CAP 5602
Summer, 2011

Lesson 3: Goals and Rules

The topics

1. matching goals and rules
2. the path program

3: the path2 program

1. Matching and Rules

In Lesson 2 we learned how prolog handles goals when the clauses are facts. But what happens when we have rules?

At any time, prolog has a stack of goals, $G_1, G_2, \ldots, G_n$ that needs to be satisfied. In that lesson we learned that it tries to satisfy G_1 first, by trying to match it with the clauses in the database, from the first down to the last. Let's assume that the clause is the rule below.

$H : -B_1, \ldots, B_m$

Prolog first relabels the variables of the rule, such that the rule and the stack have no variables in common. So, we get the rule

$H' : -B'_1, \ldots, B'_m$.

The interpreter then tries to match G_1 and H'. If it succeeds, it puts $B'_1, \ldots, B'_m$ on the top of the stack, so, the stack of goals is now $B'_1, \ldots, B'_m, G_2, , \ldots, G_n$. If it fails, prolog tries to match G_1 with the clause that follows the rule.

Let us trace the program prog3, displayed beneath.

```
% define the path based on the arc predicate.
path(X,X).        % clause 1
path(X,Y) :- arc(X,Z), path(Z,Y).        % clause 2

% the arc clauses
arc(a,b).         % clause 3
arc(b,c).         % clause 4
arc(a,d).         % clause 5
arc(c,e).         % clause 6
```

The path is defined recursively: there is a path from a node to itself (the empty path) and there is a path from X to Y, if there is an arc from X to

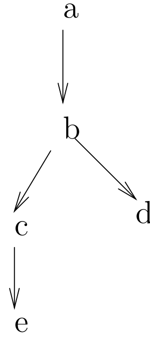

Figure 1: The graph for prog3

some node Z, and a path from Z to Y. The arc clauses describe the graph from Figure 1.

We compile this file and then we ask if there is a path from a to e. The trace of the program is shown in Figure 2

The original goal is path(a,e). Prolog tries to match it with clause 1 and fails because X cannot be matched with both a and e. So, the interpreter tries to match it with the second clause. This clause is a rule, and there is no need to relabel the clauses of the rule because the goal has no variables. The goal can be matched with the head of the rule and we get the substitution { X = a, Y = b }. we apply this substitution to the body and we put the body at the top of the stack of goals, getting arc(a,Z), path(Z,e). Prolog tries to satisfy the first goal. It does not match the first 2 clauses because both the first fact and the head of the rule have the functor path. It succeeds with fact 3 and we get the substitution { Z = b }. The stack of goals is now path(b,e). Again, the goal cannot be matched with clause 1, but it succeeds with clause 2, the matching substitution being { X=b, Y= e} and the list of goals is now arc(b,Z),path(c,e).

The goal arc(b,Z) is satisfied by clause 4 with the substitution { Z= c } and the list of goals becomes path(c,e). This goal is satisfied by clause 2 with the substitution { X=c,Y=e} and the stack of goals is arc(c,Z),path(Z,e). The goal arc(c,z) is satisfied by clause 6, with the substitution { Z=c } leaving us with the goal path(e,e). The last goal is satisfied by clause 1 and prolog write

true

2. The path predicate

Let's run more querries with the predicate path from prog3. We ask the

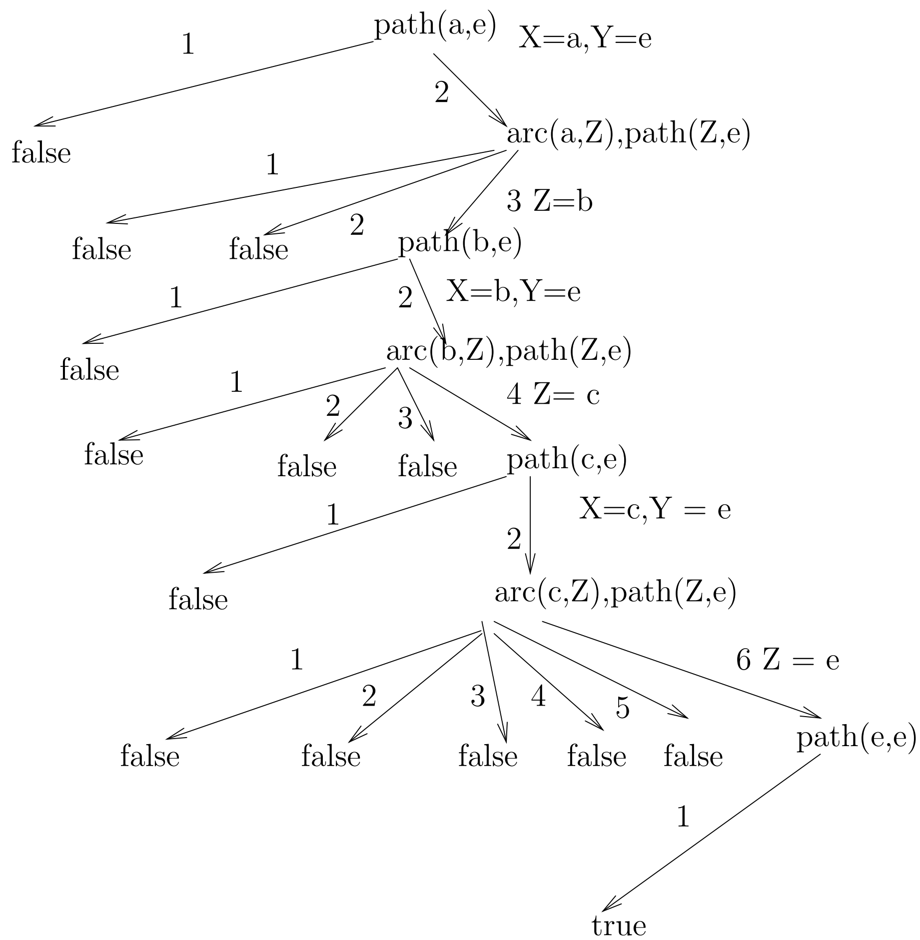

Figure 2: The trace for the query path(a,e) for prog3

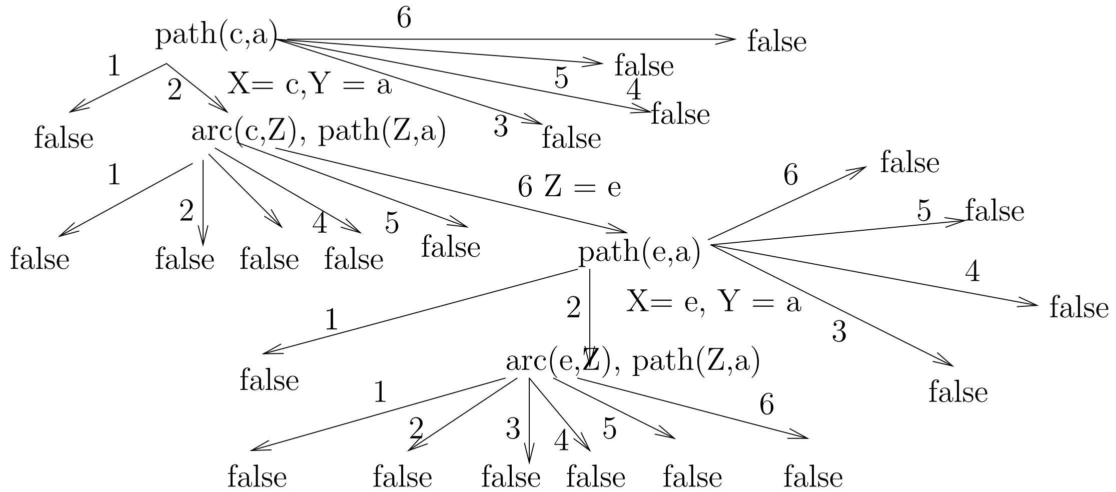

Figure 3: The trace for the query path(c,a) for prog3

query

?- path(c,a).

The interpreter prints

false.

as it should.

Figure 3 traces the execution of this query.

After the goal arc(e,Z) fails, the interpreter backtracks to path(e,a) and tries to resatisfy it with clauses 3,4,5,6. It fails because the functor of the goal is different from the functor of the clauses. This is shown with the dash arrows. from path(e,a), the interpreter backtracks path(c,a) and again tries to resatisfy this goal with the clauses 3,4,5,6. again, it fails for the same reason as path(e,a). So, the top goal fails and the computer prints

false.

However, there are problems with the path predicate, as illustrated in prog4.pl.

```
% define the path based on the arc predicate.
path(X,X).                % clause 1
path(X,Y) :- arc(X,Z), path(Z,Y).            % clause 2
% the arc clauses
arc(a,b).               % clause 3
arc(b,c).               % clause 4
```

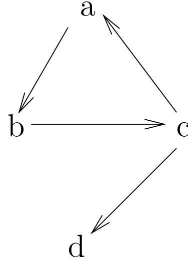

Figure 4: The graph for for prog4

```
arc(c,a).          % clause 5
arc(c,d).          % clause 6
```

The graph described by clauses 3,4,5,6 of prog4 is displayed in Figure 4.

Now let us compile this program and then ask the query below.

?- path(a,d).

We would expect a true answer because there is a path from a to d, namely a⟶b⟶c⟶d.

However, the answer is

ERROR: Out of local stack

which indicates an infinite loop.

So, let us trace the program, as shown in Figure 5. We see that the leaf path(a,d) is the same as the root, so we have a non-terminating loop. So, the loop in the graph, caused a loop in the program!

3. The path2 predicate

One way of avoiding this loop is to rewrite the path predicate. Let us place the recursive call in the second clause in front the arc goal. We have prog5.

```
% define the path2 based on the arc predicate.
path2(X,X).          % clause 1
path2(X,Y) :- path2(X,Z), arc(Z,Y).          % clause 2
% the arc clauses
arc(a,b).          % clause 3
arc(b,c).          % clause 4
arc(c,a).          % clause 5
arc(c,d).          % clause 6
```

After we consult prog5, the query

?- path2(a,d).

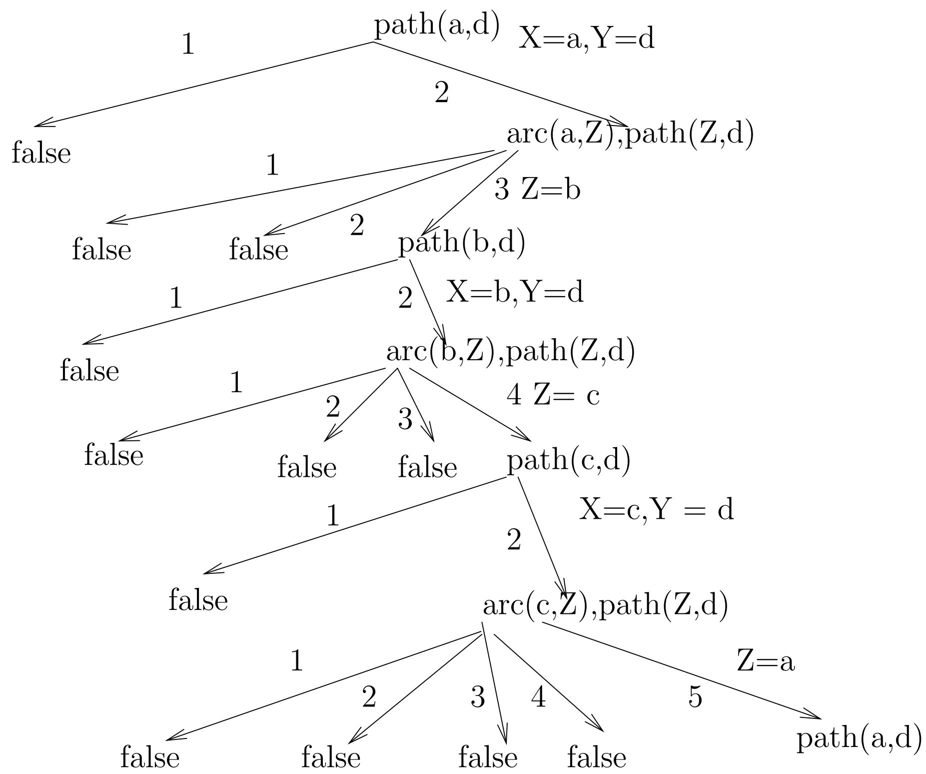

Figure 5: The trace for the query path(a,d) for prog4

produces the expected answer,

true.

Figure 6 traces the execution of the program.

From Figure 6 we can see how path2 works. It first generates the goal arc(a,d). If this fails, it generates the goal arc(a,Z),arc(Z,d). If this fails, it produces the goal arc(a,Z1),arc(Z1,Z),arc(Z,d).

So, for the goal path2(vertex1,vertex2), path2 will first check for a path of length 0, i.e. vertex1 = vertex2. If this fails, it will check if there is path of length 1 from vertex1 to vertex2. If this is not possible, it will try the paths of length 2, 3, 4, and so on. So, it there is finite path from vertex1 to vertex2, path2(vertex1,vertex2) will find it.

What is the drawback of this predicate? If there is no path from vertex1 to vertex2, path2 will try all paths of lengths 0, 1, 2, 3, and so on as before. Except that it will never stop. This is true even if there is no arrow comming out of vertex1.

We will write a better path predicate after we study lists.

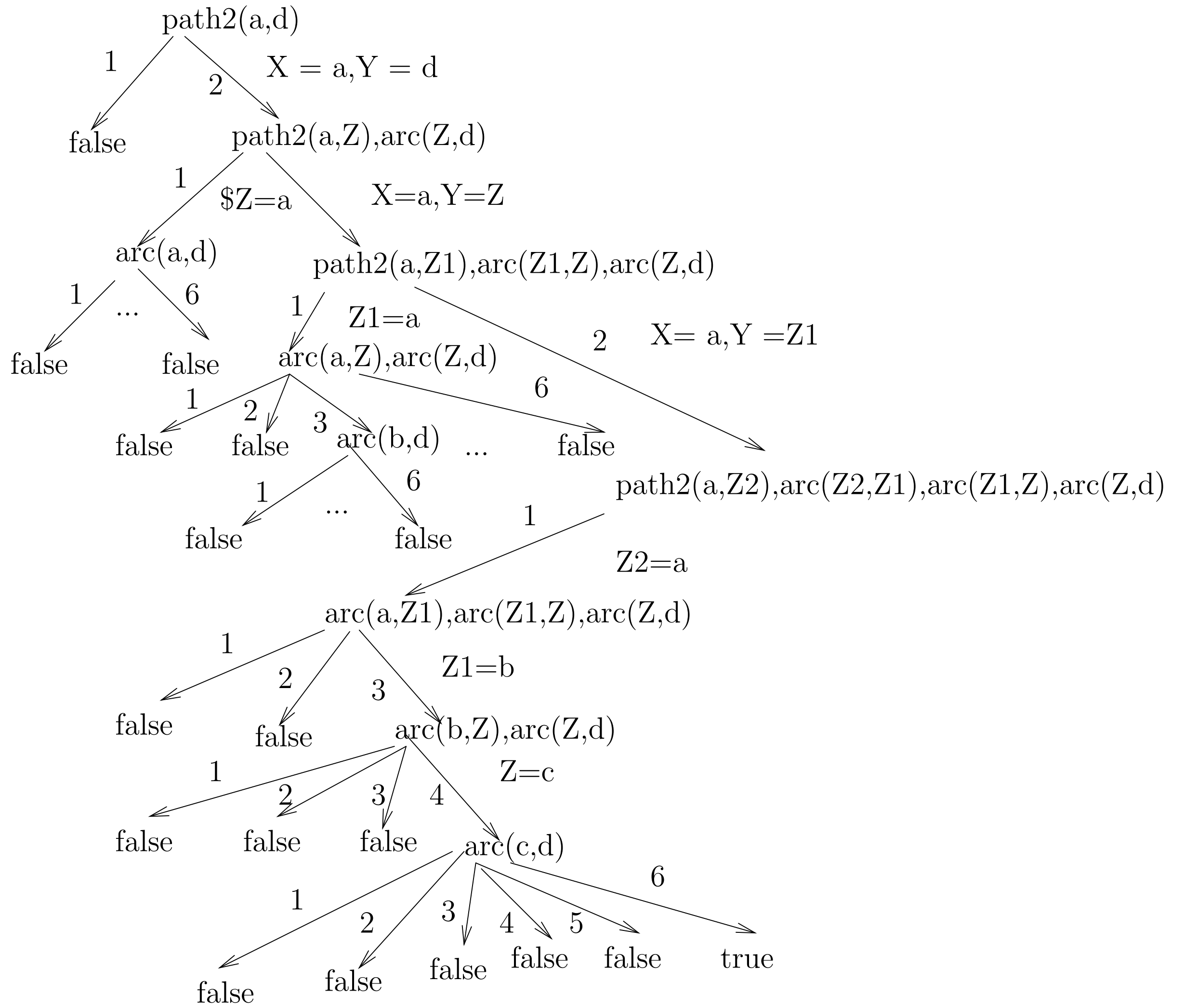

Figure 6: The trace for the query path2(a,d) for prog5