Worksheet 12

Prolog as a Theorem-Prover

The aims of this worksheet are that

- you will see that Prolog uses SLD resolution, where the search is implemented using a chronological backtracking algorithm (and is hence depth-first);
- you will learn that Prolog has a procedural semantics as well as a declarative semantics;
- you should see the consequences of Prolog's search algorithm on the semantics.

NOTE: A lot of the examples in this chapter are given for the propositional case, i.e. the predicates have no arguments. This is done simply to keep the examples small and tidy. Everything we say also applies in the case of predicates with arguments.

12.1 Horn Clauses as Prolog Programs

Positive Horn clauses are Prolog Rules and Facts

We begin by considering positive Horn clauses that contain some negative literals. For example:

$$\mathrm{p} \vee \neg \mathrm{q1} \vee \ldots \vee \neg \mathrm{qi}$$

This disjunction is, of course, equivalent to an implication:

$$(\mathrm{q1} \wedge \ldots \wedge \mathrm{qi}) \rightarrow \mathrm{p}$$

If we turn this round, we get:

$$p \leftarrow (q1 \wedge \ldots \wedge qi)$$

This now looks like a Prolog rule. We replace $\leftarrow$ by `:-`, use comma instead of $\wedge$, and end the rule with a full stop:

```
p :- q1, ..., qi.
```

Consider now positive Horn clauses that contain no negative literals, e.g.:

$$p$$

This straightforwardly corresponds to a Prolog fact:

```
p.
```

In summary, we have the following correspondences between positive Horn clauses and Prolog rules and facts:

Positive Horn clause	Prolog rule or fact
p	`p.`
$p \vee \neg q$	`p :- q.`
$p \vee \neg q \vee \neg r$	`p :- q, r.`
$p \vee \neg q \vee \neg r \vee \neg s$	`p :- q, r, s.`

Negative Horn Clauses are Prolog Queries

The following negative Horn clause

$$\neg p1 \vee \ldots \vee \neg pn$$

is equivalent to

$$\neg(p1 \wedge \ldots \wedge pn)$$

in which we see quite clearly that it is the negation of a conjunction.

We also know that Prolog queries can be conjunctions of literals. So effectively, when you issue the conjunctive query:

```
?- p1, ..., pn.
```

you are giving the Prolog system a goal, which you can think of as being something that the theorem-prover negates to give $\neg(p1 \wedge \ldots \wedge pn)$ and converts to clausal form, ending up as a negative Horn clause $\neg p1 \vee \ldots \vee \neg pn$.

12.2 Backtracking Search

Prolog is a resolution refutation theorem-prover using SLD-resolution on Horn clauses (specifically, one negative clause, the goal, and everything else positive, the program).

But Prolog does its search depth-first rather than breadth-first. Rather than searching the search space level by level, computing all SLD-resolvents at one level before going on to those at the next level, it searches path by path, exhaustively dealing with one path through the tree before moving on to the next path.

This has advantages and disadvantages.

The main advantage is that there is a way of implementing depth-first search that is very efficient (as these things go), namely to use a backtracking algorithm (see below). In particular this algorithm has the advantage that it might home in on a solution very quickly, while breadth-first search is a more plodding approach. Furthermore, to do breadth-first search requires the system to keep hold of all the resolvents at level i in order that it can compute all their descendants at level $i+1$; this requires a lot of space. Depth-first search *when implemented using backtracking* does not require so much space as we are really just exploring one path at a time. (There is some hidden need for space.)

The disadvantage has been alluded to before. Irrespective of how we search (breadth-first or depth-first), we know that some of the derivation graphs might be infinite. If a depth-first search gets onto one of these infinitely extensible paths, then you have an infinite computation (it will run until the system runs out of memory). Depth-first search is very susceptible to this. Breadth-first search, on the other hand, is a much 'fairer' strategy that gives everyone a shot before moving on to the next level. Of course, the search will still be infinite but the likelihood of getting caught on an infinite path and missing a solution because of this is avoided: if there is a solution at level i, it will be found before any path is extended to level $i+1$.

The consequence for Prolog programmers is that they have to think very carefully about the way they order parts of their programs. Certain orderings will avoid infinite searches, at least for the more common goals. We look at these complications later in this worksheet.

We know that in SLD-resolution

- one of the parents is the goal or a descendant of the goal;
- the other parent is a member of the initial axioms;
- the leftmost literals are resolved upon (and there is an ordering on all clauses).

By virtue of using depth-first search, additionally:

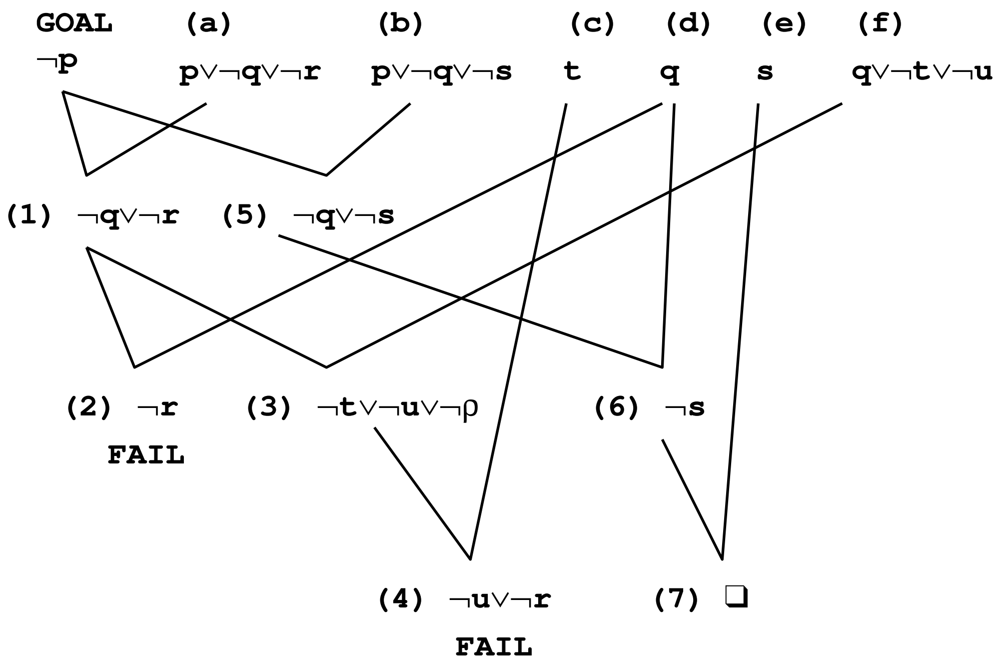
- not only is one parent the goal or one of its descendants, it is also the most recently produced unexplored descendant;

- not only is the other parent a member of the initial axioms, but these are regarded as ordered and they are tried in their given order.

Here's the algorithm for goal clause `G`:

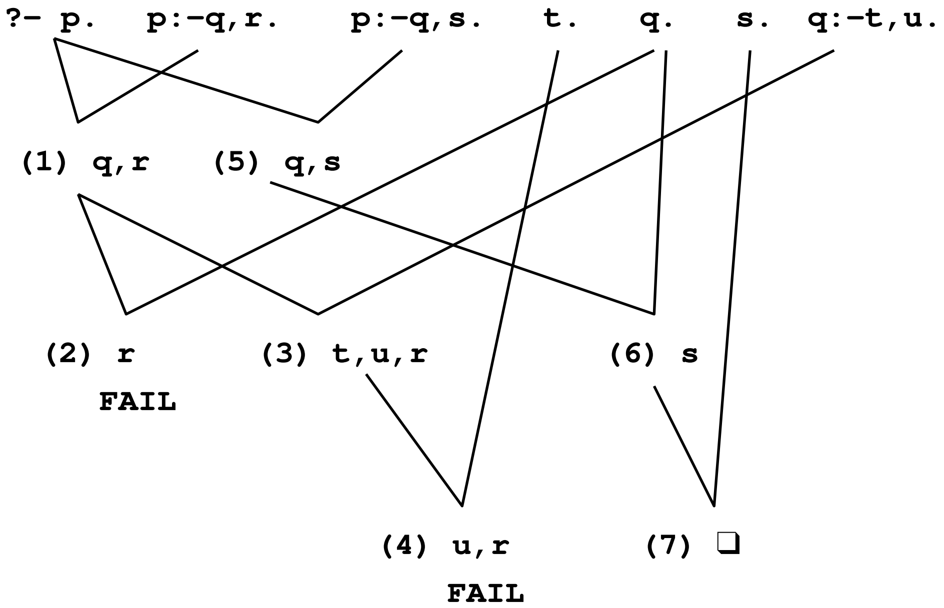
```
Backtrack-search(G)

for each clause C in the initial clauses
    taken in the order in which they appear do

   If G and C resolve on their leftmost literals,
      begin
      Create their resolvent
      If it is the empty clause then
         return answer yes and stop
      Else
         begin
         Call the resolvent G
         Backtrack-search(G)
         end
      end

end loop
```

Here's an example with the resolvents numbered so that you can see the order in which Prolog computes them.

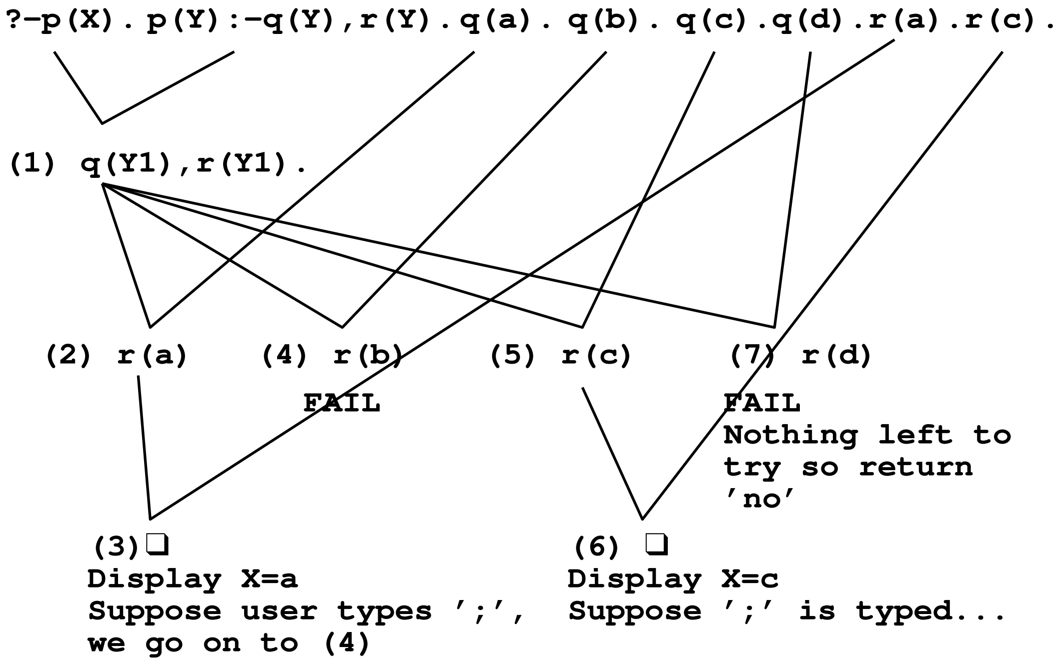


The goal is ¬p. The first clause that will resolve with the goal is (a). The resolvent is (1). Since this is the most recently produced descendant of the goal, it is one of the parents we use in the next step (if we can); this is achieved by the recursive call. So our goal is now ¬q ∨ ¬r (1). we look through the initial axioms from (a) to (f). The first one that can resolve with (1) is (d) to form (2). And (2) is the most recently produced descendant and so is our new goal. But as we look through the clauses from (a) to (f) we cannot find anything that SLD-resolves with (2). This path has failed and we must backtrack. We go back to the previously most recently produced descendant of the goal (1). Last time we resolved (1) with (d). We carry on looking through the initial axioms from (d) to (f) to see if there is anything else (1) can be resolved with. We find (f) and form (3). And so on!

Here's the same example in Prolog notation.



Here's another example in Prolog notation with some arguments to the predicates.



(These two examples show a distinction between *failure* and *absolute failure* in Prolog. Failure arises when the most recent descendant of the goal cannot be further resolved. Absolute failure is the case where the empty clause cannot be derived at all.)

Here is the trace Prolog gives for the second example:

```
?- trace.
yes

?- p(X).
    ( 1) 1 call:  p(_0) ?
    ( 2) 2 call:  q(_0) ? (1)
    ( 2) 2 exit:  q(a)
    ( 3) 2 call:  r(a) ? (2)
    ( 3) 2 exit:  r(a)
    ( 1) 1 exit:  p(a)

    X = a ; (3)

    ( 3) 2 back:  r(a) ?
    ( 3) 2 fail:  r(a)
    ( 2) 2 back:  q(_0) ?
    ( 2) 2 exit:  q(b)
    ( 4) 2 call:  r(b) ? (4)
    ( 4) 2 fail:  r(b)
    ( 2) 2 back:  q(_0) ?
    ( 2) 2 exit:  q(c)
    ( 5) 2 call:  r(c) ? (5)
    ( 5) 2 exit:  r(c)
    ( 1) 1 exit:  p(c)

    X = c ; (6)

    ( 5) 2 back:  r(c) ?
    ( 5) 2 fail:  r(c)
    ( 2) 2 back:  q(_0) ?
    ( 2) 2 exit:  q(d)
    ( 6) 2 call:  r(d) ? (7)
    ( 6) 2 fail:  r(d)
    ( 2) 2 back:  q(_0) ?
    ( 2) 2 fail:  q(_0)
    ( 1) 1 back:  p(_0) ?
    ( 1) 1 fail:  p(_0)

    no
```

I have numbered the lines of this (my numbers are at the right) to correspond with the resolvents on the diagram in the previous section.

Exercise 12.1 *Show the SLD-resolution derivation graph for the following clauses:*

> *Goal (already negated):* $\neg p(X)$
> *Initial axioms:* $p(Y) \vee \neg r(Y) \vee \neg q(Y)$, $q(a)$, $q(b)$, $r(c)$, $r(a)$

Number the clauses on your diagram to show the order in which they are computed by Prolog's depth-first search.

Convert the clauses into Prolog notation and redraw the diagram.

12.3 Procedural and Declarative Semantics

The conclusion we have reached by this point is that proof is a form of computation. Prolog in particular is both a programming language and a theorem-prover. As such, we get two perspectives on the semantics of a Prolog program, i.e. two ways of thinking about what such a program 'means'. These are the *declarative semantics* and the *procedural semantics*.

The declarative semantics is based on the standard model-theory of first-order logic. For example, the Prolog clause

```
p :- q, r.
```

has as its declarative semantics the same meaning as the clause

$$\mathrm{p} \vee \neg\mathrm{q} \vee \neg\mathrm{r}$$

From the point of view of declarative semantics, writing a logic program in Prolog is a question of axiomatising a relationship in logic (albeit using only Horn clauses). A clear example is axiomatising the ancestor relationship:

```
anc(A, D) :- par(A, D).
anc(A, D) :- par(A, Int), anc(Int, D).

par(a, b).
par(b, c).
```

The two Prolog rules are notational variants of:

$$\forall X_1 \forall X_2(\mathrm{par}(X_1, Y_1) \rightarrow \mathrm{anc}(X_1, Y_1)) \wedge$$
$$\forall X_2 \forall Y_2 \forall Z_2((\mathrm{par}(X_2, Z_2) \wedge \mathrm{anc}(Z_2, Y_2)) \rightarrow \mathrm{anc}(X_2, Y_2))$$

Only certain interpretations satisfy this program/set of clauses, as desired. Equally, certain wffs follow from the axiomatisation, e.g. anc(a, c).

The procedural semantics says *how* the output specified by the declarative semantics is obtained. We have seen that in the case of Prolog this will be a depth-first SLD-resolution search.

For the clause `p :- q, r`, the procedural semantics is: to solve problem `p`, *first* solve the subproblem `q` and *then* solve the subproblem `r`. (Or, in alternative words, to prove `p`, first prove `q` and then prove `r`.) This defines the *order* in which the goals are processed (this ordering being that which arises from using depth-first SLD-resolution).

Similarly, for the `anc` clauses above, we know that when trying to show that `anc(a, c)` follows from the clauses, Prolog first tries to prove `par(a, c)` (using the first rule), and this fails. It then tries to prove `par(a, Int)` followed by `anc(Int, c)` (by which time `Int` will have become bound to some value), and eventually this succeeds.

Discussing the order in which Prolog tries to prove things amounts to specifying Prolog's procedural semantics.

In principle, the declarative and procedural semantics ought to be the same. If this were so, since the results of the program are, in principle, determined by its declarative semantics, this should (in principle) be sufficient for writing programs. The programmer is encouraged to consider the declarative meaning of programs relatively independently of their procedural meaning. The procedural details are then left to the Prolog system. This is why we got so excited about logic programs in the first instance.

Unfortunately, however, the declarative approach is not always sufficient. Some aspects of Prolog force a procedural way of thinking upon us. In particular, in the case of Prolog, the procedural and declarative semantics are not always the same. We turn to these issues now.

12.4 Ordering

According to the declarative semantics, we can change

1. the order of clauses in a program, and
2. the order of goals in the bodies of clauses.

without affecting the declarative meaning (specifically, because of the commutativity of logical conjunction and disjunction). But, we already know that Prolog's procedural semantics *is* affected by these changes in, sometimes, very significant ways.

We are going to play with some variants of the `anc` program to show this. We will try all four possible orderings. The variants will all have the same declarative meaning. But they will not have the same procedural meaning. The goal we will consider is `?- anc(X, Y).`

`anc1` is the same as the original program.

```
anc1(A, D) :- par(A, D).
anc1(A, D) :- par(A, I), anc1(I, D).
```

In `anc2` the order of the clauses is reversed from the original.

```
anc2(A, D) :- par(A, I), anc2(I, D).
anc2(A, D) :- par(A, D).
```

In `anc3` the order of the clauses is the same as the original, but the order of the subgoals in the recursive clause is reversed.

```
anc3(A, D) :- par(A, D).
anc3(A, D) :- anc3(I, D), par(A, I).
```

In `anc4` both the order of clauses and the order of goals is changed.

```
anc4(A, D) :- anc4(I, D), par(A, I).
anc4(A, D) :- par(A, D).
```

As a quick overview, here are the results of running these programs:

```
?- anc1(X, Y).

Y = b
X = a ;

Y = c
X = b ;

Y = c
X = a ;

no

?- anc2(X, Y).

Y = c
X = a ;
```

```
Y = b
X = a ;

Y = c
X = b ;

no

?- anc3(X, Y).

Y = b
X = a ;

Y = c
X = b ;

Y = c
X = a ;

!  Out of local stack during execution.
% execution aborted

?- anc4(X, Y).

!  Out of local stack during execution.
% execution aborted
```

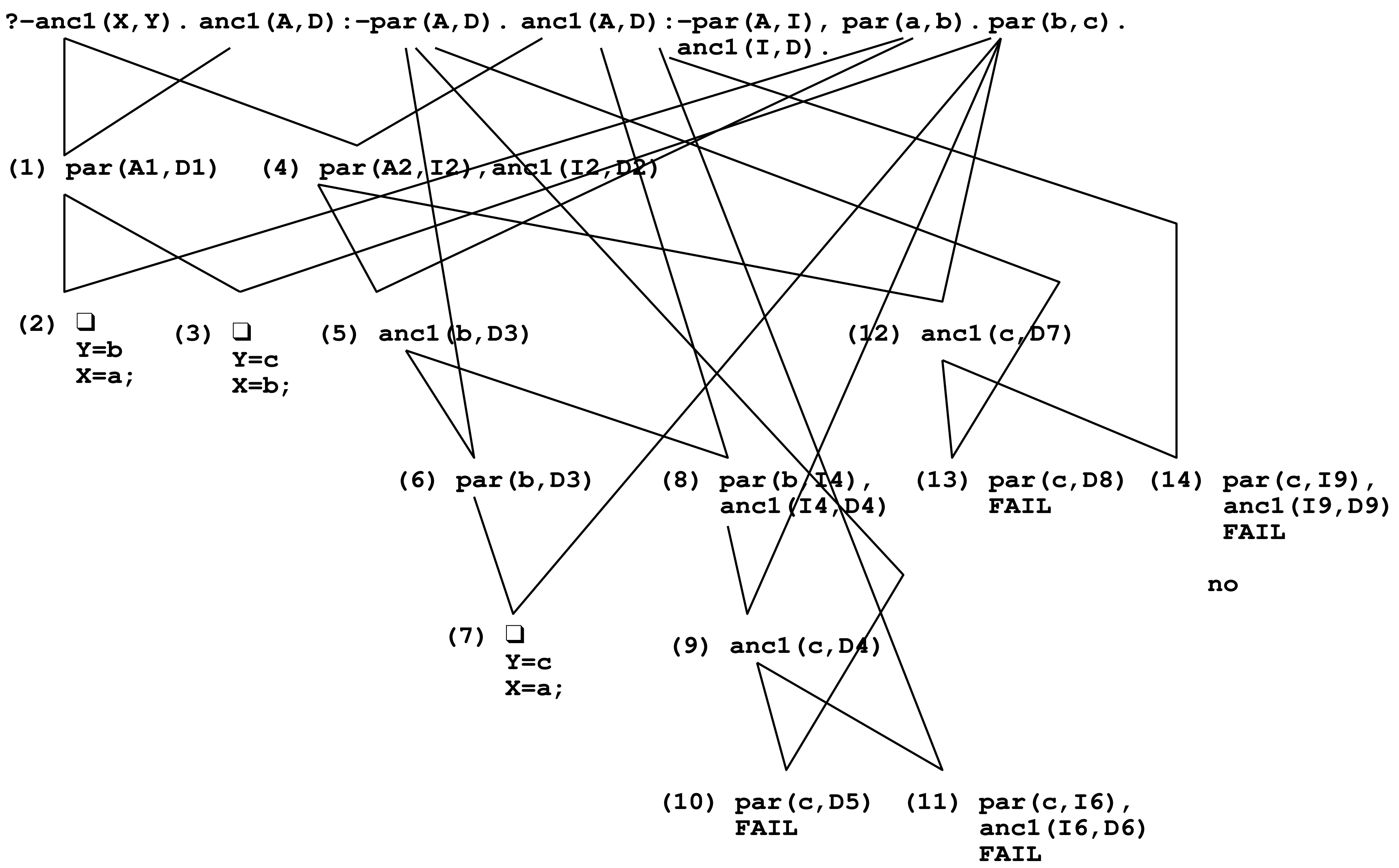
`anc1` finds all the answers and terminates. `anc2` finds them all but in a different order from `anc1`. `anc3` finds them all in the same order as `anc1` but does not terminate. `anc4` finds none of the answers and fails to terminate. These results are perhaps astonishing given that all four programs have the same declarative semantics.

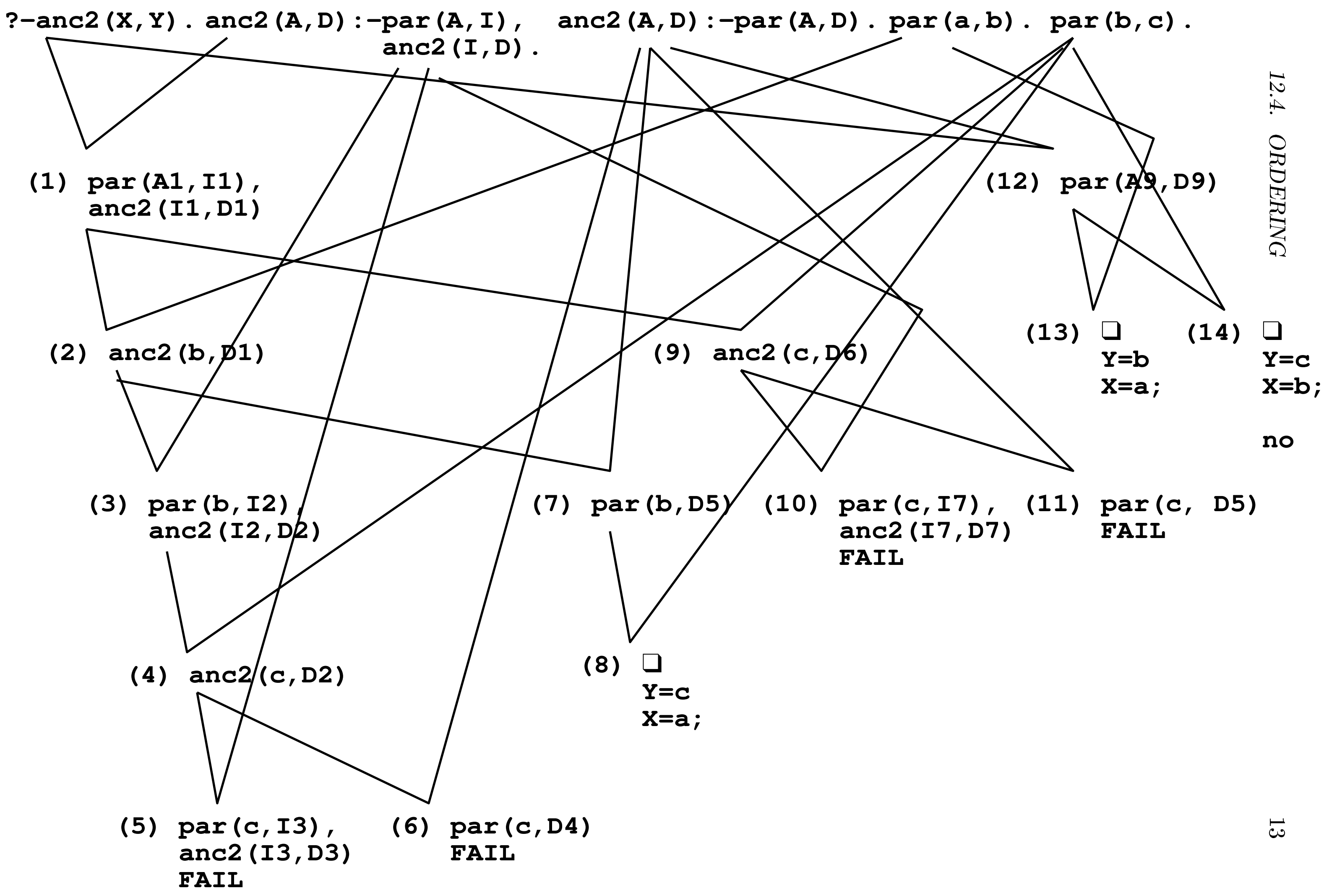
We can only understand these variations by thinking about the procedural semantics. We try to show this using a derivation graph in which nodes are numbered to indicate the order in which they are computed.

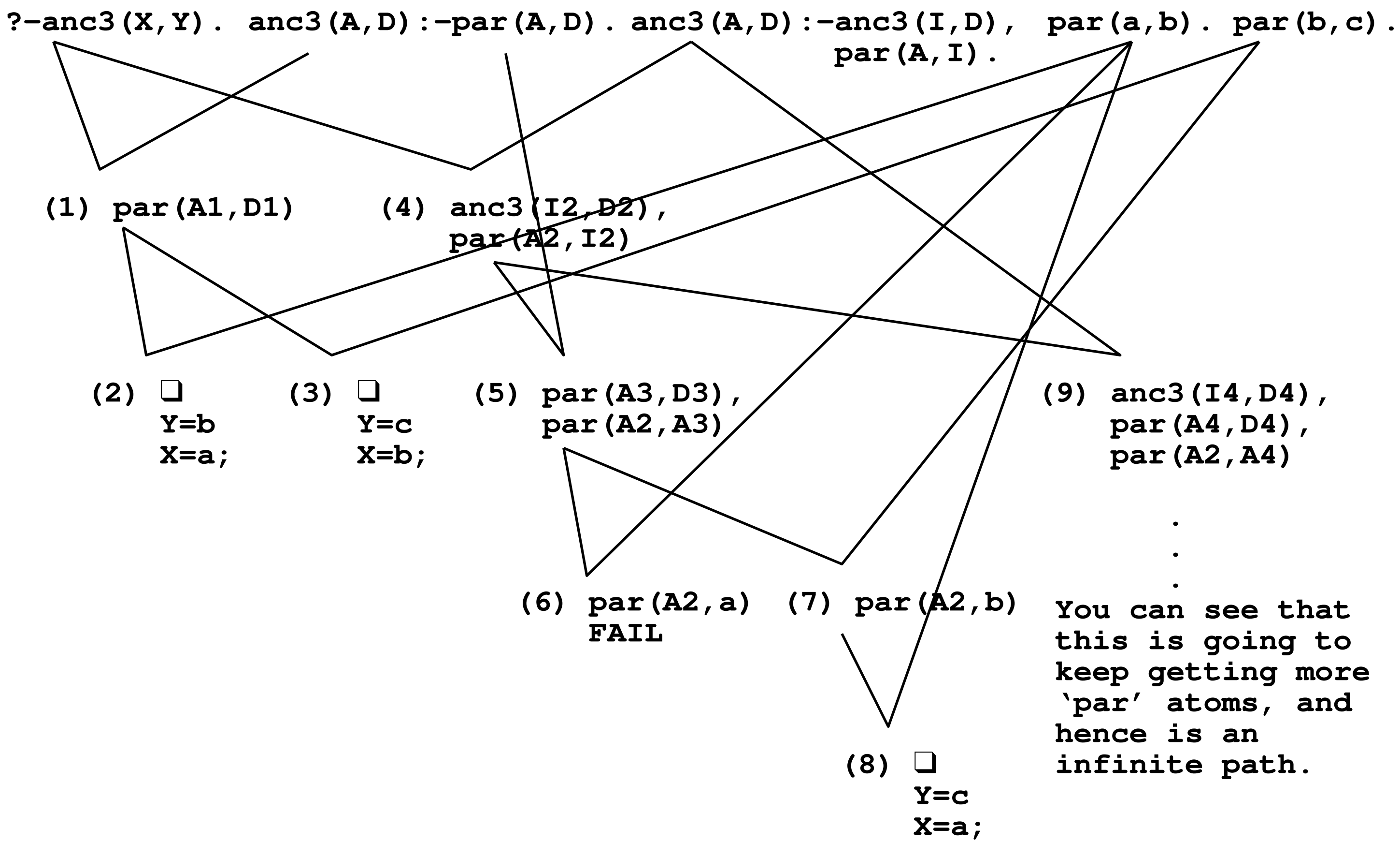
We now give four derivation graphs, one per definition.

As you can see by looking at the graphs for `anc1` and `anc2`, changing the order of clauses permutes the branches in the search tree. As the tree is searched depth-first, permuting the branches causes a different order of traversal of the derivation graph, and so a different order of finding the solutions. It's a bit as if in `anc1` Prolog outputs solutions as it goes along, while in `anc2` it travels to the end of the family tree and gives solutions on the way back.

Because the clause order in `anc3` is the same as that of `anc1`, solutions are produced as we go along. But changing the subgoal order gives a different


?-anc1(X,Y). anc1(A,D):-par(A,D). anc1(A,D):-par(A,I), par(a,b). par(b,c).
anc1(I,D).
(1) par(A1,D1)
(4) par(A2,I2),anc1(I2,D2)
(2) ❑
Y=b
X=a;
(3) ❑
Y=c
X=b;
(5) anc1(b,D3)
(12) anc1(c,D7)
(6) par(b,D3)
(8) par(b,I4),
anc1(I4,D4)
(13) par(c,D8)
FAIL
(14) par(c,I9),
anc1(I9,D9)
FAIL
no
(7) ❑
Y=c
X=a;
(9) anc1(c,D4)
(10) par(c,D5)
FAIL
(11) par(c,I6),
anc1(I6,D6)
FAIL


?-anc2(X,Y).
anc2(A,D):-par(A,I),
anc2(I,D).
anc2(A,D):-par(A,D).
par(a,b).
par(b,c).
(1) par(A1,I1),
anc2(I1,D1)
(2) anc2(b,D1)
(3) par(b,I2),
anc2(I2,D2)
(4) anc2(c,D2)
(5) par(c,I3),
anc2(I3,D3)
FAIL
(6) par(c,D4)
FAIL
(7) par(b,D5)
(8) ❑
Y=c
X=a;
(9) anc2(c,D6)
(10) par(c,I7),
anc2(I7,D7)
FAIL
(11) par(c, D5)
FAIL
(12) par(A9,D9)
(13) ❑
Y=b
X=a;
(14) ❑
Y=c
X=b;
no


?-anc3(X,Y).
anc3(A,D):-par(A,D).
anc3(A,D):-anc3(I,D), par(A,I).
par(a,b).
par(b,c).
(1) par(A1,D1)
(4) anc3(I2,D2), par(A2,I2)
(2) ❑ Y=b X=a;
(3) ❑ Y=c X=b;
(5) par(A3,D3), par(A2,A3)
(9) anc3(I4,D4), par(A4,D4), par(A2,A4)
(6) par(A2,a) FAIL
(7) par(A2,b)
(8) ❑ Y=c X=a;
. . .
You can see that this is going to keep getting more 'par' atoms, and hence is an infinite path.

```
?-anc4(X,Y).  anc4(A,D):-anc4(I,D),  anc4(A,D):-par(A,D).  par(a,b).  par(b,c).
                         par(A,I).

(1) anc4(I1,D1),
    par(A1,D1)

  (2) anc4(I2,D2),
      par(A2,D2),
      par(A1,D2)

      (3) anc4(I3,D3), par(A3,D3)
          par(A2,D3),par(A1,D3)

      ...................................and so on
```

graph (after all SLD-resolution is all about resolving on leftmost literals, so if you change goal order then you change what's leftmost). Specifically in this case, it gives an infinite branch.

`anc4`, even more obviously, has an infinite branch, and it is the first branch searched by the depth-first SLD-resolution algorithm: the infinite branch is due to the subgoal order in the recursive clause, but unlike `anc3` we don't get any solutions at all because the change of clause order has made the infinite branch the first one we tackle.

□ Type all four definitions into a file. Trace the execution of the query on each definition. Try them out with different goals (i.e. goals with different instantiation patterns).

12.5 Discussion

12.5.1 Clause order

We have found that the order of clauses is largely irrelevant: it merely affects the order in which solutions are found. Only sometimes is the order of solutions important to you (you usually think of them as a set, in which case order doesn't matter). But there is a danger of putting an infinite branch first. So my preference is generally to put the base clause of recursive definitions first. (When we move from pure Prolog to full Prolog by adding some funnies to pure Prolog, the order of clauses does become even more important.)

Of course, making matters worse is that, when Prolog programmers know something about the procedural semantics, they start to exploit it. In this case, programmers know that clauses are tried in the order in which they appear in the Prolog database, and so some programmers employ a programming trick known as a *failure-driven loop*.

A common use of failure-driven loops is when the body of the loop uses some impure Prolog that has some side-effect (e.g. that reads from or writes to the screen, or that asserts clauses into the Prolog database or retracts clauses from the Prolog database). Here's an example which deletes all `parent` clauses from the database. It uses the primitive `retract`, which has the effect of deleting a clause. But this is done repeatedly using a failure-driven loop until none is left:

```
kill_parent :- retract(parent(X, Y)),
               fail.

kill_parent.
```

When you issue a query on `kill_parent`, you always get back just a *yes* answer:

```
?- kill_parent.
yes
```

But it's the side-effects that count. What happens if the database contains the `parent` facts:

```
parent(liz, chas).
parent(chas, willie).
```

The goal `kill_parent` matches the first clause defining `kill_parent`, so the rule's right-hand side become a set of subgoals to be proved. `retract(parent(X, Y))` is solved with `X` $\mapsto$ `liz` and `Y` $\mapsto$ `chas`, and this clause, `parent(liz, chas)` is deleted from the database. Then Prolog encounters the `fail`. Now, `fail` is a predicate that is built-into many Prologs. It is a goal that always fails, i.e. it always returns `no`. (You can see that we don't really need it to be built-in. All we need to do is to use a predicate here that has no definition elsewhere in the database, as that predicate will clearly always fail.) Failure causes backtracking. On backtracking, Prolog finds another way of solving `retract(parent(X, Y))`, this time with `X` $\mapsto$ `chas` and `Y` $\mapsto$ `willie`, and so the clause `parent(chas, willie)` is removed from the database. The `fail` is reached again and so Prolog backtracks again.

Indeed Prolog will keep backtracking in this way for as long as there are `parent` clauses in the database. Once it exhausts these clauses, the system backtracks further and looks for a different way of proving `kill_parent`. There is another way, the second clause headed `kill_parent`, which succeeds trivially and so we stop.

The disadvantage of this programming 'trick' is clear: the program can no longer be understood purely declaratively. The way it works is intimately tied in with Prolog's search strategy. if someone were to give us an interpreter with a different search rule (e.g. working from the bottom of the database to its top, or working breadth-first or in parallel), then, when programs that use this trick are run on the new interpreters, they would not produce the results the programmer intended.

On balance, therefore, this way of implementing iterative behaviour is to be discouraged. (That doesn't stop me using it in worksheet 16!) At least two alternatives exist: (a) there is another way of implementing failure-driven loops, which uses Prolog negation (worksheet 13), and (b) there are some looping constructs built-in to Prolog, so-called all-solutions predicates (worksheet 13 also).

12.5.2 Goal order

The order of the subgoals, on the other hand, is critical as it can lead to infinite computations. Before we discuss infinite computations though, we bring out another reason why it is important.

Changing the order of the subgoals affects the amount of search that needs to be done, and therefore has an affect on efficiency. (Efficiency considerations, of course, are another aspect of the procedural semantics; they have nothing to do with the declarative semantics.) Generally you will want to put the most restrictive goal first. That way you are less likely to have to evaluate the second goal. For example, for the following facts:

```
q(a).
q(b).
r(i).
```

The rule

```
p1(X, Y) :- q(X), r(Y).
```

is (marginally) less efficient that the rule

```
p2(X, Y) :- r(Y), q(X).
```

□ Put them in a file, load them into Prolog, trace them and compare the traces if you don't believe me!

Actually this idea of putting the more restrictive subgoal first is a bit simplistic. What if you have a definition of `grandparent`, that uses two calls to `parent`? Which ordering do you use?

```
grandparent1(X, Z) :- parent(X, Y), parent(Y, Z).
grandparent2(X, Z) :- parent(Y, Z), parent(X, Y).
```

In one sense, one call to `parent` cannot be more restrictive than the other so it would appear not to matter. But in fact it all depends on your most likely queries. The two rules have the same efficiency on queries such as `?- grandparent(X, Y)`. But the first definition is better than the second on queries to try to find a person's grandchild, e.g. `?- grandparent1(alf, Y)`; and the second is better to try to find a person's grandparent, e.g. `?- grandparent2(X, alf)`.

Exercise 12.2 *Why?*

But this efficiency stuff is a digression. Our main concern about goal order is the way that it leads to a discrepancy between the declarative and procedural semantics: a program with correct declarative semantics may not work (e.g. `anc4`). Can we characterise when this problem will arise?

The problem with both `anc3` and `anc4`, which leads to the infinite computations, is that both contain *left-recursive* rules. A rule is left-recursive if the first goal in the body is the recursive call. An extreme example is

```
p :- p.
```

If you issue the query `?- p.`, the computation will be infinite. (There is nothing wrong with its declarative semantics though, i.e. $p \rightarrow p$, which is a tautology.)

The best solution to the problem of left recursion is avoidance. In particular, where there are multiple subgoals in the body, we should not put the recursive call first. In the case of the ancestor program this is easy: we avoid `anc3` and `anc4` and use `anc1` or `anc2` instead.

There are harder cases, however. For example, suppose we have the following facts:

```
wed(alf, mary).
wed(bill, nora).
wed(carl, olive).
```

and we want to ensure that the definition is commutative, i.e. we want `?- wed(alf, mary)` and `?- wed(mary, alf)` to both be true. If we add the rule

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

we have a left-recursive rule and infinite computations. The way round this is not to use the rule above but instead is to define a new predicate, as follows:

```
married(X, Y) :- wed(X, Y).
married(X, Y) :- wed(Y, X).
```

`married` gives us a commutative relationship.

Unfortunately, there are some programs that simply will be left-recursive, and there's nothing we can do about it. (Many of the recursive programs we developed in the worksheet using lists contained only one goal in the body and so no re-ordering was possible.) These tend to be programs that, by their nature, have infinite answers (i.e. declaratively, they axiomatise an infinite relation). However, sometimes these programs will still work, provided one or more of the arguments in a query is instantiated and the base case comes first in the program. (We also saw the complication that when program A calls program B, where program B is one of these programs that only works for certain instantiations, then we have to make a decision about typical calls to program A: although making program A left-recursive means that it will only work for certain (hopefully, typical) patterns of instantiation, the consequences of making it non-left-recursive would be that typical calls to A would result in calls to program B with an instantiation pattern for which B launches an infinite computation. `reverse` was one such program.)

12.5.3 Summary

Is all of this good or bad? It seems that much of the promise of declarative programming has gone down the drain with Prolog's control strategy. The question is whether this is just a facet of Prolog, and can be remedied by coming up with a language with which to replace Prolog, or whether it is intrinsic to logic programming. Well..., it is not intrinsic to logic programming because we can make these problems go away simply by having a language that is implemented with a completely different control strategy.

This raises the question of why Prolog didn't get given a different control strategy in the first place. The answer is that the depth-first SLD-resolution strategy, while it has its disadvantage with left-recursive rules, is otherwise extremely efficient: it can home in on solutions very quickly. So we have gone for speed. A compromise has been made. We have sacrificed completeness and soundness in the quest for efficiency. (We have sacrificed completeness because entailments of a program may not be proved to follow from the program by Prolog's control strategy, as has been discussed above. It is unsound because the unification algorithm used in resolution lacks the occurs check (as was briefly discussed in worksheet 8). To illustrate this last point, put the following fact into Prolog: `loves(X, person_loved_by(X))`. Try the goal `?-loves(Y, Y)`, i.e. does anyone love him/herself. This goal kind-of succeeds in Prolog. The answer is *`Y = person_of(person_of(person_of...`*. An infinite term (whose display on the screen causes Prolog to run out of space, nevertheless Prolog's answer is still basically 'yes'). But 'yes' is the wrong answer. `loves(Y, Y)` is not a logical consequence of the program. Try normal resolution refutation theorem-proving on the example, and you will find that unification with the occurs check fails, so there is no resolution to the empty clause. Since Prolog lacks the occurs check, this causes it to produce a 'wrong' answer: it is not sound, but the Prolog unification algorithm is faster than it otherwise would be.)

To conclude then,

- At least Prolog is on the road towards being declarative. Some declarativeness is better than none. You shouldn't criticise declarative programming or logic programming just because of the shortcomings of Prolog. Maybe a better language will emerge from the experience of using Prolog.

- You can still separate your Prolog programming into a declarative phase, when you write a program which is declaratively correct, and then a procedural phase, when you swap orderings around to handle Prolog's control strategy. Arguably the fact that there is a declarative phase to your thinking is more likely to lead to you writing a correct program.

Acknowledgements

Bratko, Flach and Sterling & Shapiro provided a little of the material in this worksheet.