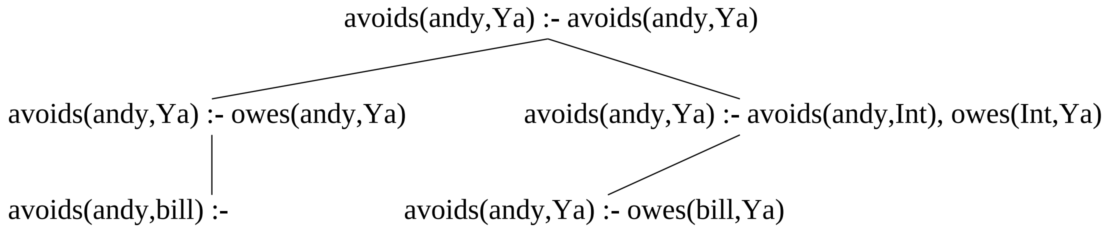


Figure 3.7: More of the evaluation of avoids(andy,Ya) for left-recursive transitive closure definition

We can write another version of transitive closure:

```
:- table avoids/2.
avoids(Source,Target) :- owes(Source,Target).
avoids(Source,Target) :-
    avoids(Source,Intermediate),
    owes(Intermediate,Target).
```

This one is left recursive. A Prolog programmer would not consider writing such a definition, since in Prolog it is guaranteed to be nonfinite. But with tabling, this definition works fine. As a matter of fact, it is generally a more efficient way to express transitive closure than is right recursion. In this section we will look at various versions of transitive closure and compare their efficiency.

Let's consider the evaluation of the same `avoids` query on the same `owes` data as above, but using the left-recursive definition of avoids.

Figure 3.6 shows the state of the initial server when it first encounters a request to a server. Note that this time the request to a server is to the server for `avoids(andy,Ya)`, and this is the server itself. (The names of the variables don't matter when finding a server; they are just "placeholders", so any server with the same arguments with the same pattern of variables works.) The server does have an answer already computed, so it can send it back to the requester (itself), and that results in the tree of Figure 3.7. Now the new leaf, created by the returned answer, can be expanded (by Program Clause Resolution) yielding a new answer, `avoids(andy,carl)`. This answer can be returned to the (only) requester for this server, and that generates a second child for the requester node; this state is shown in Figure 3.8.

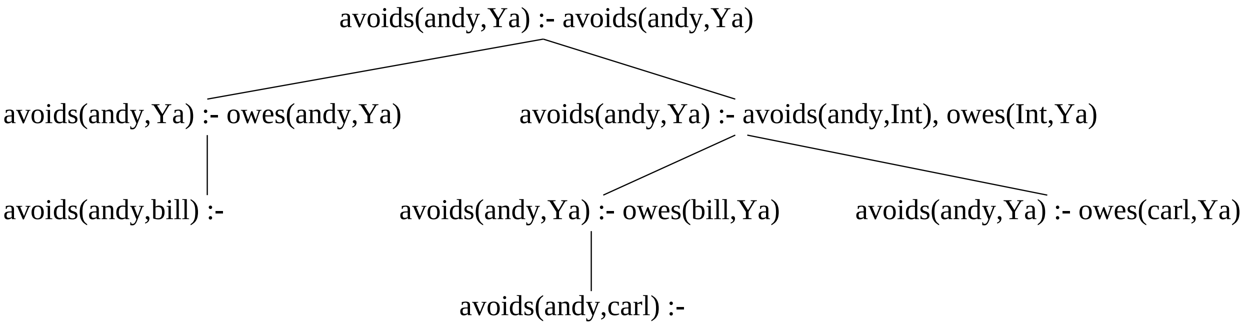


Figure 3.8: More of the evaluation of avoids(andy,Ya) for left-recursive transitive closure definition

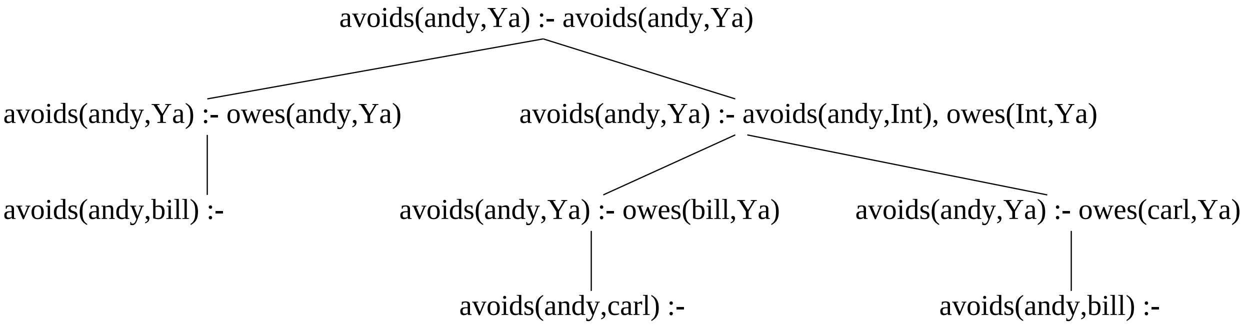


Figure 3.9: Final forest for avoids(andy,Ya) for left-recursive transitive closure definition

Now this node can be expanded (by Program Clause Resolution) to obtain the tree of Figure 3.9 Here we have generated another answer, but it is the same as one we've already generated, so returning it to the requester node will *not* generate any new children. All operations have been applied and no more are applicable, so we have reached the final forest of trees, a forest consisting of only one tree. Note that we have the correct two (distinct) answers: that andy avoids bill and andy avoids carl.

The right-recursive definition and the left-recursive definition of avoids both give us the correct answers, but the left-recursive definition (for this query) generates only one tree, whereas the right recursive definition generates several. It seems as though the left-recrsive definition would compute such queries more efficiently, and this is indeed the case.

Consider transitive closure over an `owes` relation that defines a cycle. E.g.,

```
owes(1,2).
owes(2,3).
owes(3,4).
...
owes(99,100).
owes(100,1).
```

defines a graph with a cycle of length 100. How would the trees in the forest look after evaluation of a query to `avoids(1,X)` using the right-recursive transitive closure definition? For each $n$ between 1 and 100, there is a tree with root: `avoids(`$n$`,Y)`. And each such tree will have 100 leaf answer nodes. So the forest will have at least $100^2$ nodes, and for a cycle of length $n$ the forest would be of size $O(n^2)$.

What does the forest look like if we use the left-recursive definition? It has one tree with root, `avoids(1,Y)`, and that tree has 100 answer leaves. Generalizing from the tree of Figure 3.9, we see that it is a very flat tree, and so for a cycle of length $n$, the tree (forest) would be of size $O(n)$. The left-recursive definition is indeed the more efficient to compute with. Indeed the complexity of a single-source query to the left-recursive version of transitive closure is linear in the number of edges in the graph reachable from the source node.

## 3.2 Other Datalog Examples

In the previous section we saw how tabling can finitely process certain programs and queries for which Prolog would go into an infinite loop. Tabling can also drastically improve the efficiency of some terminating Prolog programs. Consider reachability in a DAG. Prolog will terminate, but it may retraverse the same subgraph over and over again.

Let's reconsider the mostly linear `owes` graph at the end of the previous chapter (shown in Figure 2.2) on which Prolog had exponential complexity. Consider evaluating the query `avoids(andy,X)` with the left-recursive tabled definition of transitive closure. The forest for this evaluation will again consist of a single tree, and that tree will be very flat, similar in form to the one of Figure 3.9. Thus tabled evaluation will take linear time. So this is an example in which Prolog (with its right recursive definition) will terminate, but take exponential time; XSB with the left-recursive definition and tabling will terminate in linear time.

The "doubly-connected linear" graph used here may seem unusual and specially chosen, but the characteristics of the graph that cause Prolog to be exponential are not that unusual. Many naturally occurring directed graphs have multiple paths to the same node, and this is what casues the problem for Prolog. [For example, consider a graph (generated by graph-base [**?**]) that places 5-letter English words in a graph with an edge between two words if one can be obtained from the other by changing a single letter.... (get example from Juliana, and see how it works.)

Transitive closure is perhaps the most common example of a recursive query in Datalog, but other query forms can be encountered. Consider the definition of `same_generation`. Given binary relations `up` and `down` on nodes, define a binary relation on nodes that associates two nodes if one can be reached from the other by going $n$ steps up and then $n$ steps down, for some $n$. The program is:

```
same_generation(X,X).
same_generation(X,Y) :-
```

```
    up(X,Z1),
    same_generation(Z1,Z2),
    down(Z2,Y).
```

The name of the predicate arises from the fact that if we let `up` be defined by a "parent_of" relation and `down` be defined by the "child_of" relation, then `same_generation/2` defines people in the same generation.

[to be continued...]

## 3.3 Some Simple Graph Problems

Consider the problem of finding connected components in a directed graph. Assume we have a node and we want to find all the nodes that are in the same connected component as the given node.

The first thought that comes to mind is: given a node X, find those nodes Y that are reachable from X and from which you can get back to X. So we will assume that edges are given by an `edge/2` relation:

```
sameSCC(X,Y) :- reach(X,Z), reach(Z,Y).

:- table reach/2.
reach(X,X).
reach(X,Y) :- reach(X,Z), edge(Z,Y).
```

Indeed given a node X, this will find all nodes in the same strongly connected component as X, however it will in general take $O(n * e)$ time, where $n$ is the number of nodes and $e$ is the number of edges. The reason is that given an X, there are $n$ possible Z values and for each of them, we will find everything reachable from them, and each search can take $O(e)$ time.

However, we can do better. It is known that this problem can be solved in $O(e)$ time. The idea is, given a node X, to find all nodes reachable from X following edges forward. Then find all nodes reachable from X following edges backward (i.e., follow edges against the arrow.) Then intersect the two sets. That will be the set of nodes in X's SCC, because if Y is in both these sets, you can follow the edges forward from X to Y and then since there is also a backwards path from X to Y, there is forward path from Y to X, so you can get from X to Y and back to X following edges forward. So the program that does this is:

```
% sameSCC(+X,-Y)
sameSCC(X,Y) :- reachfor(X,Y), reachback(X,Y).
```

```
:- table reachfor/2, reachback/2.
reachfor(X,X).
reachfor(X,Y) :- reachfor(X,Z),edge(Z,Y).

reachback(X,X).
reachback(X,Y) :- reachback(X,Z),edge(Y,Z).
```

Let's now consider its complexity to see why it is $O(e)$. For a fixed value X, the computation of the query `reachfor(X,Y)` takes $O(e)$ time. Then we may have $O(n)$ calls to `reachback(X,Y)` (one for each Y) but they all use one underlying call to `reachback(X,_)` which takes $O(e)$ and is done only once. So when we add all that up (assuming a connected graph), we get $O(e)$ time.

## 3.4 Genome Examples

[We need to get the semantics of the genome queries from Tony. Does anybody remember?]

## 3.5 Inferring When to Table

Up to now whenever we wanted calls to a predicate to be tabled, we explicitly coded a table directive to indicate the specific predicate to table. There is a facility in XSB for the programmer to direct the system to choose what predicates to table, in which case the system will generate table directives automatically. There are two directives that control this process: `auto_table/0` and `suppl_table/1`. When such a directive is placed in a source file, it applies to all predicates in that file when it is compiled.

`auto_table/0` causes the compiler to table enough predicates to avoid infinite loops due to redundant procedure calls. The current implementation of `auto_table` uses the call graph of the program. There is a node in the call graph of a program for each predicate, `P/N`, that appears in the program. There is an edge from node for predicate `P/N` to the node for predicate `Q/M` if there is a rule in the program with an atom with predicate `P/N` in the head and a literal with predicate `Q/M` in the body. The algorithm constructs the call graph and then chooses enough predicates to table to ensure that all loops in the call graph are broken. The algorithm, as currently implemented in XSB, finds a minimal set of nodes that breaks all cycles. The algorithm can be exponential in the number of predicates in the worst case[1]. If the program is a Datalog program, i.e., it has no recursive data structures, then `auto_table` *is* guaranteed to make all queries to it terminate finitely. Termination of general programs is, of course, undecidable, and `auto_table` may or may not improve their termination characteristics.

[1] The algorithm to find such a minimal set of predicates corresponds to the *feedback vertex set* problem and is NP-Complete [6].

The goal of `auto_table` is to guarantee termination of Datalog programs, but there are other uses tabling. Tabling can have a great effect on the efficiency of terminating programs. Example 3.5.1 illustrates how a multiway join predicate can use tabling to eliminate redundant subcomputations.

**Example 3.5.1** *Consider the following set of relations describing a student database for a college:*

1. *student(StdId,StdName,Yr): Student with ID* `StdId` *and name* `StdName` *is in year* `Yr`*, where year is 1 for freshman, 2 for sophomores, etc.*
2. *enroll(StdId,CrsId): Student with ID* `StdId` *is enrolled in the course with number* `CrsId`*.*
3. *course(CrsId,CrsName): Course with number* `CrsId` *has name* `CrsName`*.*

*We define a predicate* `yrCourse/2`*, which, given a year, finds all the courses taken by some student who is in that year:*

```
yrCourse(Yr,CrsName) :-
    student(StdId,_,Yr), enroll(StdId,CrsId), course(CrsId,CrsName).
```

*Note that it will most likely be the case that if one student of a given year takes a course then many students in the same year will take that same course. Evaluated directly, this definition will result in the course name being looked up for every student that takes a given course, not just for every course taken by some student. By introducing an intermediate predicate, and tabling it, we can elminate this redundancy:*

```
yrCourse(Yr,CrsName) :-
    yrCrsId(Yr,CrsId), course(CrsId,CrsName).

:- table yrCrsId/2.
yrCrsId(Yr,CrsId) :-
    student(StdId,_,Yr), enroll(StdId,CrsId).
```

*The intermediate predicate* `yrCrsId` *is tabled and so will eliminate duplicates. Thus* `course` *will only be accessed once for each course, instead of once for each student. This can make a very large difference in evaluation time.*

In this example a table has been used to eliminate duplicates that arise from the database operations of a join and a projection. Tables may also be used to eliminate duplicates arising from unions.

The `suppl_table/1` directive is a means by which the programmer can ask the XSB system to perform such factoring automatically. The program:

```
:- edb student/3, enroll/2, course/2.
:- suppl_table(2).
yrCourse(Yr,CrsName) :-
    student(StdId,_,Yr), enroll(StdId,CrsId), course(CrsId,CrsName).
```

will automatically generate a program equivalent to the one above with the new intermediate predicate and the table declaration.

To understand precisely how `suppl_table` works, we need to understand some distinctions and definitions of deductive databases. Predicates that are defined by sets of ground facts can be designated as *extensional predicates*. The extensional predicates make up the extensional database (EDB). The remaining predicates are called *intensional predicates*, which make up the intensional database (IDB), and they usually have definitions that depend on the extensional predicates. In XSB the declaration:

```
:- edb student/3, enroll/2, course/2.
```

declares three predicates to be extensional predicates. (Their definitions will have to be given elsewhere.) We define the data dependency count of an IDB clause to be the number of tabled IDB predicate it depends on plus the number of EDB predicates it depends on (*not* through a tabled IDB predicate.) The command:

```
:- suppl_table(2).
```

instructs XSB to factor any clause which has a data dependency count of greater than two. In Example 3.5.1 the data dependency count of the original form of `join/2` is three, while after undergoing supplementary tabling, its count is two. Choosing a higher number for `suppl_table` results in less factoring and fewer implied table declarations.

The next subsection describes somewhat more formally how these transformations affect the worst-case complexity of query evaluation.

### On the Complexity of Tabled Datalog Programs

The worst-case complexity of a Datalog program (with every predicate tabled) is:

$$\sum_{clause} (len(clause) + k^{num\_of\_vars(body(clause))})$$

where $k$ is the number of constants in the Herbrand base (i.e., in the program). One can see how this can be achieved by making all base relations to be cross products of the set of constants in the program. Assume the call is completely open. Then if there are $v_1$ variables in the first subgoal, there will be $k^{v_1}$ tuples. Each of theses tuples will be extended through the second subgoal, and consider how many tuples from the second subgoal there can be: $k^{v_2}$ where $v_2$ is the number of variables appearing in the second subgoal and not appearing in the first. So to get through the second subgoal will take time $k^{v_1} * k^{v_2}$. And similarly through the entire body of the clause. Each subgoal multiplies by a factor $k^v$ where $v$ is the number of new variables. And every variable in the body of the clause is new once and only once. This is the reason for the second component in the summation above. The first component is just in case there are no variables in the clause. For an entire program one can see that the complexity (for a nonpropositional) datalog program is $O(k^v)$ where $v$ is the maximum number of variables in the body of any clause.

We can use folding to try to improve the worst-case efficiency of a Datalog program. Consider the query:

```
(7) :- p(A,B,C,D),q(B,F,G,A),r(A,C,F,D),s(D,G,A,E),t(A,D,F,G).
```

It has 7 variables (as indicated by the number in parentheses that precedes the query), so its worst-case efficiency is $O(n^7)$. However, we can fold the first two subgoals by introducing a new predicate, obtaining the following program:

```
(6) :- f1(A,C,D,F,G),r(A,C,F,D),s(D,G,A,E),t(A,D,F,G).
(6) f1(A,C,D,F,G) :- p(A,B,C,D),q(B,F,G,A).
```

This one has a maximum of 6 variables in the query or in the right-hand-side of any rule, and so has a worst-case complexity of $O(n^6)$.

We can do a couple of more folding operations as follows:

```
(5) :- f2(A,D,F,G),s(D,G,A,E),t(A,D,F,G).
(5) f2(A,D,F,G) :- f1(A,C,D,F,G),r(A,C,F,D).
(6) f1(A,C,D,F,G) :- p(A,B,C,D),q(B,F,G,A).

(4) :- f2(A,D,F,G),f3(D,G,A),t(A,D,F,G).
(4) f3(D,G,A) :- s(D,G,A,E).
(5) f2(A,D,F,G) :- f1(A,C,D,F,G),r(A,C,F,D).
(6) f1(A,C,D,F,G) :- p(A,B,C,D),q(B,F,G,A).
```

Thus far, we have maintained the order of the subgoals. If we allow re-ordering, we could do the following. For each variable, find all the variables that appear in some subgoal that it appears in. Choose the variable so associated with the fewest number of other variables. Factor those subgoals, which removes that variable (at least). Continue until all variables have the same number of associated variables.

Let's apply this algorithm to the initial query above. First we give each variable and the variables that appear in subgoals it appears in.

```
A:BCDEFG
B:ACDFG
C:ABDF
D:BCDEFG
E:ADG
F:ABGCD
G:ABFDE
```

Now E is the variable associated with the fewest number of other variables, so we fold all the literals (here only one) containing E, and obtain the program:

```
(6) :- p(A,B,C,D),q(B,F,G,A),r(A,C,F,D),f1(D,G,A),t(A,D,F,G).
(4) f1(D,G,A) :- s(D,G,A,E).
```

Now computing the new associated variables for the first clause, and then choosing to eliminate C, we get:

```
A:BCDFG
B:ACDFG
C:ABDF
D:ABCFG
F:ABGCD
G:ABFD

(5) :- f2(A,B,D,F),q(B,F,G,A),f1(D,G,A),t(A,D,F,G).
(4) f1(D,G,A) :- s(D,G,A,E).
(5) f2(A,B,D,F) :- p(A,B,C,D),r(A,C,F,D).
```

Now computing the associated variables for the query, we get:

```
a:bdfg
b:adfg
d:abfg
f:abdg
g:abfd
```

All variables are associated with all other variables, so no factoring can help the worst-case complexity, and the complexity is $O(k^5)$.

However, there is still some factoring that will eliminate variables, and so might improve some queries, even though it doesn't guarantee to reduce the worst-case complexity.

```
(4) :- f3(A,D,F,G),f1(D,G,A),t(A,D,F,G).
(5) f3(A,D,F,G) :- f2(A,B,D,F),q(B,F,G,A).
(4) f1(D,G,A) :- s(D,G,A,E).
(5) f2(A,B,D,F) :- p(A,B,C,D),r(A,C,F,D),

(3) :- f4(A,D,G),f1(D,G,A).
(4) f4(A,D,G) :- f3(A,D,F,G),t(A,D,F,G).
(5) f3(A,D,F,G) :- f2(A,B,D,F),q(B,F,G,A).
(4) f1(D,G,A) :- s(D,G,A,E).
(5) f2(A,B,D,F) :- p(A,B,C,D),r(A,C,F,D).
```

The general problem of finding an optimal factoring is conjectured to be NP hard. (Steve Skiena has the sketch of a proof.)

## 3.6 Datalog Optimization in XSB

[Do we want to do it at all, and if so, here?] I think we do want it, but I don't know about here.

# Chapter 4

# Grammars

In this chapter we will explore how tabling can be used when writing DCG grammars in XSB. Tabling eliminates redundancy and handles grammars that would infinitely loop in Prolog. This makes the "parser you get for free" in XSB one that you might well want to use.

## 4.1 An Expression Grammar

Consider the following expression grammar, expressed as a DCG. This is the "natural" grammar one would like to write for this langauge.

```
% file grammar.P
:- table expr/2, term/2.

expr --> expr, [+], term.
expr --> term.
term --> term, [*], primary.
term --> primary.
primary --> ['('], expr, [')'].
primary --> [Int], {integer(Int)}.
```

This grammar is left-recursive and would cause a problem for Prolog, but with the table declarations, XSB handles it correctly. Notice that we must table `expr/2` because the XSB parser adds 2 arguments to the nonterminal symbol `expr`. (An alternative would be to use the `auto_table` directive.) After compiling and loading this program, we can execute it:

```
| ?- [grammar].
[Compiling ./grammar]
```

```
[grammar compiled, cpu time used: 0.419 seconds]
[grammar loaded]

yes
| ?- expr([1,+,2,*,3,*,'(',4,+,5,')'],[]).
Removing open tables.

yes
| ?-
```

Here the system answers "yes" indicating that the string is recognized. (The message "Removing open tables" is given when a query with no variables evaluates to true. It indicates that once the first successful execution path is found, computation terminates, and any tables that are not fully evaluated have been deleted.) This grammar is not only more elegant than the one we wrote in Prolog for the same langauge, it is more "correct". What I mean by that is that this grammar associates "+" and "*" to the left, as is usual, rather than to the right as the did the Prolog grammar that we gave in an earlier chapter.

So far we've only seen DCG's that represent simple context-free grammars. The DCG representation is actually much more powerful and can be used to represent very complicated systems. As a first example, let's say we want to implement an evaluator of these integer expressions. We can add semantic arguments to the nonterminals above to contain the value of the subexpression recognized. The following grammar does this:

```
% file grammar.P
:- table expr/3, term/3.

expr(Val) --> expr(Eval), [+], term(Tval), {Val is Eval+Tval}.
expr(Val) --> term(Val).
term(Val) --> term(Tval), [*], primary(Fval), {Val is Tval*Fval}.
term(Val) --> primary(Val).
primary(Val) --> ['('], expr(Val), [')'].
primary(Int) --> [Int], {integer(Int)}.
```

Recall that the braces are used to indicate that the enclosed calls are calls to Prolog predicates, not to nonterminals which recognize parts of the input string.

We can compile and run this program to evaluate the expression provided and have it return its integer value as follows:

```
| ?- [grammar].
[Compiling ./grammar]
[grammar compiled, cpu time used: 0.75 seconds]
[grammar loaded]
```

```
yes
| ?- expr(Val,[1,+,2,*,3,*,'(',4,+,5,')'],[]).

Val = 55;

no
| ?-
```

Notice that the string arguments are added *after* any explicit arguments. So we wrote what looked like a procedure definition for `expr` that had only one argument, but we get a definition that has 3 arguments, and that's the way we called it at the top-level prompt.

As mentioned, this grammar treats the operators as left associative, which is the usual convention for arithmetic expressions. While it doesn't matter semantically for the operations of "+" and "$*$", which are associative operators anyway, were we to extend this evaluator to handle "$-$" or "/" (as is very easy), then this correct associativity would be critical.

## 4.2 Representing the Input String as Facts

In actuality, this way to process grammars with tabling is not as efficient as it might be. The reason is that the arguments to the tabled nonterminals consist of lists, and so each time a call is copied into the table, a long list may have to be copied. Also answers consist of tails of the list corresponding to the input string, and these may be long as well. We can use a slightly different representation for the input string to avoid this inefficiency.

Instead of representing the input string as a list, we will store it in the database, representing it by a set of facts. We will think of each word in the input sentence as being numbered, starting from 1. Then we will store the string as a set of facts of the form, `word(`$n, word$`)`, where $word$ is the $n^{th}$ word in the input string. For example, the string used in the example above, `[1,+,2,*,3,*,'(',4,+,5,')']`, would be represented by the following facts:

```
word(1,1).
word(2,+).
word(3,2).
word(4,*).
word(5,3).
word(6,*).
word(7,'(').
word(8,4).
word(9,+).
word(10,5).
word(11,')').
```

Recall that we said that the DCG translation translates lists in the body of DCG rules to calls to the predicate `'C'/3`, which is defined to process the list input. But we can redefine this predicate to look at the `word/2` predicate as follows:

```
'C'(I,W,I1) :- word(I,W), I1 is I+1.
```

(We could alternatively use `word/3` facts and explicitly store the two consecutive integers, so no computation would be involved.) With this definition of `'C'/3`, we can use the same DCG for parsing but now, rather than using lists to represent positions in the input, the system uses integers.

```
% grammar.P
:- table expr/3, term/3.
'C'(I,W,I1) :- word(I,W), I1 is I+1.

eval_string(String,Val) :-
    retractall(word(_,_)),
    assert_words(String,1,N),
    abolish_all_tables,
    expr(Val,1,N).

assert_words([],N,N).
assert_words([Word|Words],N,M) :-
        assert(word(N,Word)), N1 is N+1, assert_words(Words,N1,M).

expr(Val) --> expr(Eval), [+], term(Tval), {Val is Eval+Tval}.
expr(Val) --> term(Val).
term(Val) --> term(Tval), [*], primary(Fval), {Val is Tval*Fval}.
term(Val) --> primary(Val).
primary(Val) --> ['('], expr(Val), [')'].
primary(Int) --> [Int], {integer(Int)}.
```

Here we've defined a predicate `eval_string` to take an input string as a list, assert it into the database as a set of `word/2` facts and then to call `expr` to parse and evaluate it. Notice that we need both to retract any facts previously stored for `word/2` and to abolish any tables that were created during previous evaluations of strings. This is because old tables are no longer valid, since the new input string changes the meanings of the integers that represent positions in the input string.

We can compile and call this predicate as follows:

```
| ?- [grammar].
[Compiling ./grammar]
++Warning: Redefining the standard predicate: C / 3
```

```
[grammar compiled, cpu time used: 1.069 seconds]
[grammar loaded]

yes
| ?- eval_string([1,+,2,*,3,*,'(',4,+,5,')'],V).

V = 55;

no
| ?-
```

The warning is to alert the user to the fact that a standard predicate is being redefined. In this case, that is exactly what we want to do, and so we can safely ignore the warning.

## 4.3 Mixing Tabled and Prolog Evaluation

We can extend this evaluator in the following interesting way. Say we want to add exponentiation. We introduce a new nonterminal, `factor`, for handling exponentiation and make it right recursive, since exponentiation is right associative.

```
:- table expr/3, term/3.

expr(Val) --> expr(Eval), [+], term(Tval), {Val is Eval+Tval}.
expr(Val) --> term(Val).
term(Val) --> term(Tval), [*], factor(Fval), {Val is Tval*Fval}.
term(Val) --> factor(Val).
factor(Val) --> primary(Num), [^], factor(Exp),
    {Val is floor(exp(log(Num)*Exp)+0.5)}.
factor(Val) --> primary(Val).
primary(Val) --> ['('], expr(Val), [')'].
primary(Int) --> [Int], {integer(Int)}.
```

However, we don't table the new nonterminal. Prolog's evaluation strategy handles right recursion in grammars finitely and efficiently. In fact, Prolog has linear complexity for a simple right-recursive grammar, but with tabling it would be quadratic. Thus an advantage of XSB is that it allows tabled and nontabled predicates to be freely intermixed, so that the programmer can choose the strategy that is most efficient for the situation at hand.

## 4.4 So What Kind of Parser is it?

A pure DCG, one without extra arguments (and without look-ahead symbols which we haven't discussed at all), represents a context-free grammar, and the Prolog and XSB engines provide recognizers for it. A context-free recognizer is a program that, given a context-free grammar and an input string, responds "yes" or "no" according to whether the input string is in or is not in the language of the given grammar.

We noted that the recognizer that "you get for free" with Prolog is a recursive descent recognizer. The recognizer "you get for free" with XSB and tabling is a variant of Earley's algorithm, or an active chart recognition algorithm (ref Peter Norvig and B. Sheil.)

The worst-case complexity of the recognizer under XSB (with all recursive nonterminals tabled) is $O(n^{k+1})$ where $n$ is the length of the input string and $k$ is the maximum number of nonterminals and terminals on the right-hand-side of any rule. A little thought shows that this is consistent with the discussion of the complexity of datalog programs under XSB in Chapter ??. This is an example of a situation in which tabling turns an otherwise exponential algorithm (recursive descent) into a polynomial one (active chart recognition.)

Any grammar can be changed to another grammar that represents the same language but has two (or fewer) nonterminal symbols on the right-hand-side of every rule. This is the so-called Chomsky normal form. So if we transform a grammar into this form, then its worst-case complexity will be $O(n^3)$.

In fact, the folding and tabling that is done automatically by the XSB compiler when the `:- suppl_table.` and `:- edb word/2.` directives are given results in exactly the transformation necessary to transform a grammar to Chomsky normal form. So giving those directives guarantee the best worst-case complexity.

For unambiguous grammars, the complexity is actually $O(n^2)$. I find this an intriguing situation, which is particularly pleasant, since it is undecidable whether a context-free grammar is or is not ambiguous. So it will be undecidable to determine what the actual complexity of the algorithm is, given a grammar. But, no problem; the algorithm will automatically tune itself to the data (grammar in this case) to be more efficient in the unambiguous case.

These complexity results are very good and make the parser that "you get for free" with XSB quite a desirable parser.

## 4.5 Building Parse Trees

The desirable complexity results of the previous section hold for *recognition* of context-free languages. But often it is the case that one wants, given a string to construct the parse tree(s) for it. An easy way to do this is to add an argument to each nonterminal to contain the parse tree, and then add the necessary code to each rule to construct the appropriate tree. For example, the following rule from our expression example:

```
expr(Val) --> expr(Eval), [+], term(Tval), {Val is Eval+Tval}.
```

could become:

```
expr(Val,+(E1,T1)) --> expr(Eval,E1), [+], term(Tval,T1), {Val is Eval+Tval}.
```

We've added a second argument and in it constructed the parse tree for the entire expression phrase given the parse trees for the component expression and term phrases. All the other rules would be extended accordingly.

This is very easy to do and in almost all cases is the best way to construct the parse tree. However, from a complexity standpoint, it has certain drawbacks. It may cause us to lose the polynomial complexity we had. For example, consider the following grammar:

```
:- auto_table.
s --> b,[c]

b --> b,b.
b --> [a].
```

Here nonterminal `s` generates a list of "a"s followed by a single "c". If we compile this grammar with XSB, using the `auto_table` directive, we get a program that will recognize strings correctly in $O(n^3)$ time. But if we add parameters to construct the parse tree, thusly:

```
:- auto_table.
s(s1(B,c)) --> b(B), [c].

b(b1(B1,B2)) --> b(B1), b(B2).
b(b2(a)) --> [a].
```

it may take exponential time. Now the string $a^n c$ has exponentially many parses (all bracketings of the length $n$ string of "a"s), so it will clearly take exponential time to produce them all. This is not so much of a problem; if there are exponentially many, there's no way to produce them all without taking exponential time. However, if we try to recognize the string $a^n$, this will take exponential time to report failure. This is because the system will construct all the (exponentially many) initial parses from the nonterminal "b" and then each time fail when it doesn't find a terminating "c" in the input string.

One might think that we could easily just maintain two versions of the grammar: one with no parameters to do recognition, and one with a parse-tree parameter. Then we'd first recognize and if there were no parses, we'd simply report failure, and if there were parses, we'd reprocess the input string, this time using the parsing version of the grammar. But this doesn't always work either. For example, say we added a few rules to our grammar above to obtain:

```
:- auto_table.
s --> b,[c].
s --> g,[d].

b --> b,b.
b --> [a].

g --> g, [a].
g --> [a].
```

Here, the input string of `[a,a,a,a,a,a,d]` has only one parse, but naively parsing it with the parse-annotated grammar will construct exponentially many initial segments of parses, which come from the first rule and the rules for the nonterminal "b". So if we are serious about this problem, we must be a little more sophisticated.

We still use a recognizer, but we must mix calls to it throughout the parser. So we'll assume we have the recognizer given just previously and we want an efficient parser. We must mix calls to the recognizer in with the calls to the parsing procedures. One problem is that the DCG syntax does not support what we need, so we will have to write the routines directly above:

```
:- auto_table.
s(s(P,c),S0,S) :-
       b(S0,S1), 'C'(S1,c,S),
       b(P,S0,S1).

s(s(P,d),S0,S) :-
       g(S0,S1), 'C'(S1,d,S),
       g(P,S0,S1).

b(b(P1,P2),S0,S) :-
       b(S0,S1), b(S1,S),
       b(P1,S0,S1), b(P2,S1,S).
b(a,S0,S) :-
       'C'(S0,a,S).

g(g(P),S0,S) :-
       g(S0,S1), 'C'(S1,a,S),
       g(P,S0,S1).
g(a,S0,S) :-
       'C'(S0,a,S).
```

Here for each rule, we first use the recognizer to see whether the rule will succeed. After we know it will succeed and have computed the exact substring that each nonterminal spans, then for each nonterminal we invoke the parsing routine to construct its parse.

Now as coded here, there will be multiple tabling: the `auto_table` directive will cause b/2, g/2, b/3, and g/3 all to be tabled. However, we don't really have to table the parsing nonterminals b/3 and g/3. Since they are always called with both the starting and ending string position known, they will not loop. This program has the desired property, that either it will fail in cubic time or it will succeed in cubic time and produce each parse in linear time.

## 4.6 Computing First Sets of Grammars

The previous examples show that XSB can process grammars efficiently, using them to determine language membership and the structure of strings. But XSB can also do other kinds of grammar processing. In the following, we use XSB to compute the FIRST sets of a grammar.

$FIRST(\alpha)$, for any string of terminals and nonterminals $\alpha$, is defined to be the set of terminals that begin strings derived from $\alpha$, and if $\alpha$ derives the empty string then the empty string is also in $FIRST(\alpha)$. $FIRST_k(\alpha)$ is the generalization to length $k$ strings of terminals that are prefixes of strings derived from $\alpha$.

We will assume that a grammar is stored in a predicate `==>/2`, with the head of a rule as an atomic symbol and the body of a rule as a list of symbols. Nonterminals are assumed to be those symbols for which there is at least one rule with it as its head. (`==>/2` is declared as an infix operator.)

The predicate `first(SF,K,L)` is true if the list SF of grammar symbols derives a string whose first K terminal symbols are L.

```
% The definition of FIRST:
% first(SentForm,K,FirstList) computes firsts for a context-free grammar.
:- table first/3.
first(_,0,[]).
first([],K,[]) :- K>0.
first([S|R],K,L) :- K>0,
    (S ==> B),
    first(B,K,L1),
    length(L1,K1),
    Kr is K - K1,
    first(R,Kr,L2),
    append(L1,L2,L).
first([S|R],K,[S|L]) :- K>0,
    \+ (S ==> _),    % S is a terminal
```

```
    K1 is K-1,
    first(R,K1,L).
```

The first rule says that the empty string is in $FIRST_0(\alpha)$ for any $\alpha$. The second rule says that the empty string is in $FIRST_k(\alpha)$ for $\alpha$ being the empty sequence of grammar symbols. The third rule handles the case in which the sequence of grammar rules begins with a nonterminal, `S`. It takes a rule beginning with S and gets a string `L1` (of length `K1` $\leq$ `K`) generated by the body of that rule. It gets the remaining `K`$-$`K1` symbols from the rest of the list of input grammar symbols. And the fourth rule handles terminal symbols.

This is a relatively simple declarative (and constructive) definition of $FIRST_k$. But without the table declaration it would not run for many grammars. Any left-recursive grammar would cause this definition to loop in Prolog.

## 4.7 Linear Parsing of LL(k) and LR(k) Grammars

(This section involves a more advanced topic in XSB programming, metainterpretation. It may be helpful to read the later section on metainterpreters if the going gets tough here.)

As discussed above, parsing context-free grammars with tabling results in an Earley-type parser. This is the parser we get when we write DCGs. We can also write an XSB program that takes a grammar as input (defined in a database predicate as in the example of `first/3`) and a string (defined by the database `word/3` predicate) and succeeds if the grammar accepts the string. With such processing we can compute and use the $FIRST$ sets to make the grammar processing more deterministic. This is similar to what is done in LL(k) and LR(k) parsing, but there the emphasis is on compile-time analysis and complete determinacy. Here the approach is more interpretive and supports nondeterminacy. However, if the grammars are indeed of the appropriate form (LL(k) or LR(k)), the corresponding interpreters presented here will have linear complexity.

It is very easy to write a simple context-free grammar parser in XSB. Again, we assume that the grammar is stored in facts of the form `NT ==> Body` where `NT` is a nonterminal symbol and `Body` is a list of terminals and nonterminals.

```
:- table parse/3.

% parse(Sym,S0,S) if symbol Sym generates the string from S0 to S.
parse(Sym,S0,S) :-
    word(S0,Sym,S).
parse(Sym,S0,S) :-
    (Sym ==> Body),
    parseSF(Body,S0,S).

% parseSF(SF,S0,S) if sentential form SF generates the string from S0 to S.
```

```
parseSF([],S,S).
parseSF([Sym|Syms],S0,S) :-
    parse(Sym,S0,S1),
    parseSF(Syms,S1,S).
```

The predicate `parse/3` recognizes strings generated by a single grammar symbol; the first clause recognizes terminal symbols directly and the second clause uses `parseSF/3` to recognize strings generated by the sentential form that makes up the body of a rule for a nonterminal. `parseSF/3` simply maps `parse/3` across the sequence of grammar symbols in its sentential form argument.

Were we not to add the table declaration, we would get a recursive descent recognizer. But with the table declaration, we get the Earley-type recognizer of XSB as described above. The tabling reflects right through the programmed recognizer.

Next we add look-ahead to this recognizer by computing `FIRST` sets and making calls only when the next symbols are in the first set of the sentential form to be processed. This will give us an LL(k)-like recognizer. Hovever, the form of `FIRST` we need here is slightly different from the one above. Here we want to include the context in which a `First` set is computed. For example, we may want to compute the `FIRST_2` set of a symbol `N`, but `N` only generates one symbol. The definition of `first` above would return a list of length one for that symbol. Here we want to take into account the context in which `N` is used. For example, we may know that in the current context `N` is immediately followed by another nonterminal `M`, and we know the `FIRST` of `M`. Then we can compute the `FIRST_2` of `N` in the following context of `M` by extending out the one symbol in `first` of `N` with symbols in `FIRST` of `M`.

The following definition of `firstK/3` computes such first sets.

```
:- table firstK/3.

% firstK(+SF,+Follow,-First), where K = length(Follow)
firstK([],Follow,Follow).
firstK([Sym|SF],Follow,First) :-
    firstK(SF,Follow,SymFollow),
    ((Sym ==> Body)
     ->   firstK(Body,SymFollow,First)
     ;    First = [Sym|FirstTail]
          append(FirstTail,[_],SymFollow),
    ).
```

The predicate `firstK/3` takes a sentential form `SF`, a follow string `Follow`, and returns in `First` a first string of `SF` in the context of the follow string. The value of $k$ is taken to be the length of the list `Follow`; this will be the length of the look-ahead.

We can now extend our parser to have a top-down look-ahead component, similar to an LL(k) recognizer:

```
TEST THIS SUCKER!
:- table parseLL/4.    % without this, it is an LL(k)-like parser,
                        % but with it, what is it???

% parseLL(Sym,Follow,S0,S) if symbol Sym generates the string from S0 to S.
parseLL(Sym,Follow,S0,S) :-
    (Sym ==> Body)
     -> firstK(Body,Follow,First),
        next_str(First,S0),       % do the look-ahead, continuing only if OK
        parseLLSF(Body,Follow,S0,S)
     ;  word(S0,Sym,S).

% parseLLSF(SF,Follow,S0,S) if sentential form SF generates the string from S0 to S.
parseLLSF([],_Follow,S,S).
parseLLSF([Sym|Syms],Follow,S0,S) :-
    firstK(Syms,Follow,SymFollow),
    parseLL(Sym,SymFollow,S0,S1),
    parseLLSF(Syms,Follow,S1,S).

next_str([],_).
next_str(['$'|_],S) :- \+ word(S,_,_).  % $end of string
next_str([Sym|Syms],S) :- word(S,Sym,S1),next_str(Syms,S1).
```

The predicate `parseLL/4` recognizes a string generated by a grammar symbol, using lookahead. The condition tests whether `Sym` is a nonterminal, and if it is and can be rewritten as `Body`, it checks to see whether the next symbols in the input string belong to the first set of the body of that rule. If not, it needn't process that rule because it cannot succeed. For an LL(k) grammar, only one rule will ever apply; the others all will be filtered out by this lookahead. So in this case a rule is never tried unless it is the only rule that might lead to a parse. If the symbol is a terminal, it simply checks to see whether the symbol matches the input.

`parseLLSF/4` maps `parseLL/4` across a sequence of grammar symbols in a sentential form. It uses `firstK/3` to compute the follow strings of a symbol, which are needed in `parseLL` to compute the first strings in the correct context.

In this LL(k)-like parser, we tested that the next symbols were in the first set just before we called `parseLLSF`. We could also check the lookahead just before returning. This gives us an LR(k)-like parser, as follows:

```
% parseLR with tabling to get an lr(k)-like algorithm.

:- table parse/4.
parseLR(Sym,Follow,Str0,Str) :-
    word(Str0,Sym,Str),
    next_str(Follow,Str).
```

```
parseLR(Sym,Follow,Str0,Str) :-
    (Sym ==> RB),
    parseLRSF(RB,Follow,Str0,Str).

parseLRSF([],Follow,Str,Str) :-
    next_str(Follow,Str).
parseLRSF([Sym|SF],Follow,Str0,Str) :-
    firstK(SF,Follow,SymFollow),
    parseLR(Sym,SymFollow,Str0,Str1),
    parseLRSF(SF,Follow,Str1,Str).
```

For this parser, we compute a follow string for a sentential form, and only return from parsing that sentential form if the next input symbols match that follow string. For an LR(k) grammar, the parser will not fail back over a successful return (unless the entire string is rejected.) This allows the parser to work in linear time.

The above program was written as a Prolog program, but we can write the identical program as a DCG and have the DCG transformation put in the string variables, as follows:

```
:- table parseLR/4.
parseLR(Sym,Follow) --> [Sym], look(Follow).
parseLR(Sym,Follow) -->
    {(Sym ==> RHS)},
    parseLRSF(RHS,Follow).

parseLRSF([],Follow) --> look(Follow).
parseLRSF([Sym|SF],Follow) -->
    {firstK(SF,Follow,SymFollow)},
    parseLR(Sym,SymFollow),
    parseLRSF(SF,Follow).

look([]) --> [].
look([Word|Words]), [Word] --> [Word], look(Words).
```

This recognizer differs from an LR(k) recognizer in that it computes the look-ahead strings as needed. Also it processes each look-ahead string separately; i.e., `Follow` is a single string, not a set of strings. In an LR(k) recognizer, the lookahead tables are computed once and stored.

## 4.8 Parsing of Context Sensitive Grammars

Another more powerful form of grammars is the class of context sensitive grammars. They contain rules that have strings on both the left-hand-side and the right-hand-side, as opposed to context-free rules which require a single symbol on the left-hand-side. A constraint on context sensitive rules is that the length of the string of the left-hand side is at least one, and is less than or equal to the length of the string on the right-hand side. (Without this constraint, one gets full Turing computability and the recognition problem is undecidable.) As a simple example, consider the following context sensitive grammar:

```
1. S --> aSBC
2. S --> aBC
3. CB --> BC
4. aB --> ab
5. bB --> bb
6. bC --> bc
7. cC --> cc
```

This grammar generates all strings consisting of a nonempty sequence of a's followed by the same number of b's followed by the same number of c's. Consider the following derivation:

```
S
aSBC      rule 1
aaBCBC    rule 2
aaBBCC    rule 3
aabBCC    rule 4
aabbCC    rule 5
aabbcC    rule 6
aabbcc    rule 7
```

Even though in this simple example the rules fire in order, it's not difficult to see that rule 3 will have to fire enough times to move all the C's to the right over the B's, and then rules 4-7 will fire enough times to turn all the nonterminals into terminals.

The question now is how to represent this in XSB. We can think of the XSB DCG rules as running on a graph that starts as a linear chain representing the input string. Then each DCG context-free rule tells how we can add edges to that linear graph. For example, a DCG rule `a --> b,c.` tells us that if there is an arc from node X to node Y labeled by b and also an arc from node Y to node Z labeled by c, then we should add an arc from node X to node Z labeled by a. So the DCG rule in Prolog, `a(S0,S) :- b(S0,S1),c(S1,S).`, read right-to-left, says explicitly and directly that if there is an arc from S0 to S1 labeled b and an arc from S1 to S labeled c, then there is an arc from S0 to S labeled by a. We can think of DCG rules as rules that add labeled arcs to graphs. This is exactly the way that chart-parsing is understood.

Now we can extend this way of understanding logic grammars to context-sensitive rules. A context-sensitive rule, with say two symbols on the left-hand-side, can be seen also as a graph-generating rule, but in this case it must introduce a new node as well as new arcs. So for example, a rule such as `AB --> CD`, when it sees two adjacent edges labeled C and D, should introduce a new node and connect it with the first node of the C-arc labeling it A, and also connect it to the final node of the D-arc, labeling that new arc with B. So we add two new XSB rules for a context sensitive rule such as `AB --> CD`, as follows:

```
    a(S0,p1(S0)) :- c(S0,S1), d(S1,S).
    b(p1(S0),S) :- c(S0,S1), d(S1,S).
```

which explicitly add the arcs and nodes. We have to introduce a new name for the new node. We choose to identify the new nodes by using a functor symbol that uniquely determines the rule and left-hand internal position, and pairing it with the name of the initial node in the base arc. So in this case, p1 uniquely identifies the (only) internal position in the left-hand-side of this rule. Other rules, and positions, would have different functors to identify them uniquely.

Now we can represent the context sensitive grammar above using the following XSB rules:

```
:- auto_table.

s(S0,S) :- word(S0,a,S1),s(S1,S2),b(S2,S3),c(S3,S).
s(S0,S) :- word(S0,a,S1),b(S1,S2),c(S2,S).

c(S0,p0(S0)) :- b(S0,S1),c(S1,_S).
b(p0(S0),S) :- b(S0,S1),c(S1,S).

word(S0,a,p1(S0)) :- word(S0,a,S1),word(S1,b,_S).
b(p1(S0),S) :- word(S0,a,S1),word(S1,b,S).

word(S0,b,p2(S0)) :- word(S0,b,S1),word(S1,b,_S).
b(p2(S0),S) :- word(S0,b,S1),word(S1,b,S).

word(S0,b,p3(S0)) :- word(S0,b,S1),word(S1,c,_S).
c(p3(S0),S) :- word(S0,b,S1),word(S1,c,S).

word(S0,c,p4(S0)) :- word(S0,c,S1),word(S1,c,_S).
c(p4(S0),S) :- word(S0,c,S1),word(S1,c,S).

% define word/3 using base word (separation necessary)
word(X,Y,Z) :- base_word(X,Y,Z).

% parse a string... assert words first, then call sentence symbol
parse(String) :-
    abolish_all_tables,
    retractall(base_word(_,_,_)),
    assertWordList(String,0,Len),
    s(0,Len).
```

```
% assert the list of words.
assertWordList([],N,N).
assertWordList([Sym|Syms],N,M) :-
    N1 is N+1,
    assert(base_word(N,Sym,N1)),
    assertWordList(Syms,N1,M).
```

We can run this grammar to parse input strings as follows:

```
warren% xsb
XSB Version 1.7.2 (7/10/97)
[Sun, optimal mode]
| ?- [csgram].
[csgram loaded]

yes
| ?- parse([a,a,b,b,c,c]).
++Warning: Removing incomplete tables...

yes
| ?- parse([a,a,a,b,b,c,c,c]).

no
| ?- parse([a,a,a,a,b,b,b,b,c,c,c,c]).
++Warning: Removing incomplete tables...

yes
| ?-
```

So now we have generalized DCG's to include processing of context-sensitive grammars and languages. The builtin DCG notation doesn't support context sensitive languages, but we can write the necessary rules directly as XSB rules, as we did above. It is interesting to note that the XSB rules we generate for a single context-sensitive rule all have the same body, and that the logical implications

```
p <- r & s.
q <- r & s.
```

are logically equivalent to the single implication:

```
p & q <- r & s.
```

So it would be very natural to extend the Prolog notation to support "multi-headed" rules, which would be compiled to a set of "single-headed", i.e., regular Prolog, rules. Were we to do this, we could write the context-sensitive rule:

```
AB --> CD
```

as the single (multi-headed) XSB rule:

```
a(S0,p1(S0)), b(p1(S0),S) :- c(S0,S1), d(S1,S).
```

which looks very much like the original context sensitive rule. This suggests how we might want to extend the DCG notation to support context-sensitive rules through the support of multi-headed rules.

# Chapter 5

# Automata Theory in XSB

In this chapter we explore how we can use XSB to understand and implement some of the formal systems of automata theory. We will begin by defining finite state machines (FSM), and exploring executable specifications in XSB of string acceptance, epsilon-free machines, deterministic machines and other interesting notions.

## 5.1 Finite State Machines

We represent a finite state machine using three relations:

1. m(MachineName,State,Symbol,TargetState) which describes the transition relation for a machine, with MachineName being the name of the machine (to allow us to represent many machines using the same relation), State is a state of the FSM, Symbol is an input symbol, and TargetState is a state to which the machine transitions from State on seeing Symbol.
2. mis(MachineName,InitialState) where InitialState is the initial state of the FSM named MachineName.
3. mfs(MachineName,FinalState) where FinalState is a final state of the FSM named MachineName.

By including a MachineName in each tuple, we can use these relations to represent a number of different FSM's. This will be convenient later.

We will use the symbol `''` (the atom with the empty name) as a pseudo-symbol to represent epsilon transitions, transitions a machine can make without consuming any input symbol.

For example, the following relations represent the machine pictured in Figure 5.1, which we will call m0s1s2s, since it recognizes strings made up of a string of 0's followed by a string of 1's

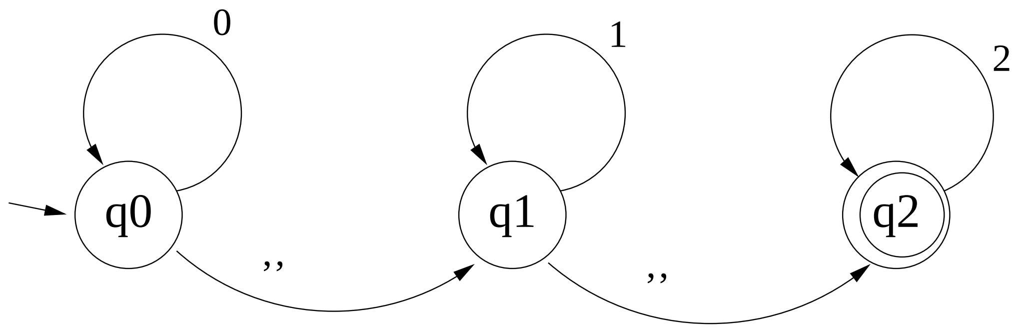


Figure 5.1: The finite state machine: m0s1s2s

followed by a string of 2's:

```
m(m0s1s2s,q0,0,q0).
m(m0s1s2s,q0,'',q1).
m(m0s1s2s,q1,1,q1).
m(m0s1s2s,q1,'',q2).
m(m0s1s2s,q2,2,q2).

mis(m0s1s2s,q0).

mfs(m0s1s2s,q2).
```

We represent strings with two relations. Again we will name strings for convenience.

1. string(StringName,Index,Symbol,Index1) where StringName is the name of the string, Symbol is the Index1-th symbol in the string and Index is Index1-1. For example, the string "001112", which we will call s1, would be represented by:

   ```
   string(s1,0,0,1).
   string(s1,1,0,2).
   string(s1,2,1,3).
   string(s1,3,1,4).
   string(s1,4,1,5).
   string(s1,5,2,6).
   ```

   and string "021", which we'll name s2, would be represented by:

   ```
   string(s2,0,0,1).
   string(s2,1,2,2).
   string(s2,2,1,3).
   ```

2. stringlen(StringName,Length) where Length is the length of the string named StringName. For example, for the previous examples, we would have the facts:

```
stringlen(s1,6).
stringlen(s2,3).
```

A FSM is said to accept a string if it executes in the following way: it starts in the initial state, and makes a transition to the next state along a path labeled by the symbol it is looking at. It consumes that symbol and then makes another transition based on the next symbol in the string. It continues in this way until all symbols are consumed, and if the machine is then in a final state, the string is accepted; otherwise it is rejected. If there is an epsilon-transition from one state to another, the machine can make such a transition without consuming a symbol of the input.

Now we can easily write a specification in XSB that defines when a machine accepts a string, as follows:

```
:- auto_table.

% A machine accepts a string if the machine starts in the initial state,
%    recognizes the string, ending in a final state and has consumed the
%    entire string.
accept(MachineName,StringName) :- mis(MachineName,StateStart),
        recognize(MachineName,StringName,StateStart,StateFinal,0,StringFinal),
        mfs(MachineName,StateFinal),
        stringlen(StringName,StringFinal).

% recognize(MachineName,StringName,MState0,MState,SLoc0,SLoc) is true
% if machine MachineName started in state MState0 can transition to
% state MState by recognizing the substring from location SLoc0 to SLoc
% of the string named StringName.

% The empty input string
recognize(_,_,MState,MState,SLoc,SLoc).
% regular transitions
recognize(MachineName,StringName,MState0,MState,SLoc0,SLoc) :-
        string(StringName,SLoc0,Symbol,SLoc1),
        m(MachineName,MState0,Symbol,MState1),
        recognize(MachineName,StringName,MState1,MState,SLoc1,SLoc).
% Epsilon transitions
recognize(MachineName,StringName,MState0,MState,SLoc0,SLoc) :-
        m(MachineName,MState0,'',MState1),
        recognize(MachineName,StringName,MState1,MState,SLoc0,SLoc).
```

The definition of `accept` says that a machine accepts a string if StateStart is the initial state of the indicated machine, and the machine transits from StateStart to StateFinal while recognizing the string starting from 0 and ending at StringFinal, and StateFinal is a final state of the machine, and StringFinal is the length of the string.

The definition of `recognize/6` describes how a machine moves through its states while recognizing (or generating) a sequence of symbols. The first clause says that for any machine and any string, when the machine stays in the same state, no symbols of the string are processed. The second clause says that a machine moves from MState0 to MState recognizing a substring if the next symbol in the substring is Symbol, and there is a transition of the current machine on that Symbol that takes the machine from MState0 to MState1, and the machine recognizes the rest of the substring from that state MState1 getting to MState. The third clause handles epsilon transitions; it says that a machine moves from MState0 to MState recognizing a substring if there is an epsilon transition from MState0 to a MState1 and the machine recognizes the entire string from MState1 to MState.

For example, accept(m0s1s2s,s1) succeeds, but accept(m0s1s2s,s2) fails:

```
warren% xsb
XSB Version 1.6.0 (96/6/15)
[sequential, single word, optimal mode]
| ?- [automata].
[automata loaded]

yes
| ?- accept(m0s1s2s,s1).
++Warning: Removing incomplete tables...

yes
| ?- accept(m0s1s2s,s2).

no
| ?-
```

This is a right-recursive definition of `recognize/6`, so it might seem that this should not need tabling. And indeed for this particular machine and string, Prolog would evaluate this definition just fine. However, there are machines for which tabling is required. Can you give an example of such a machine?

Also, it is possible to give this specification in a left-recursive manner. That is also an exercise for the reader.

### 5.1.1 Intersection of FSM's

Given two FSM's, one can ask the question as to whether there is a string that both machines accept, that is, whether the intersection of the languages accepted by the two machines is non-empty.

This turns to be possible, and not very difficult. As a matter of fact, we've already essentially written a program that does this. You might have noticed that our representations of strings and machines are actually very similar. In fact, a string can be understood as a FSM, simply by viewing the string/4 predicate as a machine transition predicate. In this case the string's states are integers, the initial state is 0 and the final state is the string length. Viewed this way, a string is simply a FSM that recognizes exactly that one string.

Viewed in this way, the verb—accept/2— predicate above, which we wrote as determining whether a FSM accepts a string, can be trivially modified to determine whether two machines accept languages with a non-empty intersection. We leave it to the reader to modify the definition of `accept/2` to check intersection, and to test it with several examples.

### 5.1.2 Epsilon-free FSM's

Two FSM's are said to be *equivalent* if they accept exactly the same set of strings. Given any nondeterministc FSM, it is always possible to find an equivalent one that has *no* epsilon transitions. In fact, given a machine as defined and represented above, we can easily define such an equivalent epsilon-free machine. So given a machine named *mach* and defined in m/4, mis/2 and mfs/2, we will define the transitions, initial state and final state for its epsilon-free version named *efree(mach)* as follows:

```
% epsilon-free machines

% first define emoves as any sequence of epsilon transitions
emoves(_,State,State).
emoves(Mach,State0,State) :-
        emoves(Mach,State0,State1),
        m(Mach,State1,'',State).


% define the transition relation of the efree machine
m(efree(Mach),State,Symbol,TargState) :-
        emoves(Mach,State,State1),
        m(Mach,State1,Symbol,State2),
        Symbol \== '',
        emoves(Mach,State2,TargState).

% define the initial and final states of the efree machine
mis(efree(Mach),IS) :- mis(Mach,IS).
mfs(efree(Mach),FS) :- mfs(Mach,FS1),emoves(Mach,FS,FS1).
mfs(efree(Mach),FS) :- mfs(Mach,FS1),emoves(Mach,FS1,FS).
```

The predicate `emoves/3` defines for any machine the set of pairs of states such that the machine can move from the first state to the second state without consuming any symbols in the input string. Then with this definition, the rule defining transitions says that an epsilon-free machine can move from State to TargState on Symbol if it can move from State to State1 using only epsilon moves, can move from State1 to State2 on seeing Symbol (which is *not* epsilon) and can make epsilon moves from State2 to TargState.

The initial state of the epsilon-free machine is exactly the initial state of the original machine. The final state of the epsilon-free machine is any state from which you can get to a final state of the original machine using only epsilon transitions, or any state you can get to from a final state using only epsilon transitions.

For example:

```
warren% xsb
XSB Version 1.6.0 (96/6/15)
[sequential, single word, optimal mode]
| ?- [automata].
[automata loaded]

yes
| ?- m(efree(m0s1s2s),So,Sym,Ta),writeln(m(efree(m0s1s2s),So,Sym,Ta)),fail.
m(efree(m0s1s2s),q0,0,q0)
m(efree(m0s1s2s),q0,0,q1)
m(efree(m0s1s2s),q0,0,q2)
m(efree(m0s1s2s),q1,1,q1)
m(efree(m0s1s2s),q1,1,q2)
m(efree(m0s1s2s),q2,2,q2)
m(efree(m0s1s2s),q0,1,q2)
m(efree(m0s1s2s),q0,1,q1)
m(efree(m0s1s2s),q1,2,q2)
m(efree(m0s1s2s),q0,2,q2)

no
| ?- mis(efree(m0s1s2s),IS),writeln(mis(efree(m0s1s2s),IS)),fail.
mis(efree(m0s1s2s),q0)

no
| ?- mfs(efree(m0s1s2s),FS),writeln(mfs(efree(m0s1s2s),FS)),fail.
mfs(efree(m0s1s2s),q2)
mfs(efree(m0s1s2s),q0)
mfs(efree(m0s1s2s),q1)

no
| ?-
```

The diagram for `efree(m0s1s2s)` is shown in Figure 5.2.

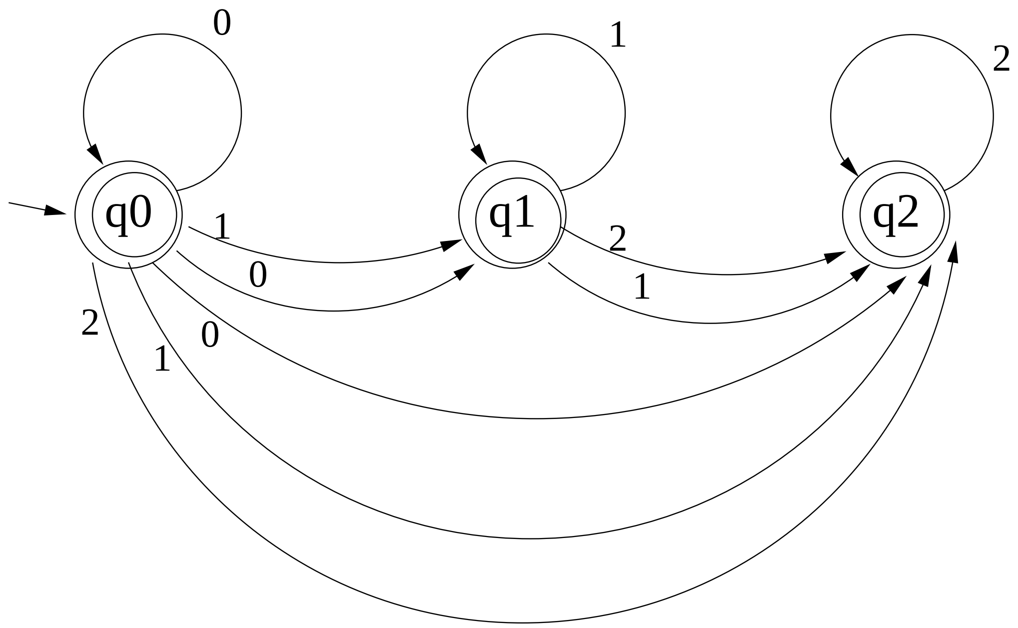


Figure 5.2: The finite state machine: efree(m0s1s2s)

### 5.1.3 Deterministic FSM's

A deterministic FSM is a machine such that for every state for any symbol there is at most one transition from that state labeled with that symbol (and there are no epsilon transitions.) This means that there is never a choice in how the machine is to proceed when it sees a symbol: there will be at most one state to move to given that symbol. The question arises as to whether given an arbitrary FSM there is always an equivalent deterministic FSM, i.e., a deterministic FSM that accepts the same language, i.e., exactly the same set of strings.

The answer turns out to be "yes", and it is not difficult to see why. Given a nondeterministic (ND) machine, we can construct a deterministic machine each of whose states corresponds to a set of states in the nondeterministic machine. The idea is that, after seeing a string, the deterministic machine will be in a state corresponding to a set of ND states just in case the ND FSM could be in any one of the ND states after seeing the same string. As a very trivial example, say we had a ND machine with three states: $q1$, $q2$, $q3$, with $q1$ the initial state and transitions from $q1$ to $q2$ on symbol $a$ and from $q1$ to $q3$ also on $a$. Then the deterministic machine would have two states, $\{q1\}$ and $\{q2, q3\}$ (each being a set of the original machine's states), and a transition from the first to the second on symbol $a$.

The following specification describes this construction. Rather than constucting *all* the states (which would necessarily be exponential in the number of states in the nondeterministic machine), we will only construct those that are reachable from the initial state. This may be a much smaller number. Also, for this particular specification to be constructive, we need to constrain the set of possible deterministic states in some way, and this seems a good way.

We will assume that the machine is an epsilon-free machine. If `efreemach` is the name of an epsilon-free machine then `det(efreemach)` is the name of an equivalent deterministic machine.

```
:- import member/2 from basics.
:- import tsetof/3 from setof.

% Assume Mach is an epsilon-free machine.
% A state is reachable if it is the initial state or if it can be
%   reached by one step from a reachable state.
reachable(Mach,S) :- mis(Mach,S).
reachable(Mach,S) :- reachable(Mach,S1),m(Mach,S1,_,S).

% The next state of the deterministic machine given a state and symbol
%   is the set of states of the nondeterministic machine which are a
%   next state starting from some element of the current state of the
%   deterministic machine. (Mach is assumed to be epsilon-free.)
m(det(Mach),State0,Sym,State) :-
    reachable(det(Mach),State0),
    tsetof(NDS, a_next(Mach,State0,Sym,NDS), State).

% A state is a next state if it is a next state reachable in one step
%   from some member of the current state of the deterministic machine.
a_next(Mach,DState,Sym,NDState) :-
    member(S1,DState),
    m(Mach,S1,Sym,NDState).

% The initial state is the singleton set consisting of the initial
%   state of the nondeterministic machine.
mis(det(Mach),[IS]) :- mis(Mach,IS).

% A final state is a reachable deterministic state that contains some
%   final state.of the nondeterministic machine.
mfs(det(Mach),FS) :- mfs(Mach,NFS), reachable(det(Mach),FS),member(NFS,FS).
```

Now we can use this specification to find a deterministic machine that is equivalent to the nondeterministic machine $m0s1s2s$:

```
| ?- m(det(efree(m0s1s2s)),S,Sy,T),writeln(m(det(m0s1s2s),S,Sy,T)),fail.
m(det(m0s1s2s),[q0],0,[q0,q1,q2])
m(det(m0s1s2s),[q0],1,[q1,q2])
m(det(m0s1s2s),[q0],2,[q2])
m(det(m0s1s2s),[q0,q1,q2],0,[q0,q1,q2])
m(det(m0s1s2s),[q0,q1,q2],1,[q1,q2])
m(det(m0s1s2s),[q0,q1,q2],2,[q2])
```

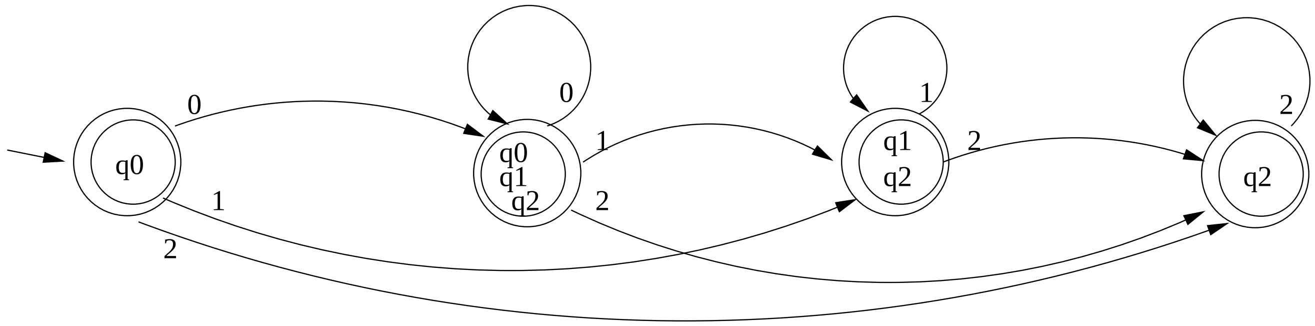


Figure 5.3: The finite state machine: det(efree(m0s1s2s))

```
m(det(m0s1s2s),[q1,q2],1,[q1,q2])
m(det(m0s1s2s),[q1,q2],2,[q2])
m(det(m0s1s2s),[q2],2,[q2])

no
| ?- mis(det(efree(m0s1s2s)),S),writeln(mis(det(m0s1s2s),S)),fail.
mis(det(m0s1s2s),[q0])

no
| ?- mfs(det(efree(m0s1s2s)),S),writeln(mfs(det(m0s1s2s),S)),fail.
mfs(det(m0s1s2s),[q1,q2])
mfs(det(m0s1s2s),[q0,q1,q2])
mfs(det(m0s1s2s),[q2])
mfs(det(m0s1s2s),[q0])

no
| ?-
```

The diagram for `det(efree(m0s1s2s))` is shown in Figure 5.3.

### 5.1.4 Complements of FSM's

One may also want to construct a machine that accepts the complement of the set of strings a given machine accepts. This turns out to be possible and reasonably easy to do, once we straighten out a minor issue. Up to now we have been mostly ignoring the alphabet of symbols that make up the strings. This has been fine for accepting strings, since if a symbol never appears in any transition, the machine can never accept any string containing that symbol. But when we talk about strings that a machine rejects, we have to give the alphabet with respect to which to take the complement. For example, what is the complement of the language that consists of all strings consisting of only $a$'s and $b$'s? It is the empty set if the alphabet is $\{a, b\}$, but if the alphabet is $\{a, b, c\}$, then it is the set of all strings over $\{a, b, c\}$ that contain at least one $c$.

Using our current representation, we will assume that the alphabet of a given machine is the set of symbols that appear on *some* transition of that machine. While this seems to be a reasonable assumption, note that it could be incorrect for the second example of the complement of all strings of $a$'s and $b$'s given above. (If we were committed to being very precise, we could add an XSB relation which for each machine defined the set of symbols in its alphabet.)

The basic idea for generating the complement machine is simply to take the original machine but to take the complement of its final states to the be final states of the complement-accepting machine. But there are a couple of things we have to guarantee before this works. First the original machine must be deterministic, since otherwise for a given string it might end in both a final and a nonfinal state. In this case it should be rejected by the machine accepting the complement language, but simply inverting the final and nonfinal states would result in it being accepted. So we will always start with a deterministic machine. Second, we have to be sure that the machine gets to some state on *every* input. That is, there must always be a transition that can be taken, regardless of the symbol being scanned. That is, every state must have an outgoing transition for every symbol in the alphabet. And here is where the importance of the alphabet is clear.

So we will separate our construction into two parts: first we will complete the machine (assuming it is deterministic) by adding transitions to a new state, called "sink," when there are no transitions on some symbols; and second we will complement a completed machine.

```
% completed machine
% A symbol is in the alphabet of a machine if it appears in a non-epsilon
%    transition.  (Note that this is our convention and for some machines
%    could be wrong.)
alphabet(Mach,Sym) :-
    m(Mach,_,Sym,_),
    Sym \== ''.

% S is a (possibly reachable) state in machine if it's initial or has an
%    incoming edge.
is_state(Mach,S) :- m(Mach,_,_,S).
is_state(Mach,S) :- mis(Mach,S).

% The initial states and final states of the completed machine are the
%   same as the original machine.
mis(completed(Mach),IS) :- mis(Mach,IS).
mfs(completed(Mach),FS) :- mfs(Mach,FS).

% Assume Mach is deterministic
% There is a transition to ''sink'' if there is no other transition on
%    this symbol from this state.
m(completed(Mach),So,Sy,sink) :-
    is_state(Mach,So),
    alphabet(Mach,Sy),
```

```
    tnot(isatransition(Mach,So,Sy)).
% Machine transitions from sink to sink on every symbol
m(completed(Mach),sink,Sy,sink) :-
    alphabet(Mach,Sy).
% Otherwise the same as underlying machine
m(completed(Mach),So,Sy,Ta) :-
    m(Mach,So,Sy,Ta).

% There is a transition if there's a state it transits to.
isatransition(Mach,So,Sy) :-
    m(Mach,So,Sy,_).
```

Now given a completed machine, we can easily generate the complement machine simply by interchanging final and nonfinal states:

```
% complement machine
% Asume machine is completed and deterministic.
% The transitions of the complement machine are the same.
m(complement(Mach),So,Sy,Ta) :- m(Mach,So,Sy,Ta).

% The initial state of the complement machine is the same.
mis(complement(Mach),S) :-  mis(Mach,S).

% A state is a final state of the complement if it is NOT the final state
%    of the underlying machine.
mfs(complement(Mach),S) :-
    is_state(Mach,S),
    tnot(mfs(Mach,S)).
```

With these definitions, we can compute the complement of our simple machine m0s1s2s:

```
| ?- [automata].
[automata loaded]

yes
| ?- m(complement(completed(det(efree(m0s1s2s)))),S,Sy,T),
     writeln(m(complement(m0s1s2s),S,Sy,T)),fail.
m(complement(m0s1s2s),[q2],1,sink)
m(complement(m0s1s2s),[q2],0,sink)
m(complement(m0s1s2s),[q1,q2],0,sink)
m(complement(m0s1s2s),sink,2,sink)
m(complement(m0s1s2s),sink,1,sink)
m(complement(m0s1s2s),sink,0,sink)
```

```
m(complement(m0s1s2s),[q1,q2],2,[q2])
m(complement(m0s1s2s),[q1,q2],1,[q1,q2])
m(complement(m0s1s2s),[q0,q1,q2],2,[q2])
m(complement(m0s1s2s),[q0,q1,q2],1,[q1,q2])
m(complement(m0s1s2s),[q0,q1,q2],0,[q0,q1,q2])
m(complement(m0s1s2s),[q2],2,[q2])
m(complement(m0s1s2s),[q0],2,[q2])
m(complement(m0s1s2s),[q0],1,[q1,q2])
m(complement(m0s1s2s),[q0],0,[q0,q1,q2])

no
| ?- mis(complement(completed(det(efree(m0s1s2s)))),S),
     writeln(mis(complement(m0s1s2s),S)),fail.
mis(complement(m0s1s2s),[q0])

no
| ?- mfs(complement(completed(det(efree(m0s1s2s)))),S),
     writeln(mfs(complement(m0s1s2s),S)),fail.
mfs(complement(m0s1s2s),sink)

no
| ?-
```

Given these definitions, we can now write a specification that determines when two machines accept the same language. With complement and intersection, we can define subset: $A \subseteq B \iff A \cap \overline{B} = \emptyset$. We leave it as an exercise for the reader to write and test such a specification.

### 5.1.5 Minimization of FSM's

Another question of interest is whether a given FSM has "redundant" states. That is, is it as small as it can be or is there a smaller machine, i.e., one with fewer states, that can recognize the same language.

So the idea is, given a machine, to see whether it has redundant states. The first step is to determine whether two states in the machine are distinguishable, i.e., whether there is some string such that when the machine is started in the respective states, one computation will lead to an accepting state and the other won't. The following specification defines (and computes) distinguishable states.

```
% Assume Mach is a deterministic machine
% S1 and S2 are distinquishable if S1 is final and S2 is not.
distinguishable(Mach,S1,S2) :-
        mfs(Mach,S1),
```

```
        is_state(Mach,S2),
        tnot(mfs(Mach,S2)).
% S1 and S2 are distinquishable if S2 is final and S1 is not.
distinguishable(Mach,S1,S2) :-
        mfs(Mach,S2),
        is_state(Mach,S1),
        tnot(mfs(Mach,S1)).
% S1 and S2 are distinguishable if some symbol Sy takes them to states that
%    are distinguishable.
distinguishable(Mach,S1,S2) :-
        m(Mach,S1,Sy,T1),
        m(Mach,S2,Sy,T2),
        distinguishable(Mach,T1,T2).
```

The first two rules say that states are distinguishable if one is final and the other is not. For this we need the constraint that the initial machine be deterministic. The third rule says that states are distinguishible if there is a symbol on which they make transitions to distinguishable states.

As an example of finding distinguishable states, we can use the following machine:

```
m(dfa,a,0,b).
m(dfa,a,1,f).
m(dfa,b,0,g).
m(dfa,b,1,c).
m(dfa,c,0,a).
m(dfa,c,1,c).
m(dfa,d,0,c).
m(dfa,d,1,g).
m(dfa,e,0,h).
m(dfa,e,1,f).
m(dfa,f,0,c).
m(dfa,f,1,g).
m(dfa,g,0,g).
m(dfa,g,1,e).
m(dfa,h,0,g).
m(dfa,h,1,c).

mis(dfa,a).
mfs(dfa,c).
```

(draw a picture. Could also use the determistic version of ms0s1s2, since it has an extra state q0, I think.)

And with this machine we get the following evaluation:

```
| ?- distinguishable(dfa,S1,S2),S1@<S2,writeln(d(S1,S2)),fail.
d(d,e)
d(d,g)
d(d,h)
d(b,d)
d(b,e)
d(b,f)
d(b,g)
d(b,c)
d(a,b)
d(a,d)
d(a,f)
d(a,g)
d(a,h)
d(a,c)
d(f,g)
d(f,h)
d(e,f)
d(e,g)
d(e,h)
d(g,h)
d(c,d)
d(c,e)
d(c,h)
d(c,g)
d(c,f)

no
| ?-
```

In the query we filtered to keep only the cases in which the first state has a smaller name than the second. This was just to avoid printing out all the commutative and reflexive pairs.

What is more interesting than finding two states that are distinguishable is finding two states that are **not** distinguishable. In this case one of the states is unnecessary and can be eliminated from the machine. So using this definition of `distinguishable`, we can construct a minimal DFSM that accepts the language given by a machine by merging indistinguishable states in that machine.

```
% min (assuming the machine is deterministic), reduce by
%   indistinguishability.
m(min(Mach),So,Sy,Ta) :-
    reachable(min(Mach),So),
    member(Ss,So),
    m(Mach,Ss,Sy,T),
    tsetof(S,indistinguishable(Mach,T,S),Ta).
```

```
% The initial (final) state is the set of states indistinguishable
%   from the initial (final) state of the base machine.
mis(min(Mach),IS) :-
    mis(Mach,Bis),
    tsetof(S,indistinguishable(Mach,Bis,S),IS).
mfs(min(Mach),FS) :-
    mfs(Mach,Bfs),
    tsetof(S,indistinguishable(Mach,Bfs,S),FS).

indistinguishable(Mach,S1,S2) :-
    is_state(Mach,S1),
    is_state(Mach,S2),
    tnot(distinguishable(Mach,S1,S2)).
```

And executing this with the previous example, we get:

```
| ?- m(min(dfa),S,Sy,T),writeln(m(min(dfa),S,Sy,T)),fail.
m(min(dfa),[a,e],1,[f])
m(min(dfa),[a,e],0,[b,h])
m(min(dfa),[f],1,[g])
m(min(dfa),[f],0,[c])
m(min(dfa),[b,h],1,[c])
m(min(dfa),[b,h],0,[g])
m(min(dfa),[g],1,[a,e])
m(min(dfa),[g],0,[g])
m(min(dfa),[c],1,[c])
m(min(dfa),[c],0,[a,e])

no
| ?- mfs(min(dfa),S),writeln(mfs(min(dfa),S)),fail.
mfs(min(dfa),[c])

no
| ?- mis(min(dfa),S),writeln(mis(min(dfa),S)),fail.
mis(min(dfa),[a,e])

no
| ?-
```

(Draw state diagram) Note that the state “d” does not appear in this collapsed machine. This is because it has no in-transitions and is not the initial state, so it doesn't appear in our reduction. It is actually equivalent to state “f”, and could be merged with it.

### 5.1.6 Regular Expressions

Regular expressions are another way of specifying finite state languages, that is sets of strings of symbols. A regular expression over an alphabet $\Sigma$ is:

1. a symbol from $\Sigma$, or
2. an expression $(RE1 * RE2)$, where $RE1$ and $RE2$ are regular expressions, or
3. an expression $(RE1 + RE2)$, where $RE1$ and $RE2$ are regular expressions, or
4. an expression $@(RE)$, where $RE$ is a regular expression.

We associate with each regular expression (RE) a set of strings over $\Sigma$ (i.e., a language.) We will use XSB (Prolog) rules to define the set of strings associated with a RE. An RE will be represented as a Prolog term, and the definition below shows that we are using $*$ for concatentation, $+$ for alternation, and @ for iteration.

The following program, when given a regular expression, accepts strings that are in the language represented by that expression:

```
% Is StringName in the language represented by Exp?
reacc(Exp,StringName) :-
    reacc(Exp,StringName,0,F),
    stringlen(StringName,F).

% An atom represents itself
reacc(A,S,From,To) :- atomic(A),string(S,From,A,To).
% Concatenation of E1 and E2
reacc((E1*E2),S,From,To) :-
    reacc(E1,S,From,M),
    reacc(E2,S,M,To).
% Alternation
reacc((E1+_E2),S,From,To) :- reacc(E1,S,From,To).
reacc((_E1+E2),S,From,To) :- reacc(E2,S,From,To).
% Iteration if 0 or more occurrences
reacc(@(_E),_S,From,From).
reacc(@(E),S,From,To) :-
    reacc(@(E),S,From,Mid),
    reacc(E,S,Mid,To).
```

Now we can test whether string s1 (00112) is in the language represented by the regular expression $@(0) * @(1) * @(2)$:

```
| ?- reacc(@(0)* @(1)* @(2),s1).
++Warning: Removing incomplete tables...

yes
| ?-
```

and it is.

It turns out that regular expressions represent exactly the same languages that finite state machines do. Given a regular expression, we can construct a finite state machine that recognizes the same language (and vice versa.) So first we will construct a machine given a regular expression. To make the construction easy, we will represent machine names and states in an interesting way. Given a regular expression RE, we will use m(RE) to name the machine for recognizing the same langauge as RE. And the initial state for the constructed machine that recognizes the language of RE will be named i(RE) and the final state (there will always be exactly one in our construction) will be f(RE). Note that we are free to name the machines and states anything we want, so this choice is simply a convenience.

The following rules define the FSM given a regular expression:

```
% The machine for an atomic RE, simply transits from its initial state
%    to its final state on the given atomic symbol.
%    (All the others will be epsilon transitions.)
m(re(RE),i(RE),RE,f(RE)) :- atomic(RE).

% To recognize concatenated expressions:
% Connect the initial state of the compound expr to the initial state of
%    the first subexpr.
m(re(RE1*RE2),i(RE1*RE2),'',i(RE1)).

% Connect the final state of the first subexpr to the initial state of
%     the second subexpr.
m(re(RE1*RE2),f(RE1),'',i(RE2)).

% Connect the final state of the second subexpr to the final state of
%     the compound expr.
m(re(RE1*RE2),f(RE2),'',f(RE1*RE2)).

% And finally must add the transitions of the machines for the
%    subexpressions.
m(re(RE1*_RE2),S,Sy,T) :- m(re(RE1),S,Sy,T).
m(re(_RE1*RE2),S,Sy,T) :- m(re(RE2),S,Sy,T).

% The process is analogous for alternation.
m(re(RE1+RE2),i(RE1+RE2),'',i(RE1)).
```

```
m(re(RE1+RE2),i(RE1+RE2),'',i(RE2)).
m(re(RE1+RE2),f(RE1),'',f(RE1+RE2)).
m(re(RE1+RE2),f(RE2),'',f(RE1+RE2)).
m(re(RE1+_RE2),S,Sy,T) :- m(re(RE1),S,Sy,T).
m(re(_RE1+RE2),S,Sy,T) :- m(re(RE2),S,Sy,T).

% and for iteration
m(re(@(RE)),i(@(RE)),'',f(@(RE))).
m(re(@(RE)),i(@(RE)),'',i(RE)).
m(re(@(RE)),f(RE),'',f(@(RE))).
m(re(@(RE)),f(@(RE)),'',i(@(RE))).
m(re(@(RE)),S,Sy,T) :- m(re(RE),S,Sy,T).

% and the initial and final states are just those named i() and f().
mis(re(RE),i(RE)).
mfs(re(RE),f(RE)).
```

As an example, consider the following execution

```
| ?- m(re(a*b*c),S,Sy,T),writeln(m(re(a*b*c),S,Sy,T)),fail.
m(re(a * b * c),i(a * b * c),,i(a * b))
m(re(a * b * c),f(a * b),,i(c))
m(re(a * b * c),f(c),,f(a * b * c))
m(re(a * b * c),i(a * b),,i(a))
m(re(a * b * c),f(a),,i(b))
m(re(a * b * c),f(b),,f(a * b))
m(re(a * b * c),i(a),a,f(a))
m(re(a * b * c),i(b),b,f(b))
m(re(a * b * c),i(c),c,f(c))

no
| ?- m(det(efree(re(a*b*c))),S,Sy,T),writeln(m(re(a*b*c),S,Sy,T)),fail.
m(re(a * b * c),[i(a * b * c)],a,[f(a),i(b)])
m(re(a * b * c),[f(a),i(b)],b,[f(b),f(a * b),i(c)])
m(re(a * b * c),[f(b),f(a * b),i(c)],c,[f(c),f(a * b * c)])

no
| ?-
```

Here we first constructed the FSM that recognizes the single string *abc*, which has 9 transitions, most of them epsilon transitions. In the second query, we found the deterministic version of that machine. Notice that this dropped the number of transitions to the minimal three.

The final problem concerning FSM's and regular expressions that we will consider is that of, given a machine, constructing an equivalent regular expression.

```
re(S,T,0,Mach,RE) :- is_state(S),is_state(T),S\==T,tsetof(Sy,m(Mach,S,Sy,T),RE).
re(S,S,0,Mach,[''|RE]) :- is_state(S),tsetof(Sy,m(Mach,S,Sy,T),RE).
re(I,J,K,Mach,[RE1* @(RE2) * RE3,RE4]) :- K>0,
    K1 is K-1,
    re(I,K,K1,Mach,RE1),
    re(K,K,K1,Mach,RE2),
    re(K,J,K1,Mach,RE3),
    re(I,J,K1,Mach,RE4).
```

## 5.2 Push-Down Automata

# Chapter 6

# Dynamic Programming in XSB

Dynamic Programming is the name for a general strategy used in algorithms when one organizes the computation to be done in such a way that subproblems are evaluated only once instead of many times. With this description of the dynamic programming strategy, one can see that the tabling strategy of XSB is a *dynamic* dynamic programming strategy. That is, regardless of how the computation is structured at compile time (by the programmer), tabling ensures that subproblems are evaluated only once. So this suggests that problems amenable to dynamic programming solutions might be particularly appropriate for evaluating with XSB. This is indeed the case, and in this chapter we will see a number of examples.

These problems have a common characteristic. They all can be solved by writing down a simple specification of the problem. However, if one thinks as a Prolog programmer about the execution of the specification, it seems horrendously redundant and inefficient. But executing it with tabling declarations eliminates the redundancy and actually turns the specification into an efficient algorithm.

## 6.1 The Knap-Sack Problem

The first problem we will consider is the knap-sack problem. The idea is that we have a knap-sack and a bunch of things of various sizes to put in it. The question is whether there is a subset of the things that will fit exactly into the knap-sack. The problem can be formally stated as follows:

> Given $n$ items, each of integer size $k_i$ ($1 \leq i \leq n$), and a knap-sack of size K. 1) determine whether there is a subset of the items that sums to K. 2) Find such a subset.

We will represent the items and their sizes by using a set of facts `item/2`, where `item(3,5)` would mean that the third item is of size 5.

To determine whether there is a subset of items that exactly fill the knap-sack, we can just nondeterministicaly try all alternatives.

```
% ks(+I,+K) if there is a subset of items 1,...,I that sums to K.
ks(0,0).                % the empty set sums to 0
ks(I,K) :- I>0,         % don't include this Ith element in the knapsack
    I1 is I-1, ks(I1,K).
ks(I,K) :- I>0,         % do include this Ith element in the knapsack
    item(I,Ki), K1 is K-Ki, K1 >= 0, I1 is I-1, ks(I1,K1).
```

The first clause says that the empty set takes no space in the sack. The second clause covers the case in which the Ith item is *not* included in the sack. The third clause handles the case in which the Ith item *is* included in the sack.

This program could be exponential in the number of items, since it tries all subsets of items. However, there are only $I^2$ possible distinct calls to `ks/2`, so tabling will make this polynomial.

This program just finds whether a packing of the knapsack exists; it doesn't return the exact set of items that fit. We could simply add a third argument to this definition of `ks/2` which would be the list of items added to the knap-sack. But that might then build an exponential-sized table. For example with every item of size one, there are exponentially many items to include to make a sum. So instead of simply adding another parameter and tabling that predicate, we will use `ks/2` to avoid constructing a table unnecessarily. Note that this is similar to how we constructed a parse tree for a grammar by using the recognizer. Notice that `ksp/3` uses `ks/2` in its definition.

```
ksp(0,0,[]).
ksp(I,K,P) :- I>0,
    I1 is I-1, ks(I1,K),
    ksp(I1,K,P).
ksp(I,K,[I|P]) :- I>0,
    item(I,Ki), K1 is K-Ki, K1 >= 0, I1 is I-1, ks(I1,K1),
    ksp(I1,K1,P).
```

(There is something going on here. Can we figure out a syntax or conventions to make this uniform?)

```
% ks(+I,+K) if there is a subset of items 1,...,I that sums to K.
ks(0,0).
ks(I,K) :- ks1(I,K,_).
  ks1(I,K,I1) :- I>0,I1 is I-1, ks(I1,K).
ks(I,K) :- ks2(I,K,_).
  ks2(I,K,I1) :- I>0, item(I,Ki), K1 is K-Ki, K1 >= 0, I1 is I-1, ks(I1,K1).
```

```
ksp(0,0,[]).
ksp(I,K,P) :- ks1(I,K,I1), ksp(I1,K,P).
ksp(I,K,[I|P]) :- ks2(I,K,I1), ksp(I1,K1,P).
```

## 6.2 Sequence Comparisons

Another problem where dynamic programming is applicable is in the comparison of sequences. Given two sequences A and B, what is the miniml number of operations to turn A into B? The allowable operations are: insert a new symbol, delete a symbol, and replace a symbol. Each operation costs one unit.

A program to do this is:

```
/* sequence comparisons.  How to change one sequence into another.
A=a_1 a_2 ... a_n
B=b_1 b_2 b_3 ... b_m
Change A into B using 3 operations:
    insert, delete, replace: each operation costs 1.
*/

% c(N,M,C) if C is minimum cost of changing a_1...a_N into b_1...b_M
:- table c/3.
c(0,0,0).
c(0,M,M) :- M > 0..            % must insert M items
c(N,0,N) :- N > 0.             % must delete N items
c(N,M,C) :- N > 0, M > 0,
        N1 is N-1, M1 is M-1,
        c(N1,M,C1), C1a is C1+1,        % insert into A
        c(N,M1,C2), C2a is C2+1,        % delete from B
        c(N1,M1,C3),                    % replace
                a(N,A), b(M,B), (A==B -> C3a=C3; C3a is C3+1),
        min(C1a,C2a,Cm1), min(Cm1,C3a,C).       % take best of 3 ways

min(X,Y,Z) :- X =< Y -> Z=X ; Z=Y.

% example data
a(1,a). a(2,b). a(3,b). a(4,c). a(5,b). a(6,a). a(7,b).
b(1,b). b(2,a). b(3,b). b(4,b). b(5,a). b(6,b). b(7,b).
```

The first three clauses for `c/3` are clear; most of the work is done in the last clause. It reduces the problem to a smaller problem in three different ways, one for each of the operations of insert, delete, and replace. Each reduction costs one unit, except that "replacement" of a symbol by itself costs nothing. It then takes the minimum of the costs of these ways of turning string A into string B.

In Prolog this would be exponential. With tabling it is polynomial.

## 6.3 ??

Dynamic Programming, e.g. optimal association for matrix multiplication

Searching, games (see Bratko)

Pruning (alpha-beta search)

# Chapter 7

# HiLog Programming

XSB includes a capapbility to process programs which have complex terms in predicate of functor position. This allows programmers to program in a higher-order syntax, and so this extension of Prolog is called HiLog. Programmers can think of programming with parameterized predicates or with predicate variables. HiLog also supports a new way of programming with sets. We will explore these issues in this chapter.

HiLog is actually a very simple extension to Prolog. The definition of a basic term in Prolog is as follows:

- A term is a atomic symbol or a variable, or
- A term is of the form: $s(t_1, t_2, ..., t_n)$ where $s$ is an atomic symbol and the $t_i$ are terms.

Note that the symbol in functor (or predicate) position must be a symbol. HiLog generalizes this to allow an arbitrary term itself. So the definition of a term in HiLog is:

- A term is a atomic symbol or a variable, or
- A term is of the form: $t_0(t_1, t_2, ..., t_n)$ where the $t_i$ are terms.

Computationally these terms are matched just as Prolog terms, so intuitively HiLog programs work very similarly to Prolog programs. However, they encourage different ways of thinking about programs and support different programming paradigms.

## 7.1 Generic Programs

Because one can use a complex term as a predicate in HiLog, one can program "generic predicates." For example, consider a predicate function, i.e., a function that takes a predicate and returns another predicate. An interesting such predicate function might be `closure`. `closure` takes a binary predicate and returns a predicate for the transitive closure of the corresponding binary relation. So for example, `closure(parent)` would be the transitive closure of the `parent` relation, i.e., the ancestor relation, and `closure(child)` would be the descendent relation. We can define this `closure` predicate function in HiLog as follows:

```
closure(R)(X,Y) :- R(X,Y).
closure(R)(X,Y) :- R(X,Z), closure(R)(Z,Y).
```

Now given any binary relation, one can use use this definition to compute its closure. For example, we can define a binary predicate, `parent` as follows:

```
:- hilog parent.
parent(able,adam).
parent(able,eve).
parent(cain,adam).
parent(cain,eve).
etc
```

and then we can use the generic definition of closure to find anscestors:

```
| ?- closure(parent)(cain,X).
  etc.
```

Notice that we must declare the symbol `parent` to be a hilog symbol using the directive:

```
:- hilog parent.
```

This is necessary because the XSB system allows a mixture of HiLog programming and Prolog programming, and the system distinguishes between HiLog symbols and Prolog symbols in how it represents them. The HiLog term $t_0(t_1, t_2, ..., t_n)$ is represented as the Prolog term $apply(t_0, t_1, t_2, ..., t_n)$. Thus the system must know, for example, that `parent` is a hilog symbol so it knows to represent `parent(cain,adam) as the Prolog term \verb`apply(parent,cain,adam)—.

Another useful generic predicate is `map`. `map` takes a binary function and returns a function that when given a list, returns the list that results from applying that function to each element of the given list. Again, we can write a natural definition for it:

```
map(F)([],[]).
map(F)([X|Xs],[Y|Ys]) :- F(X,Y), map(F)(Xs,Ys).
```

So, for example, we can use this generic function to add one to every element of a list, double every element of a list, or square every element of a list. Given the definitions:

```
:- hilog successor,double,square.
successor(X,Y) :- Y is X+1.
double(X,Y) :- Y is X+X.
square(X,Y) :- Y is X*X.
```

we can do

```
| ?- [(hilog)].
[Compiling ./hilog]
% Specialising partially instantiated calls to apply/3
[hilog compiled, cpu time used: 0.59 seconds]
[hilog loaded]

yes
| ?- map(successor)([2,4,6,8,10],L).

L = [3,5,7,9,11];

no
| ?- map(double)([2,4,6,8,10],L).

L = [4,8,12,16,20];

no
| ?-
```

This definition of `map` is a bit more general than the one normally found in functional languages, which is not surprising since Prolog is a relational language and this is really a relational definition. For example, `map(successor)` is relation a relation over pairs of lists. If we give to `map` a nonfunctional relation, then the map of that relation is also nonfunctional.

(Think of an interesting example.)

Another interesting example is the generic function `twice`. `twice` takes an input function (or relation) and returns a function that applies the input function twice. (From DHDWarren and MVanEmden.) In standard mathematical notation: $twice(f)(x) = f(f(x))$. By turning `twice` into a relation and essentially writing down this definition, we get:

```
twice(F)(X,R) :- F(X,U), F(U,R).
```

And we can run it:

```
| ?- [twice].
[Compiling ./twice]
[twice compiled, cpu time used: 0.659 seconds]
[twice loaded]

yes
| ?- twice(successor)(1,X).

X = 3;

no
| ?- twice(twice(successor))(1,X).

X = 5;

no
| ?- twice(twice(square))(2,X).

X = 65536;

no
| ?- twice(twice(twice(double)))(1,X).

X = 256;

no
| ?-
```

This interesting thing here is that `twice(f)` for a function `f` produces a function similar to `f`, so we can apply `twice` to a result of `twice` and get a quadrupling (or octupling, ...) effect.

We can add another rule for `twice` (and make it a hilog symbol):

```
:- hilog twice.
twice(X,twice(X)).
```

This rule says that applying `twice` itself to a function argument gives a term representing the resulting function. So now we can even apply twice to itself to produce a function that we can then apply to one of basic functions to produce a function to apply to a number (that lives in the house that Jack built), as follows:

```
| ?- twice(twice)(double,Fun),Fun(1,X).

Fun = twice(twice(double))
```

```
X = 16;

no
| ?- twice(twice(twice))(double,Fun),Fun(1,X).

Fun = twice(twice(twice(twice(double))))
X = 65536;

no
| ?- twice(twice(twice))(successor,Fun),Fun(1,X).

Fun = twice(twice(twice(twice(successor))))
X = 17;

no
| ?-
```

DHDWarren (and a followup paper by Martin vanEmden et al.) explore issues around using Prolog to implement higher-order aspects of functional languages. This example is taken from there, but is expressed in HiLog's syntax, rather than Prolog's. HiLog's syntax makes the development more perspicuous.

(Do we(I) want to develop a bit more of lambda calculus, and show how to do more general higher-order programming?)

## 7.2 Object Centered Programming in XSB with HiLog

HiLog can also be used to program in an object-centered way.

Object oriented programming: C-logic and variants.

(Dealing with pragmatics of HiLog in XSB? and modules and recompilation? Interactions with tabling.)

# Chapter 8

# Debugging Tabled Programs

4-port Table Debugger.

The names may not be the best, but they should be clear.

1. Call
   - Call Untabled
   - Call Tabled: New Subgoal
   - Call Tabled: Use Incomplete Table
   - Call Tabled: Use Completed Table
2. Exit
   - Exit Untabled
   - Check/Insert Answer followed by
     - Redundant Answer - fail
     - Insert Answer – succeed
3. Redo
   - Retry Program Clause
   - Retry Answer Clause
4. Fail
   - Fail Untabled
   - Check Complete followed by
     - Completing tables (table list)
     - Rescheduling Answers for (table list)

How does tabling affect the debugger commands

Old commands :

Abort cleans up uncompleted tables.

Skip , Leap should work.

Break allows tables to be partially visible.

New Commands:

Ancestors. At least for tables.

Various Table examination mechanisms built upon Table builtins.

Show incomplete tabled subgoals.

Show returns for a table.

Show ancestors for each suspension of an incomplete tabled subgoal.

# Chapter 9

# Aggregation

In logic programming it is often the case that one wants to compare the various solutions to a single goal with each other. For example, we may have an employee relation that stores employee names, their departments and salaries, and we want to find, say, the total salary of all the employees. This requires querying the employee relation to retrieve the salaries and then combining all the solutions to find their sum.

Prolog provides several general predicates, the so-called all-solutions predicates, to allow a programmer to do such things. The all-solution predicates accumulate all the solutions to a particular goal into a list. The programmer can then use normal recursive programs to compute the desired function over that list, for example, sum of the elements.

To be concrete, to find the total of the salaries, a prolog programmer could write:

```
:- bagof(Sal,(Name,Dept)^employee(Name,Dept,Sal),Salaries),
   listsum(Salaries,MaxSal).
```

The predicate `listsum/2` simply takes a list of numbers and returns their sum. The all-solutions predicate `bagof/2` takes a template, a goal, and returns a list of instantiations of the template, one for each solution to the goal. In this case, the template is simply the variable `Sal`, so we will get back a list of salaries. The goal is:

```
(Name,Dept)^employee(Name,Dept,Sal)
```

The variables in the term before the ^ symbol indicate the *existential* variables in the goal. The values of `Sal` in successful solutions to the goal are accumulated into a single list, regardless of the values of the existential variables. In this case, we want all salary values, regardless of the employee's name or department. Another possibly interesting alternative query would be:

```
:- bagof(Sal,(Name)^employee(Name,Dept,Sal),Salaries),
```

```
    maximum(Salaries,TotalSals).
```

This query, without the variable `Dept` being existentially quantified, groups together solutions that have the same department, and returns nondeterministically, for each department, the list of salaries for employees in that department. So this is what one would get using a "group by" query in the database language SQL.

The `bagof/3` all-solutions predicate is used here because we don't want to eliminate duplicate salaries. If two employees have the same salary, we want to add both numbers; we want the sum of salaries, not the sum of *different* salaries.

One computational disadvantage of Prolog's all-solutions predicates is that regardless of the function to be computed over the bag (or set) of solutions, that list must still be created. To accumulate the sum of a set of numbers, it certainly seems inefficient to first construct a list of them and then add them up. Clearly one could just accumulate their sum as they are produced. In XSB if the goal is tabled, the situation is exacerbated, in that the set of solutions is in the table already; building another list of them seems a bit redundant.

XSB, being an extension of Prolog, supports its all-solutions predicates, but it also uses tabling to support several other such predicates, which are described in this chapter.

## 9.1 Min, Max, Sum, Count, Avg

In XSB we use HiLog and tables to support aggregation. In HiLog one can manipulate predicates, or the names of sets. So we can construct a set, or bag really, that contains all the salaries in our example of the simple employee relation:

```
:- hilog salaries.
salaries(Sal) :- employee(_Name,_Dept,Sal).
```

The symbol `salaries` is the name of a unary predicate that is true of all salaries, or rather is the name of a *bag* of all salaries. It is a bag since it may contain the same salary multiple times. XSB provides a predicate `bagSum` which can be used to sum up the elements in a named bag. So given the definition of the HiLog predicate `salaries/1` above, we can get the sum of all the salaries with:

```
:- bagSum(salaries,TotalSals).
```

The first argument to `bagSum` is the name of a bag, and the second is bound to the sum of the elements in the bag.

We can also do a "group by" to get total salaries within departments as follows. We define a parameterized predicate, `sals(Dept)`, to be the bag of salaries of employees in department `Dept`, as follows:

```
    sals(Dept)(Sal) :- employee(_Name,Dept,Sal).
```

This rule says that `Sal` is in the bag named `sals(Dept)` if there is an employee with some name who works in department `Dept` and has salary `Sal`.

Now with this definition, we can define a predicate, `deptPayroll/2`, that associates with each department the sum of all the salaries of employees in that department:

```
    deptPayroll(Dept,Payroll) :- bagSum(sals(Dept),Payroll).
```

XSB provides analogous aggregate operators, `bagMin/2`, `bagMax/2`, `bagCount/2`, `bagAvg/2`, to compute the minimum, maximum, count, and average, of a bag, respectively.

As an interesting example of aggregation and recursion, consider the following problem. Say our university is considering instituting a policy that guarantees that no supervisor shall make less than anyone he or she supervises. Since they do not want to lower anyone's salary, to initiate such a policy, they will have to give raises to supervisors who make less than one of their employees. And this may cascade up the chain of command. We want to write a program that will calculate how much they will have to spend initially to start this new program. The following program does this:

```
% inlcude needed predicates
    :- import bagMax/2, bagSum/2 from aggregs.
    :- hilog maximum, sum, raise.
    maximum(X,Y,Z) :- X>=Y -> Z=X; Z=Y.
    sum(X,Y,Z) :- Z is X+Y.

% The total cost is the sum of the raises.
    totcost(Cost) :- bagSum(raise,Cost).

% A raise is the max of the posible new salaries (own and
% subordinates' salaries) minus the old salary.
    raise(Raise) :-
        bagMax(possNewSal(Emp),NSal),
        emp(Emp,_,OSal),
        Raise is NSal-OSal.

% A possible new salary is either one's old salary or the max of the
% possible new salaries of one's immediate subordinates.
    possNewSal(Emp)(Sal) :- emp(Emp,_,Sal).
    possNewSal(Emp)(Sal) :-
        dept(Dept,Emp), emp(Sub,Dept,_),
        bagMax(possNewSal(Sub),Sal).
```

```
% dept(Dept,Mgr): department data
    dept(univ,provost).
    dept(ceas,deanCEAS).
    dept(cs,chairCS).
    dept(ee,chairEE).

% emp(Name,Dept,Salary):
    emp(provost,univ,87000).
    emp(deanCEAS,univ,91000).
    emp(chairCS,ceas,95000).
    emp(chairEE,ceas,93000).
    emp(prof1CS,cs,65000).
    emp(prof2CS,cs,97000).
    emp(prof1EE,ee,90000).
    emp(prof2EE,ee,94000).
```

Here is the execution of this program for this data:

```
| ?- [raises].
[Compiling ./raises]
% Specialising partially instantiated calls to apply/3
% Specialising partially instantiated calls to apply/2
[raises compiled, cpu time used: 1.669 seconds]
[raises loaded]

yes
| ?- totcost(C).

C = 19000;

no
| ?-
```

And indeed, it would cost $19,000 to upgrade everyone's salary appropriately.

We can combine aggregation with dynamic programming (which is actually what is happening to some extent in the previous example) to nice effect.

A variation on the knapsack problem discussed above in the section on dynamic programming uses aggregation to find an optimal knapsack packing. (This example taken most recently from JLP 12(4) 92, Clocksin, Logic Programming specification and exectuion of dynamic-programming problems. He refers to Sedgewick, Algorithms A-W 88.) Recall that in the earlier knapsack problem we were given a set of integers and we try to see if, given a target integer, we can choose a subset of the given integers that sum to the target integer. The version we consider here is a bit more complicated, and perhaps more realistic. Here we have a set of kinds of items; each item has a value and a size (both are integers.) We are given a knapsack of a given capacity and we want to pack the knapsack with the items that maximize the value of the total. We assume that there is an unlimited supply of items of each kind.

This problem can be formulated as follows:

```
:- import bagMax/2 from aggregs.
:- hilog maximum.
maximum(X,Y,Z) :- X>=Y -> Z=X; Z=Y.

:- table cap/2.
% cap(Size,Cap) if capacity of knapsick of size Size is Cap.
cap(I,Cap) :- I >= 0, bagMax(small_cap(I),Cap).

% small_cap(BigSize)(BigCap) if there is some item with ISize and IVal
% such that the capacity of a knapsack of size (BigSize-ISize) has
% capacity (BigCap-Ival).
small_cap(BigSize)(BigCap) :-
item(ISize,IVal),
SmallSize is BigSize-ISize,
cap(SmallSize,SmallCap),
BigCap is IVal+SmallCap.
% every knapsack (>=0) has capacity of 0.
small_cap(BigSize)(0) :- BigSize >= 0.
```

Here the tabling of `cap/2` is not necessary, since the call to `bagMax/2` is tabled automatically.

A simple example of executing this program is:

```
%Data:
item(10,18).
item(8,14).
item(6,10).
item(4,6).
item(2,2).

| ?- [aggreg].
[Compiling ./aggreg]
[aggreg compiled, cpu time used: 0.861 seconds]
[aggreg loaded]

yes
| ?- cap(48,C).
```

```
C = 86;

no
| ?- cap(49,C).

C = 86;

no
| ?-
```

And we can see that indeed to fill a knapsack of size 48, one should take four of item 10 (for total value 72), and one of item 8 (with value 14), giving us a total value of 86.

Another problem that combines dynamic programming and aggregation is the problem of finding the way to associate a matrix chain product to minimize the cost of computing the product.

```
:- import bagMin/2 from aggregs.
:- hilog minimum.
minimum(X,Y,Z) :- X=<Y -> Z=X; Z=Y.

% mult_costI,J,C) if C is the cost of the cheapest way to compute the
% product M_I x M_{I+1} x ... x M_J.
mult_cost(I,I,0).
mult_cost(I,J,C) :- I<J, bagMin(factor(I,J),C).

% factor(I,J) is true of costs obtained by computing the product of
% matrices between I and J by factoring the chain at any point between
% I and J and assuming optimal costs for the two factors.
factor(I,J)(C) :-
I1 is I-1,
J1 is J-1,
between(I,K,J1),
mult_cost(I,K,C1),
K1 is K+1,
mult_cost(K1,J,C2),
r(I1,Ri1), r(K,Rk), r(J,Rj),
C is C1+C2+Ri1*Rk*Rj.

between(X,X,_).
between(X,Y,Z) :- X < Z, X1 is X+1, between(X1,Y,Z).

% r(I,N) if N is the number of rows in the I-1st matrix.  (The last is
% the number of columns in the last matrix.)
r(0,5).
```

```
r(1,3).
r(2,6).
r(3,9).
r(4,7).
r(5,2).
```

## 9.2 BagReduce and BagPO

Actually, XSB provides two basic aggregation operators, `bagReduce/4` and `bagPO/3`, which are used to define all those predicates described in the previous section. We can also use the basic operators to define our own aggregation operators to do specialized things.

The `bagReduce/4` predicate takes a bag name, an operator and its identity, and composes the elements of the bag using the operator. For example, we can use `bagReduce/4` to define `bagSum/2`. First we define the sum operator, which must be a 3-ary HiLog operator:

```
:- hilog sum.
sum(X,Y,Z) :- Z is X + Y.
```

Then we can use this operator in `bagReduce/4` to define `bagSum/2`:

```
bagSum(Bag,Res) :- bagReduce(Bag,Res,sum,0).
```

The `bagReduce/4` predicate intuitively works as follows: It finds the first element of the bag, applies the operator to the identity and that element and stores the result in the table. It then finds the second element in the bag, applies the operator to the element in the table and that second element, and replaces the current element in the table by this new result. It continues this way: for each new element the operator is applied to the current value and the new element and the result replaces the current value. The final value in the table is returned as the result.

As another simple example, we can define `bagMin/2` by:

```
:- hilog minimum.
minimum(X,Y,Min) :- X =< Y -> Min = X ; Min = Y.

bagMin(Bag,Res) :- bagReduce(Bag,Res,minimum,1.0e+38).
```

(assuming that 1.0e+38 is the maximum representable number.)

A slightly more complicated example is the definition of `bagAvg/2`, which requires a more complex operator that must both sum and count the elements of the bag. It can be defined as follows:

```
:- hilog sumcount.
sumcount([S|C],X,[S1|C1]) :- S1 is S+X, C1 is C+1.

bagAvg(Bag,Avg) :-
    bagReduce(Bag,[Sum|Count],sumcount,[0|0]),
    Avg is Sum/Count.
```

The `bagPO/3` operator is also a metapredicate in that it takes a predicate as a parameter. It takes a HiLog binary predicate that defines a partial order. Given a bag, it returns nondeterministically all the maximal elements in the bag under the given partial order.

Consider the following example in which we try to find nice ways to explore a park while going from one point to another in it. Say the park has various interesting things to see and paths between them. We'll represent the paths in the park as a directed acyclic graph, with the points of interest as the nodes. (Both the acyclicity and the directedness of the graph might be somewhat unrealistic, but they can both be relaxed.) The goal now is, given a source and a destination node, find all "maximal" paths from the source to the destination. The idea is that we want to take a more-or-less direct route to our target, but we'd like to see as many points of interest as is reasonable along the way.

The following program will compute the set of such maximal paths.

```
    :- import bagPO/3 from aggregs.
    :- import member/2,append/3 from basics.

% stroll(X,Y,Path) if Path is a way to go from X to Y seeing many things.
    stroll(X,Y,Path) :- bagPO(walk(X,Y),BPath,subset), reverse(BPath,Path).

% subset(L1,L2) if L1 is a subset of L2.
    :- hilog subset.
    subset([],_L).
    subset([X|L1],L2) :- member(X,L2), subset(L1,L2).

% L is in walk(X,Y) if L is a (reversed) path from X to Y.
% (must be tabled because of left-recursion.)
    :- table walk(_,_)(_).
    walk(X,Y)([Y,X]) :- edge(X,Y).
    walk(X,Y)([Y|P]) :- walk(X,Z)(P), edge(Z,Y).
```

Here `walk(X,Y)` is a parameterized predicate name which represents the set of paths that go from node `X` to node `Y`. Each path is represented as a list of nodes (in reverse order of traversal.) The `bagPO` aggregation takes just the maximal paths, since we want the alternatives that allow us to see as many points of interest as possible. Here is the execution of the program on the data shown.

```
    edge(a,b).
    edge(b,c).
    edge(b,d).
    edge(c,d).
    edge(d,e).
    edge(a,f).
    edge(f,g).
    edge(g,e).

| ?- [aggreg].
[aggreg loaded]

yes
| ?- stroll(a,e,P).

P = [a,b,c,d,e];

P = [a,f,g,e];

no
| ?-
```

Some aggregate operations can be implemented using either `bagReduce/4` or `bagPO/3`. `bagMax/2` is a good example. Both of the following definitions are correct:

```
:- hilog maximum.
maximum(X,Y,Z) :- X >= Y -> Z = X ; Z = Y.
bagMax(Bag,Max) :- bagReduce(Bag,Max,maximum,-1.0e38).


:- hilog lt.
lt(X,Y) :- X < Y.
bagMax(Bag,Max) :- bagPO(Bag,Max,lt).
```

In such cases it is more efficient to use `BagReduce/4` because it can take advantage of the fact that at any point in time, there will be at most one value in the table.

## 9.3 Recursive Aggregation

Aggregation interacts in an interesting way with tabling. We've already seen that we can use them both: in the scenic-path example, we needed tabling to compute the paths using left recursion. We also saw an interesting recursive application of aggregation in the salary-raising example. We continue to explore the interaction of tabling, aggregation and recursion in this section by developing a program to compute the shortest paths between nodes in a (positive) weighted graph.

### 9.3.1 Shortest Path

To compute the shortest path between two points in a graph, we first define a HiLog predicate `short_path(Source,Target)`, that given two nodes returns short paths from the first to the second. There may be several short paths between two nodes, but we will be sure that one of them must be the shortest path:

```
% There's a short path if there's an edge,
short_path(X,Y)(D) :- edge(X,Y,D).
% or if there is a short path to a predecessor and then an edge.
short_path(X,Y)(D) :-
    bagMin(short_path(X,Z),D1),
    edge(Z,Y,D2),
    D is D1 + D2.
```

The first clause says that there is a short path from `X` to `Y` of length `D` if there is an edge from `X` to `Y` with weight `D`. The second clause says there is a short path from `X` to `Y` of length `D` if we take the minimum of the short paths from `X` to a predecessor (`Z`) of `Y` and we get `D` by adding the distance along the edge to `Y` from the predecessor.

Now to get the shortest path, we simply take the shortest of the short paths:

```
% The shortest path is the minimum of the short paths
shortest_path(X,Y,D) :- bagMin(short_path(X,Y),D).
```

This program in fact works for cyclic graphs, as long as all loops have nonnegative distance. To see why it works, we must look at it more closely. Normally we think of computing an aggregation by creating all the elements in the bag, and then performing the aggregation on the entire set. However, doing that here, with a cyclic graph, would result in a bag with infinitely many elements, since there are infinitely many different paths through a cyclic graph. It is clear that we can't construct and test every element in a bag of infinitely many elements. `bagMin` must return an answer before it has seen all the elements. Notice that if a graph has a self-loop, say from node `a` to node `a`, then a `D` such that `short_path(X,a)(D)` is defined in terms of the minimum of a bag that contains `D` itself. This turns out to be well-defined, because the minimum operator is monotonic. It works computationally because in the case of recursive definitions, the `bagMin` may return an answer before it has seen all the answers. At any point it returns the best one it has seen so far: if another one comes along that is better, it returns that one; if another comes along that is no better, it just ignores it, and fails back to find another.

So the order in which answers are generated can effect how much computation is done. If poor answers are returned first and many paths are computed using those poor answers, then all that work is unnecessary and will have to be done again with improved answers. Whereas if the best answer is returned first, then much less total computation will have to be done. So the complexity of this routine is dependent on the scheduling strategy of the underlying engine. We will look at these issues more later.

### 9.3.2 Reasoning with Uncertainty: Annotated Logic

We will look at examples including computing with annotated logic and Fitting's LP over bilattices.

```
:- import bagMin/2 from aggregs.
:- hilog minimum.
minimum(X,Y,Z) :- X =< Y -> Z=X ; Z=Y.

sumlist([],0).
sumlist([X|L],S) :- sumlist(L,S1), S is S1+X.

:- op(500,xfx,@).

G:D :- orFun(G,D).
orFun(G,D) :- bagMin(andFun(G),D).

andFun(G)(D) :- G@L,sumlist(L,D).

p(X,Y)@[D] :- edge(X,Y):D.
p(X,Y)@[D1,D2] :- p(X,Z):D1,edge(Z,Y):D2.

edge(a,b)@[5].
edge(b,d)@[6].
edge(b,c)@[1].
edge(c,e)@[3].
edge(e,d)@[1].
edge(a,c)@[7].
edge(c,d)@[2].
```

### 9.3.3 Longest Path

longest path.

```
% longest path (without loops)
```

```
:- import bagMax/2 from aggregs.
:- import member/2 from basics.

longpath(X,Y,P)(D) :- edge(X,Y,D), P=[F|R], \+member(F,R).
longpath(X,Y,P)(D) :-
bagMax(longpath(X,Z,[Z|P]),D), edge(Z,Y,D), Y\==Z, \+member(Y,P).

:- hilog maximum.  maximum(X,Y,Z) :- X @< Y -> Z=Y ; Z=X.

edge(a,b,5).
edge(a,b,6).
edge(a,d,4).
edge(d,b,5).
edge(b,c,3).
edge(b,c,4).
```

## 9.4 Scheduling Issues

Discuss breadth-first-like scheduling. Give example of graph that has exponential shortest-path with depth-first scheduling. Give benches on both depth-first and breadth-first scheduler. (For benches, maybe just give relative times, so as not to date it.)

(ts) I agree, Ive been using relative times.

## 9.5 Stratified Aggregation

Issues of stratified findall here, stratified aggregation?

# Chapter 10

# Negation in XSB

Negation in the context of logic programming has received a lot of attention. Prolog implements a kind of negation-as-failure inference rule, succeeding the negation of a goal if the goal itself cannot be successfully proven. This implements a kind of closed-world assumption, in that a proposition is assumed to be false if it cannot be proven to be true. This is a useful operator, which can be used to represent (and program) interesting situations.

Consider the standard example of defining the predicate `bachelor` using the predicates `married` and `male`:

```
bachelor(X) :- male(X), \+ married(X).

male(bill).
male(jim).

married(bill).
married(mary).
```

The rule says that an individual is a bachelor if it is male and it is not married. (`\+` is the negation-as-failure operator in Prolog.) The facts indicate who is married and who is male. We can interrogate this program with various queries and we get the following:

```
| ?- [negation].
[Compiling ./negation]
[negation compiled, cpu time used: 0.1 seconds]
[negation loaded]

yes
| ?- bachelor(bill).
```

```
no
| ?- bachelor(jim).

yes
| ?- bachelor(mary).

no
| ?- bachelor(X).

X = jim;

no
| ?-
```

as expected. The closed-world assumption is applied here: for example, when there is no fact that says that jim is married, we assume that he is not married. Also when there is no fact saying that mary is male, we assume she is not.

Before we get completely carried away with using negation, we need to look at situations in which there are problems. There are two sources of problems: floundering and nonstratification. Let's first consider a floundering query and program. Say we wrote the bachelor rule as:

```
bachelor(X) :- \+ married(X), male(X).
```

This looks to be an equivalent definition, since after all, the comma is conjunction so the order of literals in the body of a program (as simple as this one) shouldn't matter. But now consider the results of the same queries as above when they are submitted to this program:

```
| ?- bachelor(bill).

no
| ?- bachelor(jim).

yes
| ?- bachelor(mary).

no
| ?- bachelor(X).

no
| ?-
```

The answers are fine for the specific queries concerning bill, jim and mary, but the general query asking for all bachelors fails, whereas we would expect it to generate the answer jim. The reason is that the generated subquery of `\+ married(X)` is able to prove `married(X)` is true (for X=bill (and X=mary as well)), and so `\+ married(X)` fails. The problem is that the implementation of the operator `\+` only works when applied to a literal containing no variables, i.e., a ground literal. It is not able to generate bindings for variables, but only test whether subgoals succeed or fail. So to guarantee reasonable answers to queries to programs containing negation, the negation operator must be allowed to apply only to ground literals. If it is applied to a nonground literal, the program is said to *flounder*. Prolog systems in general allow the `\+` operator to be applied to nonground literals and so the programmer may get unexpected results. Often another operator, `not`, is provided which acts just like `\+` except that it gives an error message when applied to a nonground literal. (In XSB `not` and `\+` give the same, unsafe, results.)

The other problem that may arise in programs with negation, that of nonstratification, is a bit more subtle. !!!Instead of this example, use shave/2 and say the barber shaves everyone who doesn not shave himself. Who shaves the barber?

shave(john,john). shave(bill,bill). shave(barber,X) :- not shaves(X,X).

:- shaves(barber,barber).

!!! Say we want to define a predicate, `normal`, that is true of all reasonable sets. A set is normal if it doesn't contain itself as a member. (A set containing itself is rather wierd; think about it.) So we give the rule:

```
normal(S) :- \+ in(S,S).
```

where the predicate `in` denotes membership. Now we want to have the constant `n` denote the set of all normal sets. So `X` is in `n` just in case `X` is a normal set. The following rule reflects this:

```
in(X,n) :- normal(X).
```

Now consider what happens if we ask this program whether `n` is a normal set: `normal(n)`, which reduces to `\+ in(n,n)`, which reduces to `\+ normal(n)`. So to show that `n` is normal, we have to show that `n` is not normal. Clearly there is something a little odd here, and you may recognize a similarity to Russell's well-known paradox. The oddity is that the predicate `normal` is defined in terms of its own negation. Normally we consider rules to define predicates and this is an odd kind of cyclicity which we often want to avoid. Programs that avoid such cycles through negation in their definitions are called *stratified* programs. Notice that Prolog would go into an infinite loop whan asked queries that involve a cycle through negation.

XSB does not improve over Prolog in handling floundering queries; all calls to negative subgoals must be ground in XSB for the closed-world interpretation of negation to be computed. XSB does extend Prolog in allowing nonstratified programs to be evaluated, and we will discuss how it does that later in the chapter. However, first we will explore how we can use tabling with stratified programs to compute some interesting results.

## 10.1 Stratified Negation

As an example of stratified negation, consider the situation in which we have a set of terms and a nondeterministic reduction operation over them. Then given a term, we want to reduce it until further operations don't simplify it any more. We will allow there to be cycles in the reduction operation and assume that terms that reduce to each other are equivalently fully reduced.

This situation can be abstractly modeled by considering the terms to be nodes of a directed graph with an edge from N1 to N2 if the term at N1 directly reduces to the term at N2. Now consider the strongly connected components (SCCs) of this graph, i.e. two nodes are in the same SCC if each can be reached from the other. We will call an SCC a final SCC if the only nodes reachable from nodes in that SCC are others in that SCC. Now given a node, we want to find all nodes reachable from that node that are in final SCCs.

So first we define reachable:

```
:- table reachable/2.
reachable(X,Y) :- reduce(X,Y).
reachable(X,Y) :- reachable(X,Z), reduce(Z,Y).
```

Next we can define `reducible` to be true of nodes that can be further reduced, i.e., those nodes from which we can reach other nodes that cannot reach back:

```
reducible(X) :- reachable(X,Y), tnot(reachable(Y,X)).
```

`tnot` is the negation operator for tabled goals. It checks to see that the call doesn't flounder, giving an error message if it does. It can be applied only to a single goal, and that goal must be a tabled predicate. With this predicate we can next define the predicate `fullyReduce` that is true of pairs of nodes such that the first can be reduced to the second and the second is not further reducible:

```
:- table reducible/1.
fullyReduce(X,Y) :- reachable(X,Y),tnot(reducible(Y)).
```

Note that we must table `reducible` because `tnot` can only be applied to predicates that are tabled.

So with these definitions and the following graph for reduce:

```
reduce(a,b).
reduce(b,c).
reduce(c,d).
reduce(d,e).
reduce(e,c).
```

```
reduce(a,f).
reduce(f,g).
reduce(g,f).
reduce(g,k).
reduce(f,h).
reduce(h,i).
reduce(i,h).
```

we can ask queries such as:

```
| ?- fullyReduce(a,X).

X = c;

X = h;

X = d;

X = k;

X = i;

X = e;

no
| ?-
```

which returns all nodes in final SCCs reachable from node `a`.

However, we may now wish to generate just one representative from each final SCC, say the smallest. We can do that with the following program:

```
fullyReduceRep(X,Y) :- fullyReduce(X,Y), tnot(smallerequiv(Y)).

smallerequiv(X) :- reachable(X,Y), Y@<X, reachable(Y,X).
```

Now we get:

```
| ?- fullyReduceRep(a,X).

X = c;

X = h;
```

```
X = k;

no
| ?-
```

Note that this is an example of a stratified program. The predicate `reachable` is in the lowest stratum; then `reducible` is defined in terms of the negation of `reachable` so it is in the next stratum; then `fullyReduce` is defined in terms of the negation of `reducible`, so it is in the third stratum. `smallerequiv` is in the first stratum with `reachable`; and `fullyReduceRep` is in the same stratum as `fullyReduce`.

********************************

Issues of safety.

## 10.2 Approximate Reasoning

Use course prerequisites example. Introduce undefined truth value. Add undetermined facts for courses currently being taken. Then requests for whether have satisfied requirements will give: true if satisfied without any current course false if not satisfied even if all current courses are passed undetermined if satisfaction depends on outcome of current courses.

Categorization examples? Propositional Horn clauses for bird identification. Allow negation as failure, explicit negation.

```
cardinal :- red, crested.
bluejay :- blue, crested.
robin :- red_breasted, ~crested.

Use undef as a kind of null value?:
binarize relation
undef :- ~undef.
%emp(Name,Sal,Age) -> empsal/2 and empage/2
emp(david,50000,_|_) is represented as ``facts'':

empsal(david,50000).
empage(david,_X) :- undef.

or empage(david,X) :- between(45,X,55),undef.
(so can fail if not between 45 and 55.)
```

## 10.3 General Negation

Well-founded semantics of non-stratified negation.

We need examples! Planning? Use for-all type problems, e.g. to find if all nodes reachable from a given node are red, find if it is not the case that there exists a node reachable from the given node that is not red.

# Chapter 11

# Meta-Programming

## 11.1 Meta-Interpreters in XSB

Meta-interpreters. How one can write meta-interpreters and table them and get tabled evaluation. I.e. tabling "lifts" through meta-interpreters.

Could do XOLDTNF metainterpreter, exponential, but programmable. Probably want to use aggregation.

### 11.1.1 A Metainterpreter for Disjunctive Logic Programs

Do disjunctive LP metainterpreter

### 11.1.2 A Metainterpreter for Explicit Negation

Do explicit negation metainterpreter.

## 11.2 Abstract Interpretation

Abstract Interpretation examples. (partial evaluation, and assert.)

Show low overhead of tabling in meta-interpreter, due to how tables are implemented as tries.

### 11.2.1 AI of a Simple Nested Procedural Language

(see warren/xsb-tests/procabsint/*)

In this section we will see how to use XSB to construct a simple abstract interpreter for a procedural programming language. Such abstract interpreters can be used to do various kinds of data flow analyses. The abstract interpretator that we develop here is actually quite sophisticated; for example, it does interprocedural analysis. The interesting part here is how easy and straightforward it is to construct one with XSB, and therefore how we can be confident of its correctness.

The idea is to first construct a concrete interpreter for the object language. The Prolog programming language makes this particularly easy. We can run the concrete interpreter on various object programs and make sure that it is working reasonably well. After we have the concrete interpreter, we can easily change the operations to be abstract operations that operate over the abstract domain. Then we could execute programs over the abstract domain. This sounds easy (and it is in XSB) but a couple of issues arise. First, when computing over the abstract domain, the outcome of conditional tests cannot normally be determined. This means that what was a deterministic concrete program becomes a nondeterministic abstract program. Since we can't know which branch a specialization of the abstract program might take, we have to try them all. Now in XSB, that is not a problem, since XSB supports nondeterminism. Second, since we can't determine the exact outcome of conditionals, we don't know when to exit from loops. So a simple execution of most any abstract program would loop infinitely. What is really wanted is a least fixed point which will give reachable abstract states, and tabling in XSB gives exactly that. So XSB is ideally suited for this kind of abstract interpretaton problem.

Consider how this approach works in a specific case. The XSB (actually Prolog) program shown below is an interpreter for a simple procedural language that supports nested procedures, static scoping, and call-by-value parameter passing. It is far from trivial and we will discuss how each component works.

First we discuss how the execution environment is maintained. When a procedure is executing, it must have access to all variables that are visible to it. With each invocation of a procedure there is an activation record (AR) that stores its local variables. This is maintined in our interpreter as a simple list of (variable-name, variable-value) pairs. So when a procedure is executing, it will have access to its own AR that stores its local variables. But it must also have access to variables global to it, i.e., those in enclosing blocks. These are in ARs for the enclosing procedures. So the state for a procedure is kept as a list of ARs, the first being the procedure's own AR, the second being the AR of the immediately enclosing procedure, etc. We call such a list of AR's a Stack.

The following simple predicates get and set variable values in a Stack. They take a level number, indicating how global the variable is: 0 indicates local, 1 indicates in the immediately enclosing block, etc. They also take a Stack, and a variable name.

```
:- import append/3, memberchk/2 from basics.
```

```
% getVal(+N,+Stack,+Var,-Val)
getVal(0,[AR|_],Var,Val) :- memberchk((Var,Val),AR).
getVal(N,[_AR|Stack],Var,Val) :- N>0, N1 is N-1, getVal(N1,Stack,Var,Val).

% setVal(+N,+StackIn,+Var,+Val,-StackOut)
setVal(0,[AR|StackIn],Var,Val,[NAR|StackIn]) :-
repl_pair(AR,Var,Val,NAR).
setVal(N,[AR|StackIn],Var,Val,[AR|StackOut]) :-
N > 0, N1 is N-1,
setVal(N1,StackIn,Var,Val,StackOut).

repl_pair([(Var,_)|AR],Var,Val,[(Var,Val)|AR]) :- !.
repl_pair([P|AR],Var,Val,[P|NAR]) :- repl_pair(AR,Var,Val,NAR).
```

The interpreter takes as input the abstract syntax tree of an object program.

```
% evaluate a program

eval(module(_Name,Block)) :-
evalBlock(Block,[],0,[],_Stack).

evalBlock(block(Decls,Stmts),Pars,K,Stack0,Stack) :-
remFirst(K,Stack0,Stack1),
append(Pars,Decls,Locals),
evalStmts(Stmts,[Locals|Stack1],[_|Stack2]),
addFirst(K,Stack0,Stack2,Stack).

remFirst(0,L,L).
remFirst(N,[_|L0],L) :- N>0, N1 is N-1, remFirst(N1,L0,L).

addFirst(0,_,Stack2,Stack2).
addFirst(N,[AR|Stack0],Stack2,[AR|Stack]) :-
N>0, N1 is N-1,
addFirst(N1,Stack0,Stack2,Stack).

evalStmts([],Stack,Stack).
evalStmts([Stmt|Stmts],Stack0,Stack) :-
evalStmt(Stmt,Stack0,Stack1),
evalStmts(Stmts,Stack1,Stack).

evalStmt(assign(var(I,Name),Exp),Stack0,Stack) :-
evalExp(Exp,Stack0,Val),
setVal(I,Stack0,Name,Val,Stack).
evalStmt(while(Bool,Stmts),Stack0,Stack) :-
```

```
evalExp(Bool,Stack0,BVal),
(BVal =:= 0
 -> Stack = Stack0
 ; evalStmts(Stmts,Stack0,Stack1),
evalStmt(while(Bool,Stmts),Stack1,Stack)
).
evalStmt(if(Bool,Then,Else),Stack0,Stack) :-
evalExp(Bool,Stack0,BVal),
(BVal =\= 0
 -> evalStmts(Then,Stack0,Stack)
 ; evalStmts(Else,Stack0,Stack)
).
evalStmt(call(I,Name,ActPars),Stack0,Stack) :-
getVal(I,Stack0,Name,proc(Forms,Body)),
evalPars(ActPars,Forms,Stack0,ParLocals),
evalBlock(Body,ParLocals,I,Stack0,Stack).
evalStmt(print(Exps),Stack,Stack) :-
eval_print_exps(Exps,Stack).
evalStmt(dump,Stack,Stack) :-
writeln(Stack),nl.

evalPars([],[],_Stack,[]).
evalPars([Exp|Exps],[(Var,_)|Vars],Stack,[(Var,Val)|Decls]) :-
evalExp(Exp,Stack,Val),
evalPars(Exps,Vars,Stack,Decls).

eval_print_exps([],_).
eval_print_exps([Exp|Exps],Stack) :-
evalExp(Exp,Stack,Val),
writeln(Val),
eval_print_exps(Exps,Stack).

evalExp(int(V),_,V).
evalExp(var(I,Name),Stack,Val) :-
getVal(I,Stack,Name,Val).
evalExp(op(+,E1,E2),Stack,V) :-
evalExp(E1,Stack,V1),
evalExp(E2,Stack,V2),
V is V1+V2.
evalExp(op(*,E1,E2),Stack,V) :-
evalExp(E1,Stack,V1),
evalExp(E2,Stack,V2),
V is V1*V2.
evalExp(op(-,E1,E2),Stack,V) :-
evalExp(E1,Stack,V1),
```

```
evalExp(E2,Stack,V2),
V is V1-V2.
evalExp(op(<,E1,E2),Stack,V) :-
evalExp(E1,Stack,V1),
evalExp(E2,Stack,V2),
(V1<V2 -> V=1; V=0).
evalExp(op(>,E1,E2),Stack,V) :-
evalExp(E1,Stack,V1),
evalExp(E2,Stack,V2),
(V1>V2 -> V=1; V=0).
evalExp(op(=,E1,E2),Stack,V) :-
evalExp(E1,Stack,V1),
evalExp(E2,Stack,V2),
(V1=:=V2 -> V=1; V=0).


% compose stmt operations
evalStmts([],Stack,Stack).
evalStmts([Stmt|Stmts],Stack0,Stack) :-
    evalStmt(Stmt,Stack0,Stack1),
    evalStmts(Stmts,Stack1,Stack).

% extract envs for called block and exec body in that context
evalBlock(block(Decls,Stmts),Pars,Level,Stack0,Stack) :-
    keepTail(Level,Stack0,Stack1),
    append(Pars,Decls,Locals),
    evalStmts(Stmts,[Locals|Stack1],[_|Stack2]),
    replTail(Level,Stack0,Stack2,Stack).

% compute value of an expression in a context
evalExp(Exp,Stack,Val) :- ......

% evaluate a statement, generating new Stack
evalStmt(assign(var(I,Name),Exp),Stack0,Stack) :-
    evalExp(Exp,Stack0,Val), setVal(I,Stack0,Name,Val,Stack).
evalStmt(while(Bool,Stmts),Stack0,Stack) :-
    evalExp(Bool,Stack0,BVal),
    (BVal =:= 0 -> Stack = Stack0
                ;  evalStmts(Stmts,Stack0,Stack1),
                   evalStmt(while(Bool,Stmts),Stack1,Stack)).
evalStmt(if(Bool,Then,Else),Stack0,Stack) :-
    evalExp(Bool,Stack0,BVal),
    (BVal =\= 0 -> evalStmts(Then,Stack0,Stack)
                ;  evalStmts(Else,Stack0,Stack)).
evalStmt(call(I,Name,ActPars),Stack0,Stack) :-
```

```
getVal(I,Stack0,Name,proc(Forms,Body)),
evalPars(ActPars,Forms,Stack0,ParLocals),
evalBlock(Body,ParLocals,I,Stack0,Stack).
```

To obtain an abstract interpreter that does uninitialized variable analysis:

Underline evalExp call in assignment clause to point out that its definition is changed to implement abstract operations over the abstract domain of {uninitialized, hasValue}. Constant is mapped to hasValue. Binary ops return hasValue if both their operands are hasValue, otw uninitialized.

Add

```
:- table evalStmt/3.
```

at top.

Cross out the BVal =?= 0 conditions from the while and if-then-else clauses, to get regular disjunctions, instead of Prolog's if-then-else.

# Chapter 12

# XSB Modules

The XSB module system and how it works.

Does this deserve a chapter of its own, or should it be under System Facilities.

# Chapter 13

# Handling Large Fact Files

## 13.1 Compiling Fact Files

Certain applications of XSB require the use of large predicates defined exclusively by ground facts. These can be thought of as "database" relations. Predicates defined by a few hundreds of facts can simply be compiled and used like all other predicates. XSB, by default, indexes all compiled predicates on the first argument, using the main functor symbol. This means that a call to a predicate which is bound on the first argument will quickly select only those facts that match on that first argument. This entirely avoids looking at any clause that doesn't match. This can have a large effect on execution times. For example, assume that p(X,Y) is a predicate defined by facts and true of all pairs ¡X,Y¿ such that $1 <= X <= 20, 1 <= Y <= 20$. Assume it is compiled (using defaults). Then the goal:

```
| ?- p(1,X),p(X,Y).
```

will make 20 indexed lookups (for the second call to p/2). The goal

```
| ?- p(1,X),p(Y,X).
```

will, for each of the 20 values for X, backtrack through all 400 tuples to find the 20 that match. This is because p/2 by default is indexed on the first argument, and not the second. The first query is, in this case, about 5 times faster than the second, and this performance difference is entirely due to indexing.

XSB allows the user to declare that the index is to be constructed for some argument position other than the first. One can add to the program file an index declaration. For example:

```
:- index p/2-2.
```

```
p(1,1).
p(1,2).
p(1,3).
p(1,4).
...
```

When this file is compiled, the first line declares that the p/2 predicate should be compiled with its index on the second argument. Compiled data can be indexed on only one argument (unless a more sophisticated indexing strategy is chosen.)

## 13.2 Dynamically Loaded Fact Files

The above strategy of compiling fact-defined predicates works fine for relations that aren't too large. For predicates defined by thousands of facts, compilation becomes cumbersome (or impossible). Such predicates should be dynamically loaded. This means that the facts defining them are read from a file and asserted into XSB's program space. There are two advantages to dynamically loading a predicate: 1) handling of much larger files, and 2) more flexible indexing. Assume that the file qdata.P contains 10,000 facts defining a predicate q(X,Y), true for $1 <= X <= 100, 1 <= Y <= 100$. It could be loaded with the following command:

```
| ?- load_dyn(qdata).
```

XSB adds the ".P" suffix, and reads the file in, asserting all clauses found there. Asserted clauses are by default indexed on the first argument (just as compiled files are.)

Asserted clauses have more powerful indexing capabilities than do compiled clauses. One can ask for them to be indexed on any argument, just as compiled clauses. For dynamic clauses, one uses the executable predicate $index/3$. The first argument is the predicate to index; the second is the field argument on which to index, and the third is the size of hash table to use. For example,

```
| ?- index(q/2,2,10001).

yes
| ?- load_dyn(qdata).
[./qdata.P dynamically loaded, cpu time used: 22.869 seconds]

yes
| ?-
```

The index command set it so that the predicate $q/2$ would be indexed on the second argument, and would use a hash table of size 10,001. It's generally a good idea to use a hash table size that is an odd number that is near the expected size of the relation. Then the next command, the $load_dyn$, loads in the data file of 10,000 facts, and indexes them on the second argument.

It is also possible to put the *index* command in the file itself, so that it will be used when the file is dynamically loaded. For example, in this case the file woulstart with:

```
:- index(q/2,2,10001).

q(1,1).
q(1,2).
q(1,3).
...
```

Unlike compiled cclauses, asserted clauses can be indexed on more than one argument. To index on the second argument if it is bound on call, or on the first argument if the second is not bound and the first is, one can use the index command:

```
:- index(q/2,[2,1],10001).
```

This declares that two indexes should be build on $q/2$, and index on the second argument and an index on the first argument. If the first index listed cannot be used (since that argument in a call is not bound), then the next index will be used. Any (reasonable) number of indexes may be specified. (It should be noted that currently an idex takes 16 bytes per clause.)

Managing large extensional relations load_dyn, load_dync, cvt_canonical. Database interface, heterogeneous databases (defining views to merge DB's)

## 13.3 Indexing Static Program Clauses

For static (or compiled) user predicates, the compiler accepts a directive that performs a variant of $unificationfactoring$ [4].

....

## Bibliographic Notes

The idea of using program transformations as a general method to index program clauses was presented in a rough form by [7] [4] extented these ideas to factor unifications ...

# Chapter 14

# Table Builtins

table builtins: get_calls, get_returns, abolish_all_tables, ...

trie_assert, trie_retract (or maybe in section on large files, or maybe in a separate chapter on indexing.)

Do example to extract parses from a table created by recognition of a string in a CF grammar. (How to do, maybe interpreter.)

As examples, how about suspend / resume (which Rui has working at least partly) and the cursors / server example?

# Chapter 15

# XSB System Facilities

compiler options

Foreign code interface

Calling XSB from C

# Bibliography


[1] T. Chen, I.V. Ramakrishnan, and R. Ramesh. Multistage indexing algorithms for speeding Prolog execution. In *Proc. of the Int'l Conf. and Symp. on Logic Programming*, pages 639–653, 1992.

[2] W. Chen and D.S Warren. A goal-oriented approach to computing well founded semantics. In *Joint Intl. Conf. and Symp. on Logic Programming*, 1992.

[3] W. Chen and D.S. Warren. Towards effective evaluation of general logic programs. Technical report, SUNY at Stony Brook, 1993. A shortened version of this paper appeared in the *Proceedings of the 12th PODS*.

[4] S. Dawson, C. R. Ramakrishnan, I. V. Ramakrishnan, K. Sagonas, S. Skiena, T. Swift, and D. S. Warren. Unification factoring for efficient execution of logic programs. In *Proc. of the 22nd Symp. on Principles of Programming Languages*. ACM, 1995.

[5] S. Dawson, C.R. Ramakrishnan, I.V. Ramakrishnan, and T. Swift. Compilation aspects of optimizing tabled logic programs. Technical report, SUNY at Stony Brook, 1994.

[6] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[7] T. Hickey and S. Mudambi. Global compilation of Prolog. *Journal of Logic Programming*, 7:193–230, 1989.

[8] J. Naughton, R. Ramakrishnan, Y. Sagiv, and J. Ullman. Argument reduction through factoring. In *Proc. of the 15th Int'l Conf. on Very Large Data Bases*, pages 173–182. VLDB End., 1989.

[9] R. O'Keefe. *The Craft of Prolog*. MIT Press, 1990.

[10] L. Plumer. Termination proofs for logic programming based on predicate inequalities. In *Logic Programming: Proc. of the Seventh Int'l Conf.*, pages 634–648, 1990.

[11] I.V. Ramakrishnan P. Rao, K. Sagonas, T. Swift, and D.S. Warren. Efficient table access mechanisms for logic programs. In *International Conference on Logic Programming*, 1995. To Appear.

[12] R.C. Sekar, R. Ramesh, and I.V. Ramakrishnan. Adaptive pattern matching. *Siam J. Computing*, 1994. To Appear.

[13] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[14] J. Verschaetse and Danny De Schreye. Deriving termination proofs for logic programming using abstract procedures. In *Logic Programming: Proc. of the Eighth Int'l Conf.*, pages 301–315, 1991.