Utrecht University

Introduction to Prolog 2

Recursion and Lists

Alvaro H. C. Correia

27th of May 2019

Utrecht University

Recursion

A First Example

Ancestry

Suppose we have the following family.

```
parent(rik, rob).
parent(rob, ann).
parent(ann, sam).
parent(sam, jon).
```

A First Example

Ancestry

How do we define a predicate to check if `X` is an ancestor of `Y`?

```
parent(rik, rob).
parent(rob, ann).
parent(ann, sam).
parent(sam, jon).
```

A First Example

Ancestry

We know that a parent is an ancestor.

```
parent(rik, rob).
parent(rob, ann).
parent(ann, sam).
parent(sam, jon).
ancestor(X, Y) :- parent(X, Y).
```

```
?- ancestor(rik, rob).
true.
?- ancestor(rob, sam).
false.
```

A First Example

Ancestry

We also know that a grandfather is an ancestor.

```
parent(rik, rob).
parent(rob, ann).
parent(ann, sam).
parent(sam, jon).
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :-
    parent(X, Z),
    parent(Z, Y).
```

```
?- ancestor(rik, rob).
true.
?- ancestor(rob, sam).
true.
```

A First Example

Ancestry

But we can only extend the predicate so many times...

```
parent(rik, rob).
parent(rob, ann).
parent(ann, sam).
parent(sam, jon).
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :-
    parent(X, Z),
    parent(Z, Y).
```

```
?- ancestor(rik, rob).
true.
?- ancestor(rob, sam).
true.
?- ancestor(rik, jon).
false.
```

A First Example

Ancestry

Recursion, however, can be applied an infinite number of times.

```
parent(rik, rob).
parent(rob, ann).
parent(ann, sam).
parent(sam, jon).
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :-
    parent(X, Z),
    ancestor(Z, Y).
```

```
?- ancestor(rik, rob).
true.
?- ancestor(rob, sam).
true.
?- ancestor(rik, jon).
true.
```

A First Example

Ancestry

Why can we not use the anonymous variable instead of Z in the last rule?

```
parent(rik, rob).
parent(rob, ann).
parent(ann, sam).
parent(sam, jon).
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :-
    parent(X, Z),
    ancestor(Z, Y).
```

```
?- ancestor(rik, rob).
true.
?- ancestor(rob, sam).
true.
?- ancestor(rik, jon).
true.
```

Loops in Prolog

There is no explicit loop in Prolog

Consider a standard for loop in C.

Procedural

```
int printer(int N) {
    for (int i=N; i>0; i--) {
        printf("%d", N);
    }
    return 0;
}
```

Declarative

Loops in Prolog

There is no explicit loop in Prolog

In Prolog we have to resort to recursion.

Procedural

```
int printer(int N) {
    for (int i=N; i>0; i--) {
        printf("%d", N);
    }
    return 0;
}
```

Declarative

```
printer(N) :-
    N > 0,
    write(N),
    N2 is N - 1,
    printer(N2).
```

Loops in Prolog

There is no explicit loop in Prolog

In recursion we call the same predicate as condition.

Procedural

```
int printer(int N) {
    for (int i=N; i>0; i--) {
        printf("%d", N);
    }
    return 0;
}
```

Declarative

```
printer(N) :-
    N > 0, % stop condition
    write(N), % statement
    N2 is N - 1, % increment
    printer(N2). % recursion
```

Loops in Prolog

There is no explicit loop in Prolog

Both programs print 54321.

Procedural

```
int printer(int N) {
    for (int i=N; i>0; i--) {
        printf("%d", N);
    }
    return 0;
}
```

Declarative

```
printer(N) :-
    N > 0, % stop condition
    write(N), % statement
    N2 is N - 1, % increment
    printer(N2). % recursion
```

Loops in Prolog

There is no explicit loop in Prolog

Just for the record, there are recursions in C as well.

Procedural

```c
int printer(int N) {
    if (N > 0) {
        printf("%d", N);
        printer(N-1);
    }
    return 0;
}
```

Declarative

```
printer(N) :-
    N > 0, % stop condition
    write(N), % statement
    N2 is N - 1, % increment
    printer(N2). % recursion
```

Loops in Prolog

There is no explicit loop in Prolog

As an exercise try to write a program that prints 12345 instead.

Procedural

```
int printer(int N) {
    if (N > 0) {
        printf("%d", N);
        printer(N-1);
    }
    return 0;
}
```

Declarative

```
printer(N) :-
    N > 0, % stop condition
    write(N), % statement
    N2 is N - 1, % increment
    printer(N2). % recursion
```

Improving the Stop Condition

Base case without failure

Our program stops when `N > 0` fails. It then returns false.

```
printer(N):-
    N > 0, % stop condition
    write(N), % statement
    N2 is N - 1, % increment
    printer(N2). % recursion
```

Improving the Stop Condition

Base case without failure

If the program finds a solution we would rather have it return true.

```
printer(N):-
    N > 0, % stop condition
    write(N), % statement
    N2 is N - 1, % increment
    printer(N2). % recursion
```

Improving the Stop Condition

Base case without failure

Two solutions.

Base case at the beginning

```
printer(N) :- N =< 0.
printer(N):-
    write(N), % statement
    N2 is N - 1, % increment
    printer(N2). % recursion
```

Base case at the end

```
printer(N):-
    N > 0, % stop condition
    write(N), % statement
    N2 is N - 1, % increment
    printer(N2). % recursion
printer(_).
```

A Second Example

Calculating the factorial function

$n! = n \cdot (n-1) \cdot (n-2) ... 3 \cdot 2 \cdot 1$

A Second Example

Calculating the factorial function

`fac(N, F)` is true if `F = N!`

Pseudocode

```
fac(N, F) :-
    fac(N-1, Fx),
    F is N * Fx.
```

The factorial of N is the factorial of N-1 times N.

A Second Example

Calculating the factorial function

`fac(N, F)` is true if `F = N!`

```
fac(N, F) :-
    Nx is N - 1,
    fac(Nx, Fx),
    F is N * Fx.
```

That does not throw errors in Prolog, but the program does not terminate. Why?

A Second Example

Calculating the factorial function

`fac(N, F)` is true if `F = N!`

```
fac(0, 1). % base case
fac(N, F) :-
    Nx is N - 1,
    fac(Nx, Fx),
    F is N * Fx.
```

That does not throw errors in Prolog,

but the program does not terminate.

Why?

We need some sort of stop condition.

In Prolog we call this base case.

A Second Example

Calculating the factorial function

`fac(N, F)` is true if `F = N!`

```
fac(0, 1). % base case
fac(N, F) :-
    Nx is N - 1,
    fac(Nx, Fx),
    F is N * Fx.
```

Our base case has a limitation

It will not stop the interpreter if `N<0`.

A Second Example

Calculating the factorial function

`fac(N, F)` is true if `F = N!`

```
fac(0, 1). % base case
fac(N, F) :-
    N > 0,
    Nx is N - 1,
    fac(Nx, Fx),
    F is N * Fx.
```

Our base case has a limitation

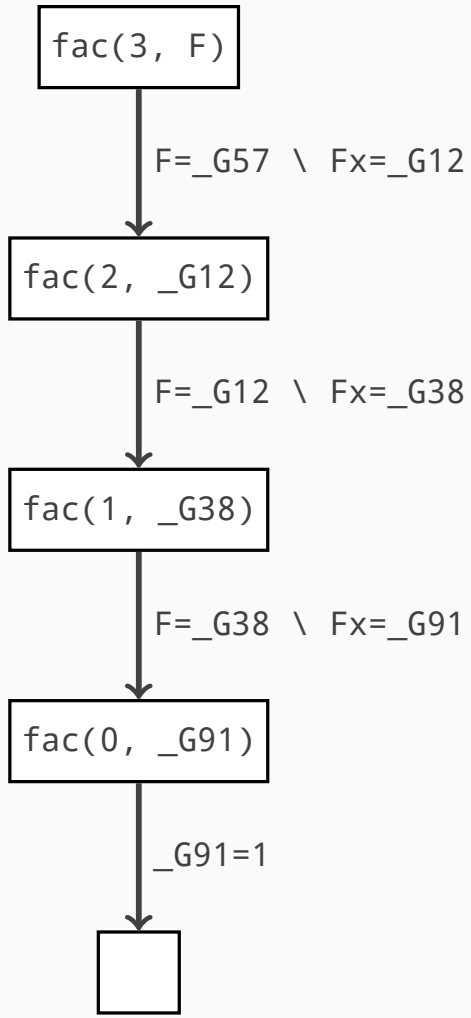
It will not stop the interpreter if `N<0`.

So we check whether `N>0`.

If not, the search fails and Prolog stops there.

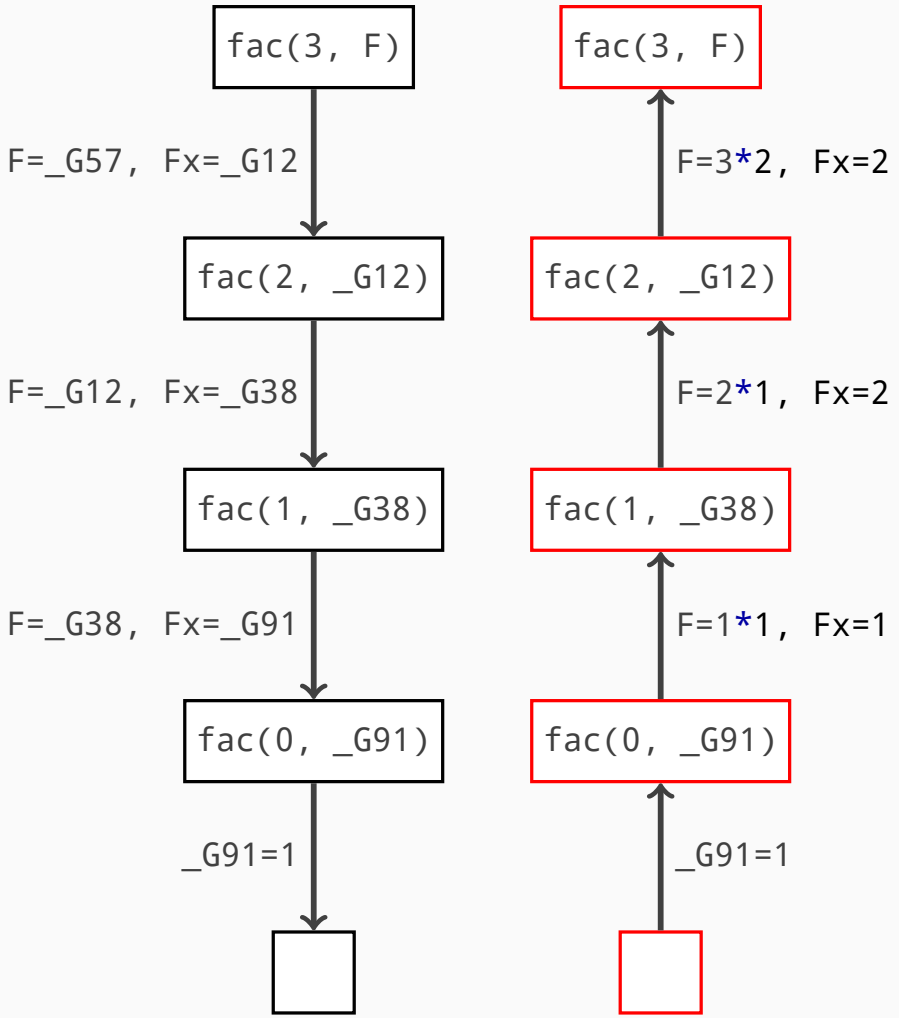
Search Tree - Factorial

```
fac(0, 1). % base case
fac(N, F) :-
    N > 0,
    Nx is N - 1,
    fac(Nx, Fx),
    F is N * Fx.
```



Search Tree - Factorial

```
fac(0, 1). % base case
fac(N, F) :-
    N > 0,
    Nx is N - 1,
    fac(Nx, Fx),
    F is N * Fx.
```



The Order Matters

The base case always come first

What happens when the base case is at the very end?

```
fac(N, F) :-
    N > 0,
    Nx is N - 1,
    fac(Nx, Fx),
    F is N * Fx.
fac(0, 1). % base case
```

The Order Matters

The base case always come first

What happens when the base case is at the very end?

```
fac(N, F) :-
    N > 0,
    Nx is N - 1,
    fac(Nx, Fx),
    F is N * Fx.
fac(0, 1). % base case
```

Remember Prolog goes through the program in order. That is called *resolution with a selection function.*

The Order Matters

The base case always come first

What happens when the base case is at the very end?

```
fac(N, F) :-
    N > 0,
    Nx is N - 1,
    fac(Nx, Fx),
    F is N * Fx.
fac(0, 1). % base case
```

Prolog will always try to match a goal with the recursive rule first and will never reach the base case that would make it stop.

Declarative vs Procedural Meaning

Declarative Meaning

Concerns the relations that hold in the program and thus defines what the output should be.

Procedural Meaning

Concerns how we obtain the output, that is, how the interpreter actually evaluates the relations in the program.

Declarative vs Procedural Meaning

We should consider both meanings when programming in Prolog

Very same declarative meaning, but very different procedural meaning.

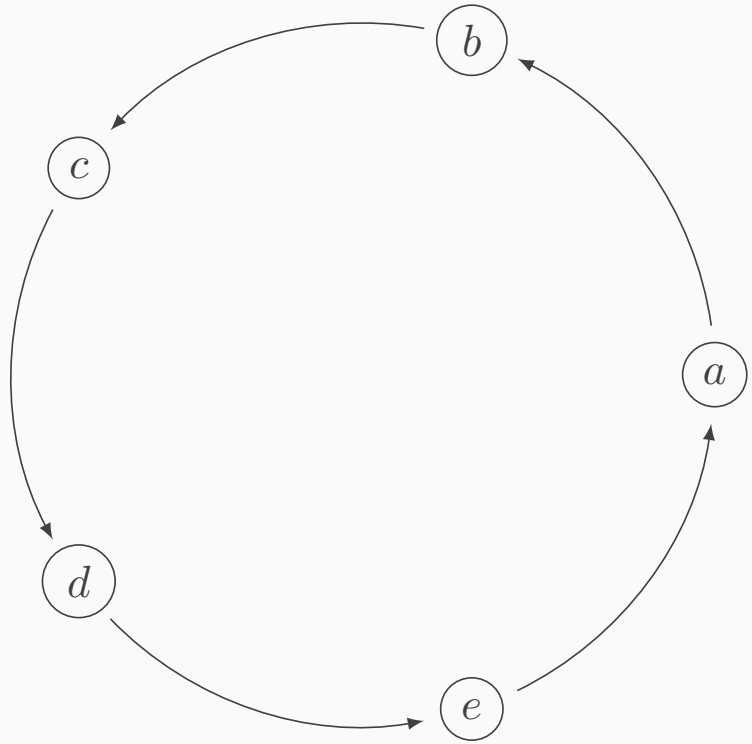
```
fac(0, 1). % base case
fac(N, F) :-
    N>0,
    Nx is N - 1,
    fac(Nx, Fx),
    F is N * Fx.
```

```
fac(N, F) :-
    N>0,
    Nx is N - 1,
    fac(Nx, Fx),
    F is N * Fx.
fac(0, 1). % base case
```

A Third Example

Connections in a graph

How to define all connections in a succinct way?

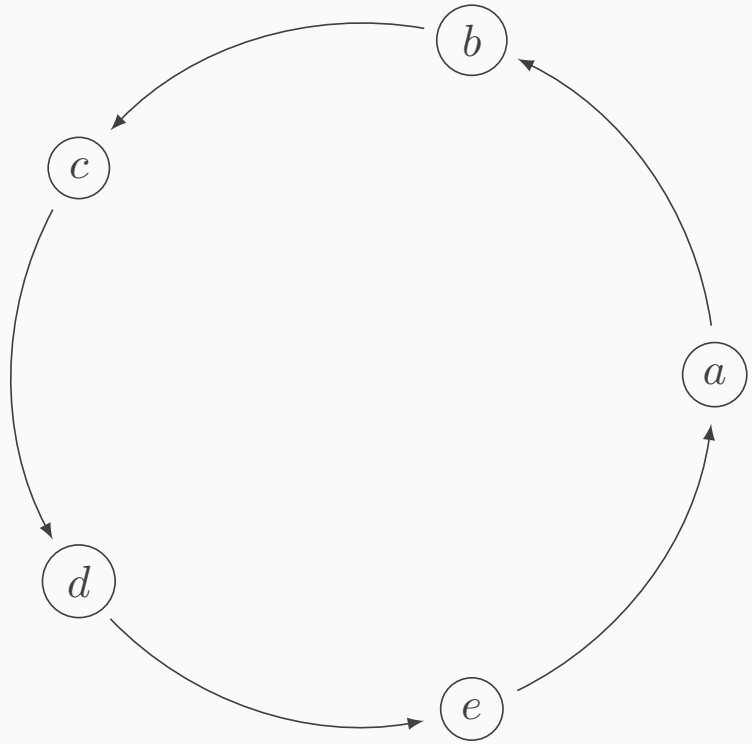


A Third Example

Connections in a graph

We start by defining all arcs in the graph.

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

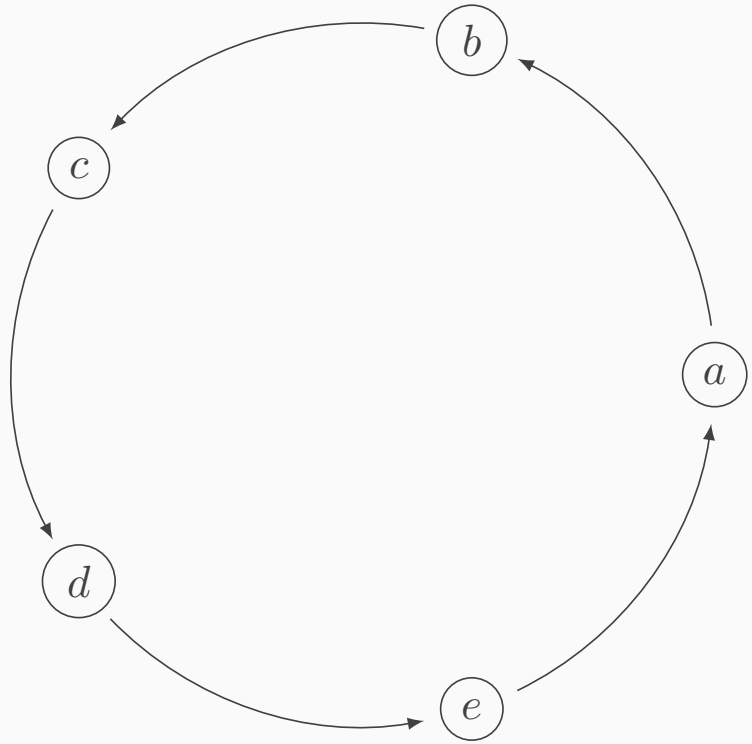


A Third Example

Connections in a graph

We could extend that predicate (abusing the definition of an arc).

```
arc(a, b).
arc(b, c).
arc(c, d).
arc(d, e).
arc(e, a).
arc(X, Y) :-
    arc(X, Z),
    arc(Z, Y).
```



A Third Example

Connections in a graph

However, we get an infinite loop. Why?

```
arc(a, b).
arc(b, c).
arc(c, d).
arc(d, e).
arc(e, a).
arc(X, Y) :-
    arc(X, Z),
    arc(Z, Y).
```

```
?- arc(b, X).
X = c;
X = d;
X = e;
X = a;
X = b;
…
X = c;
X = d;
```

A Third Example

Connections in a graph

One reason is that we have a cycle in the graph.

```
arc(a, b).
arc(b, c).
arc(c, d).
arc(d, e).
arc(e, a).
arc(X, Y) :-
    arc(X, Z),
    arc(Z, Y).
```

```
?- arc(b, X).
X = c;
X = d;
X = e;
X = a;
X = b;
...
X = c;
X = d;
```

A Third Example

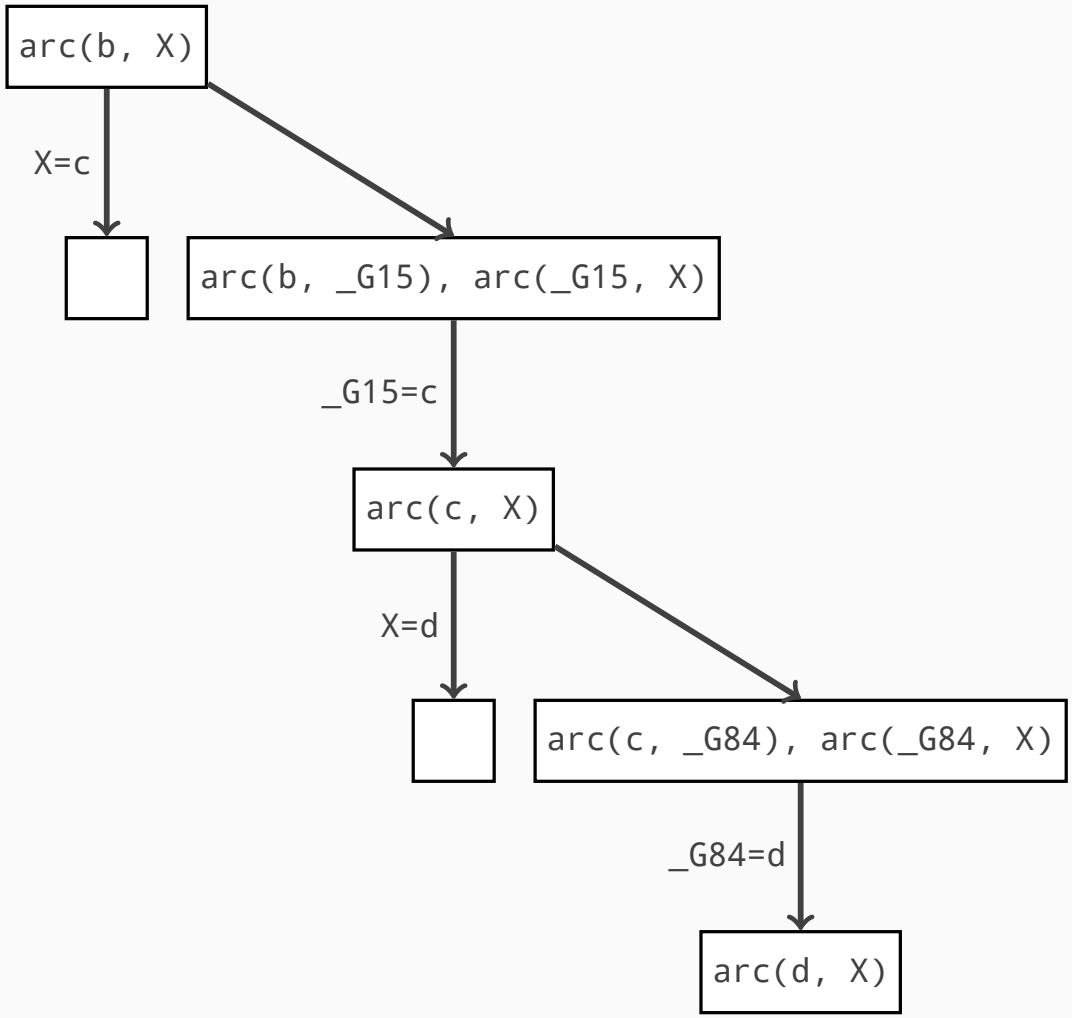
Connections in a graph

But even without the cycle our program breaks! Why?

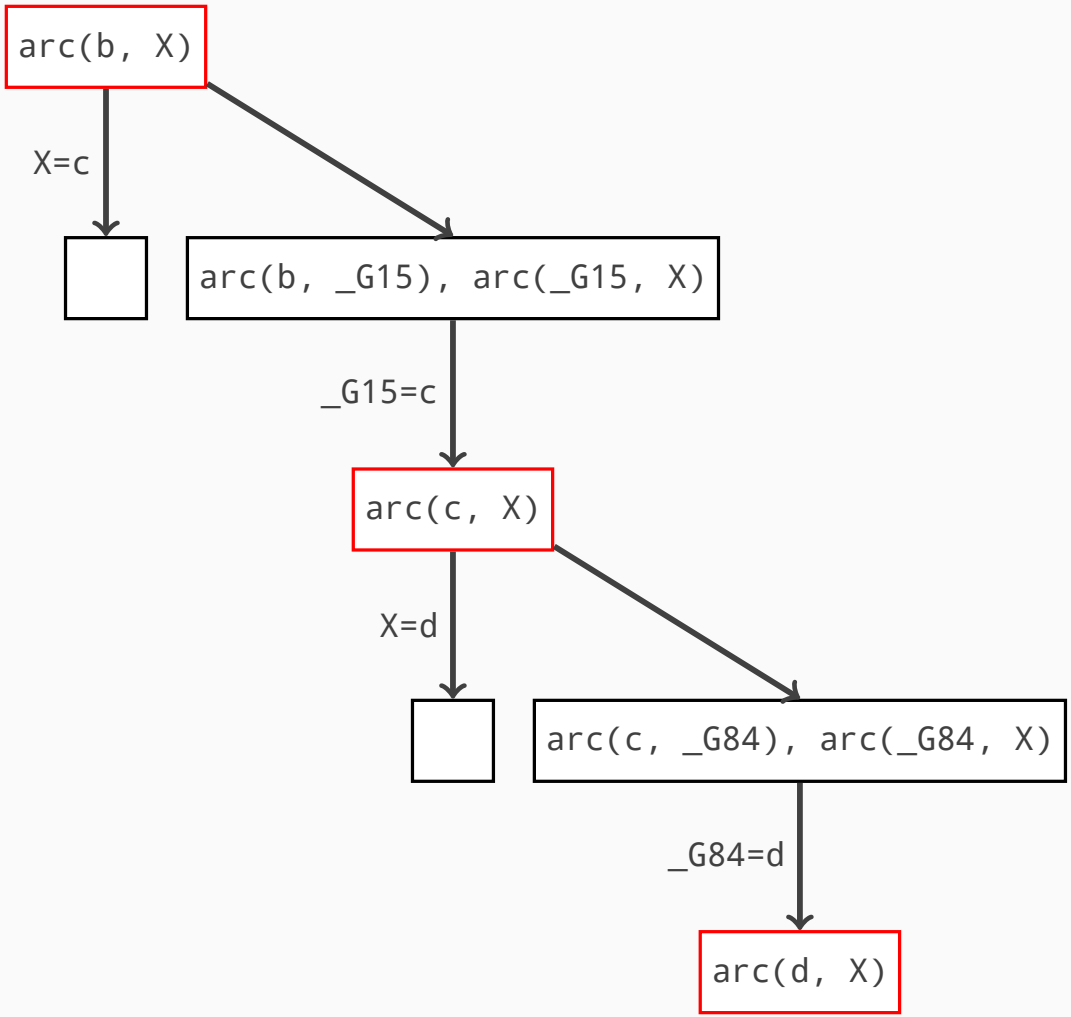
```
arc(a, b).
arc(b, c).
arc(c, d).
% arc(d, e).
arc(e, a).
arc(X, Y) :-
    arc(X, Z),
    arc(Z, Y).
```

```
?- arc(b, X).
X = c;
X = d;
ERROR: Stack limit exceeded
```

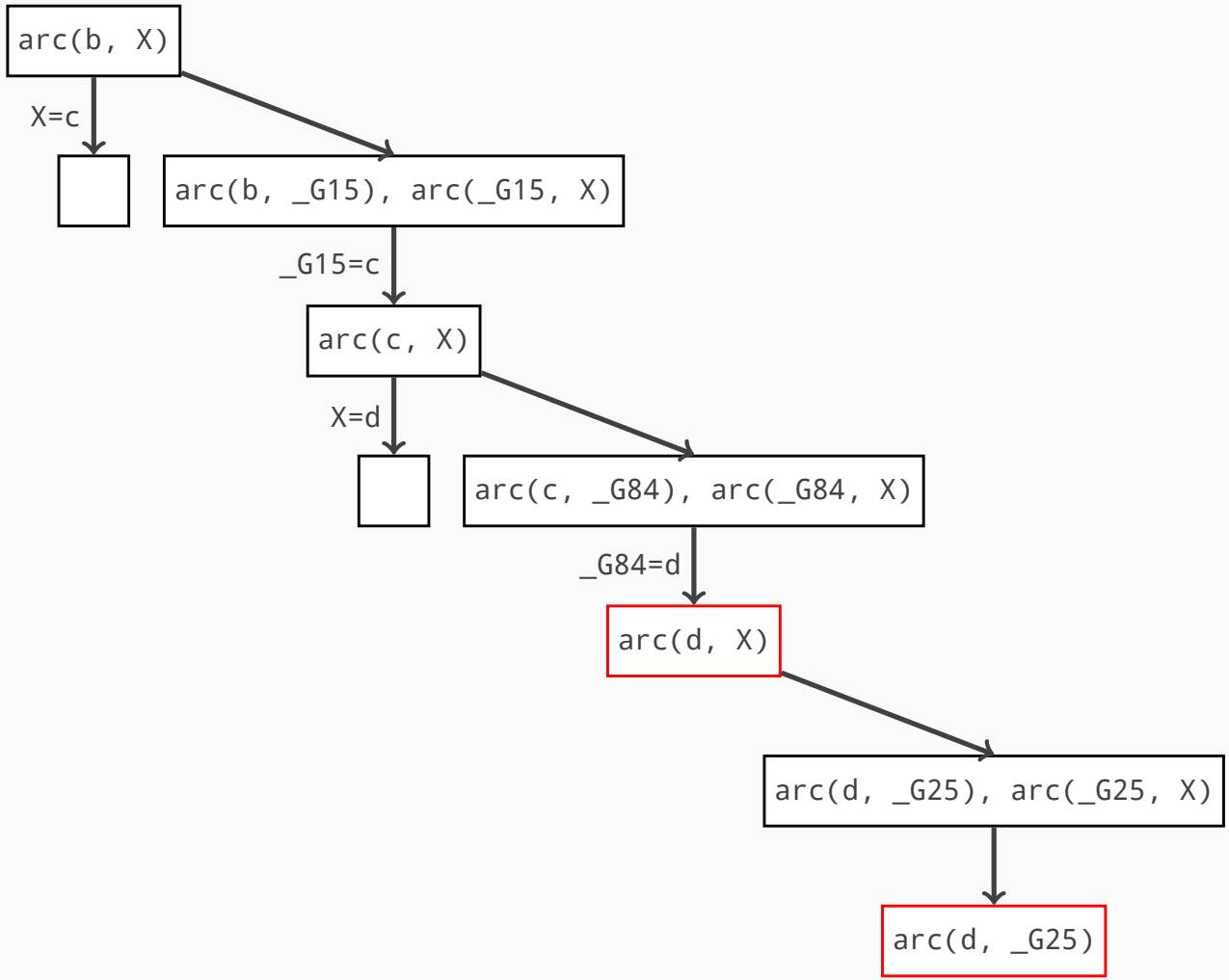
Transitivity Infinite Loop



Transitivity Infinite Loop



Transitivity Infinite Loop



Transitive Closure

We need to define a new predicate

`path` is the *transitive closure* of `arc`.

```
arc(a, b).
arc(b, c).
arc(c, d).
arc(d, e).
arc(e, a).
path(X, Y) :- arc(X, Y).
path(X, Y) :-
    arc(X, Z),
    path(Z, Y).
```

Transitive Closure

We need to define a new predicate

The infinite loop is only due to the cyclic relations.

```
arc(a, b).
arc(b, c).
arc(c, d).
arc(d, e).
arc(e, a).
path(X, Y) :- arc(X, Y).
path(X, Y) :-
    arc(X, Z),
    path(Z, Y).
```

```
?- path(b, X).
X = c;
X = d;
X = e;
X = a;
X = b;
…
X = c;
X = d;
```

Transitive Closure

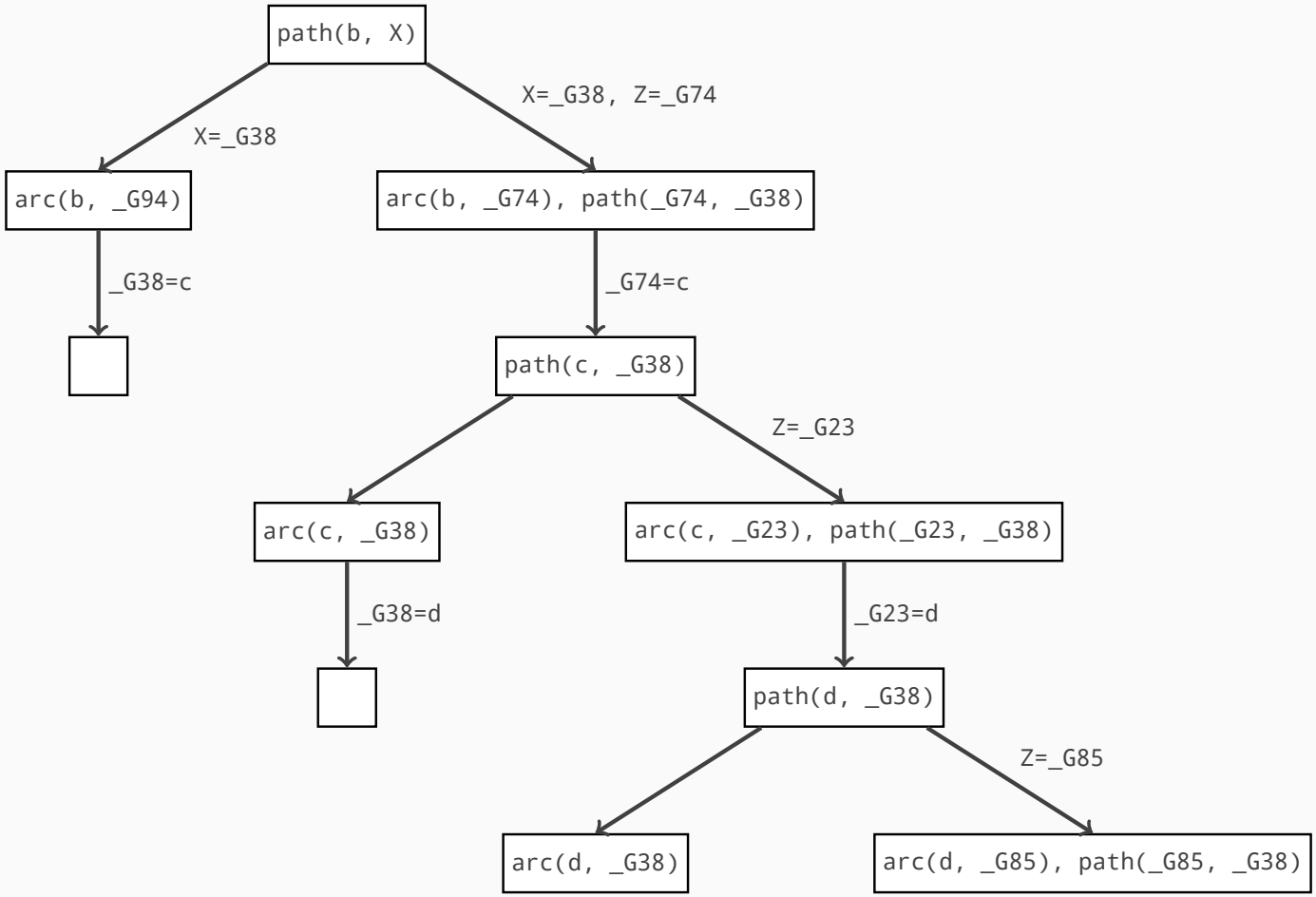
We need to define a new predicate

`path` is properly defined.

```
arc(a, b).
arc(b, c).
arc(c, d).
% arc(d, e).
arc(e, a).
path(X, Y) :- arc(X, Y).
path(X, Y) :-
    arc(X, Z),
    path(Z, Y).
```

```
?- path(b, X).
X = c;
X = d.
```

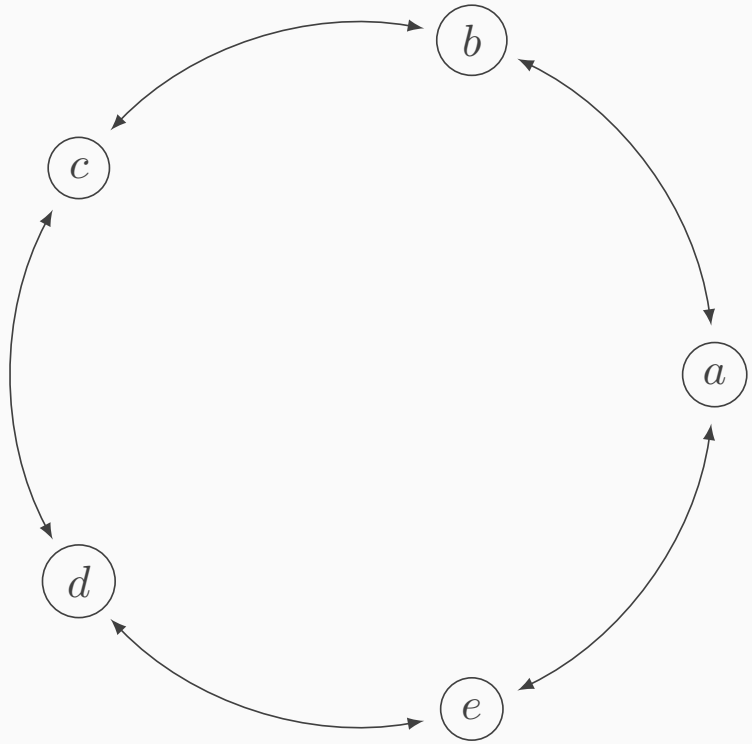
Transitive Closure - Search Tree



Symmetric Connections in a graph

What if we have bidirected arcs?

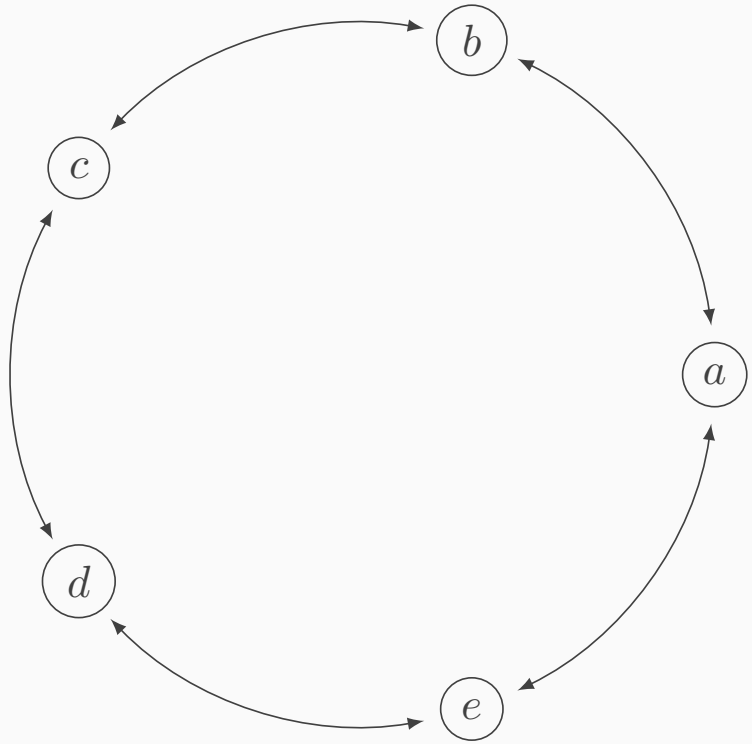
```
arc(a, b).
arc(b, c).
arc(c, d).
arc(d, e).
arc(e, a).
```



Symmetric Connections in a graph

The naive way is to define a symmetric relation for the `arc` predicate.

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



Symmetric Relations

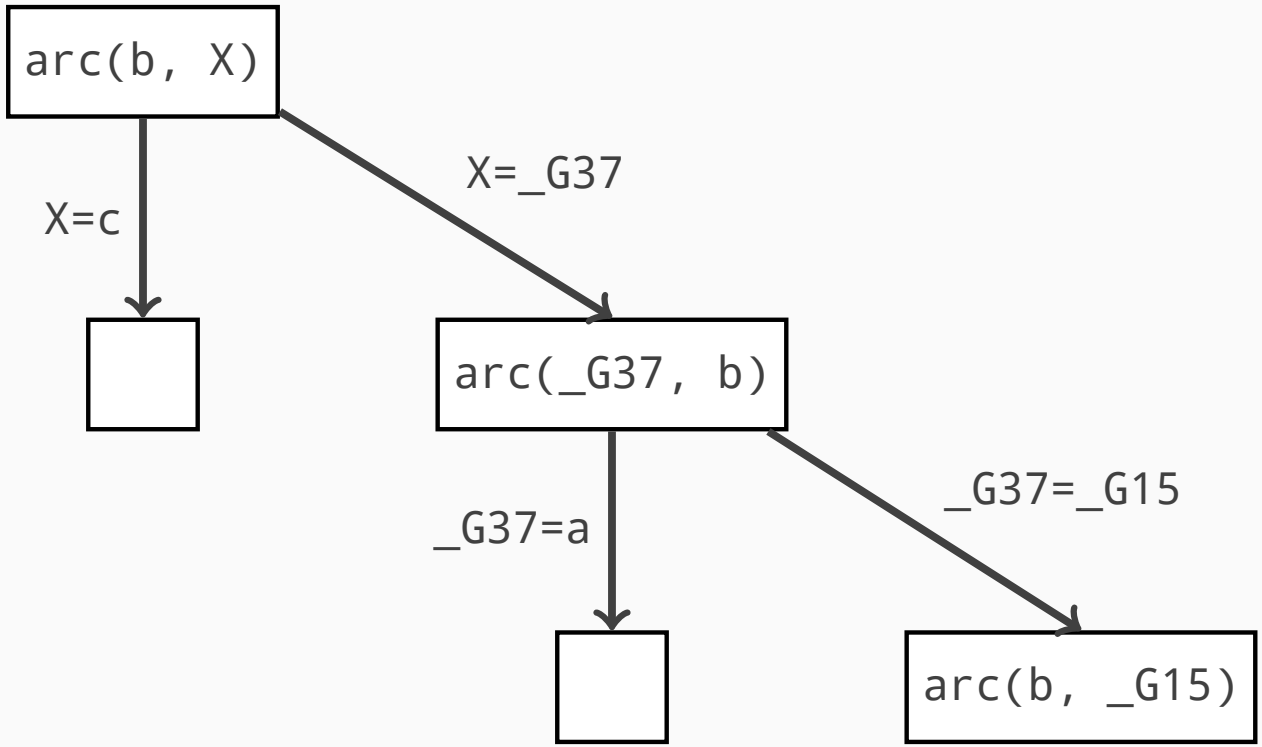
Symmetric Connections in a graph

That leads to infinite loop. Why?

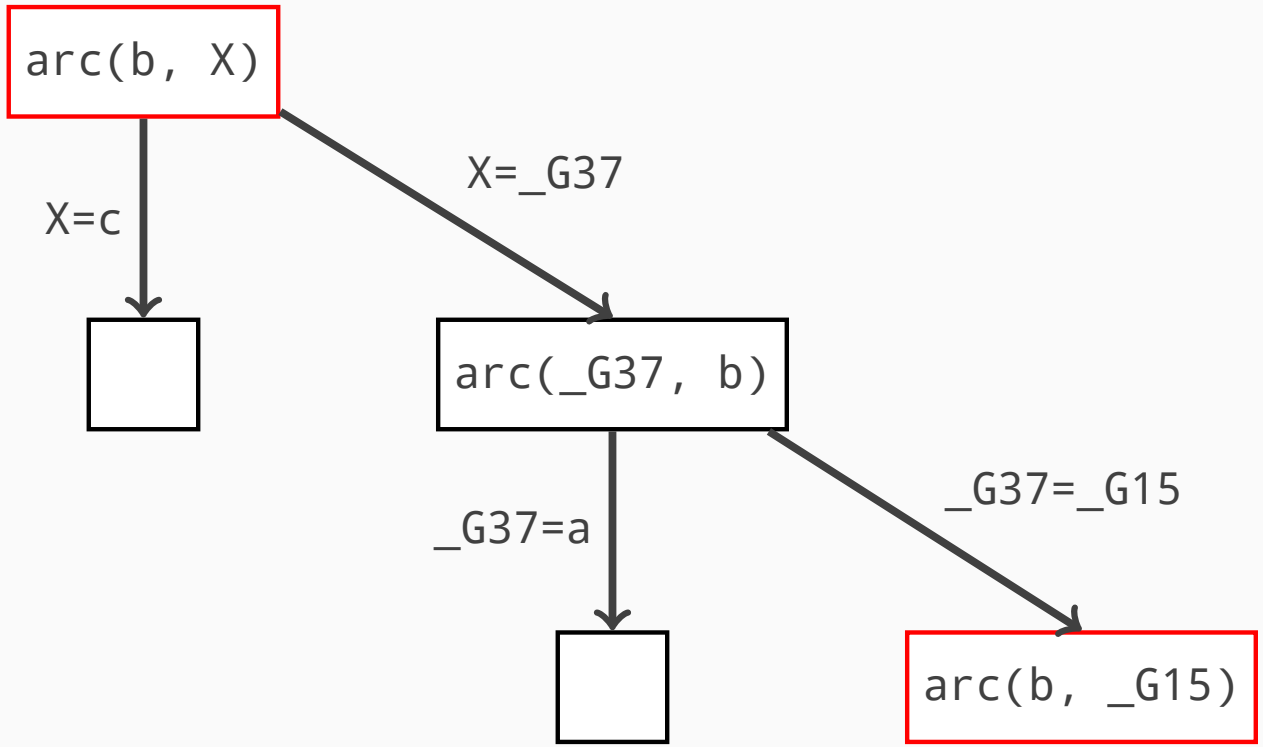
```
arc(a, b).
arc(b, c).
arc(c, d).
arc(d, e).
arc(e, a).
arc(X, Y) :- arc(Y, X).
```

```
?- path(b, X)
X = c;
X = a;
X = c;
X = a;
…
X = c;
X = a;
```

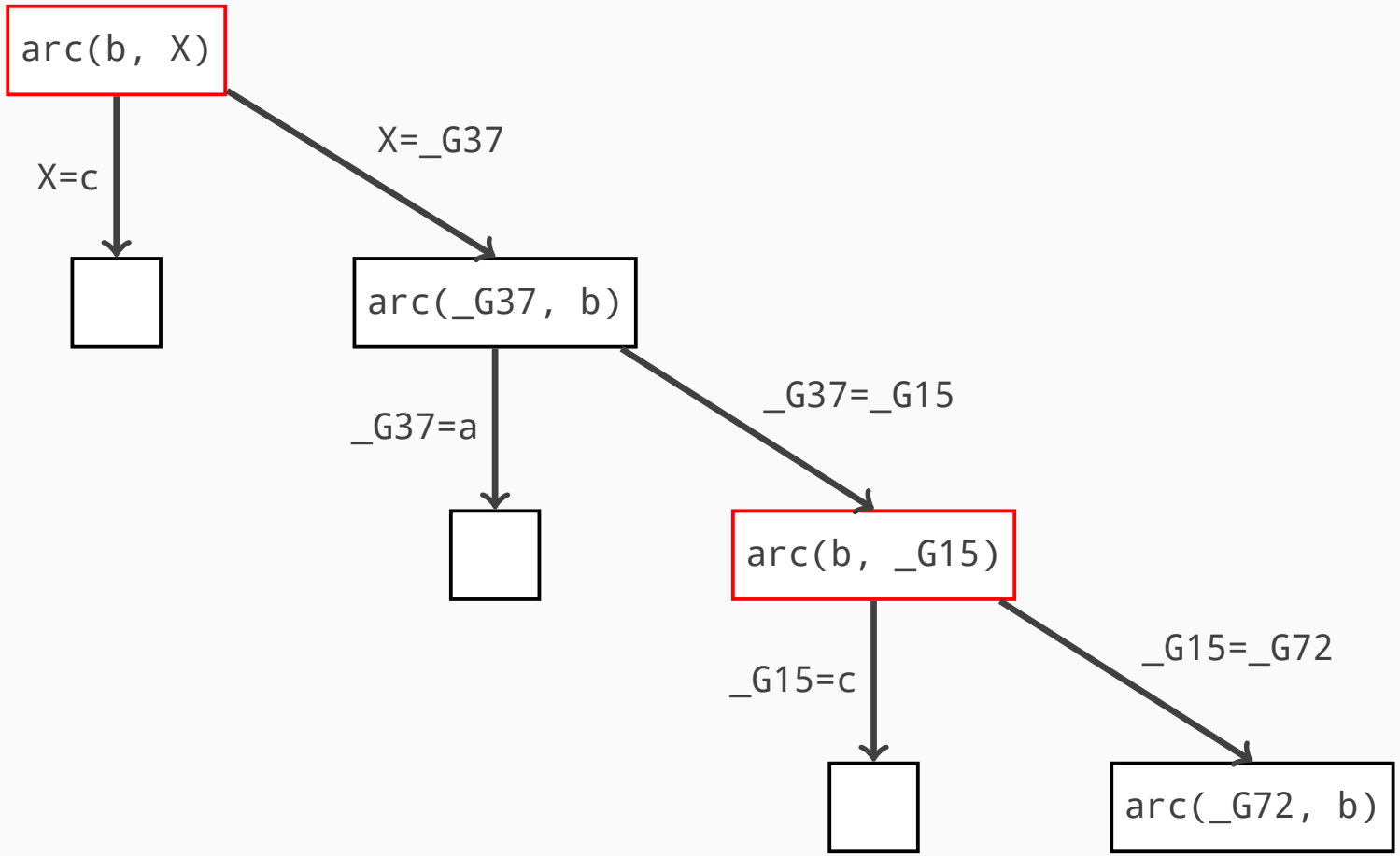
Symmetry Relations - Infinite Loop



Symmetry Relations - Infinite Loop



Symmetry Relations - Infinite Loop



Symmetric Closure

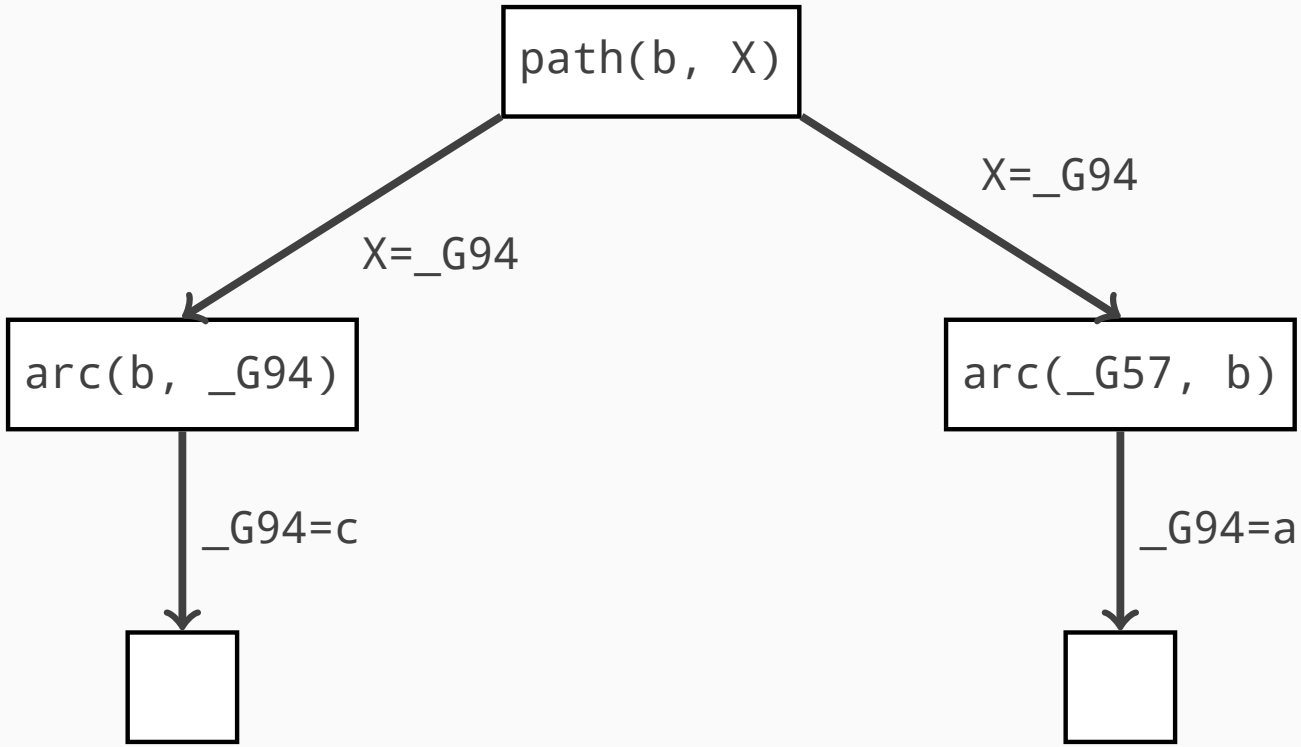
Again we need to define a new predicate

`path` is the *symmetric closure* of `arc`.

```
arc(a, b).
arc(b, c).
arc(c, d).
arc(d, e).
arc(e, a).
path(X, Y) :- arc(X, Y).
path(X, Y) :- arc(Y, X).
```

```
?- path(b, X).
X = c;
X = a.
```

Symmetric Closure - Search Tree



Closures

Symmetric Closure

If it holds for `(X, Y)`, then it must hold for `(Y, X)`.

```
path(X, Y) :- arc(X, Y).
path(X, Y) :- arc(Y, X).
```

Transitive Closure

If it holds for `(X, Z)` and `(Z, Y)`, then it must hold for `(X, Y)`.

```
path(X, Y) :- arc(X, Y).
path(X, Y) :- arc(X, Z), path(Z, Y).
```

Combining both closures

We can define both the symmetric and transitive closures

That solves our problem if there are no cycles in the graph.

```
arc(a, b).
arc(b, c).
arc(c, d).
% arc(d, e).
arc(e, a).
path(X, Y) :- arc(X, Y).
path(X, Y) :- arc(Y, X).
path(X, Y) :-
    arc(X, Z),
    path(Z, Y).
```

```
?- path(b, X).
X = c;
X = a;
X = d;
X = b;
X = c.
```

Combining both closures

We can define both the symmetric and transitive closures

But the problem remains if there is a cycle.

```
arc(a, b).
arc(b, c).
arc(c, d).
arc(d, e).
arc(e, a).
path(X, Y) :- arc(X, Y).
path(X, Y) :- arc(Y, X).
path(X, Y) :-
    arc(X, Z),
    path(Z, Y).
```

```
?- path(b, X).
X = c;
X = a;
X = d;
X = b;
X = c.
…
X = c;
X = a;
…
```

Combining both closures

We can define both the symmetric and transitive closures

To solve this we need to keep track of the nodes we visited. We need lists!

```
arc(a, b).
arc(b, c).
arc(c, d).
arc(d, e).
arc(e, a).
path(X, Y) :- arc(X, Y).
path(X, Y) :- arc(Y, X).
path(X, Y) :-
    arc(X, Z),
    path(Z, Y).
```

```
?- path(b, X).
X = c;
X = a;
X = d;
X = b;
X = c.
…
X = c;
X = a;
…
```

Lists

Examples of Lists

A list is simply a collection of Prolog data objects.

We enclose a list with a pair of brackets.

`[a, b, c, d]` list of atoms

Examples of Lists

A list is simply a collection of Prolog data objects.

A list may contain any combination of Prolog objects.

`[a, b, c, d]`	list of atoms
`[a, B, f(c), 42, 'list']`	list of anything

Examples of Lists

A list is simply a collection of Prolog data objects.

Including list elements.

`[a, b, c, d]`	list of atoms
`[a, B, f(c), 42, 'list']`	list of anything
`[a, [a, b], [a, b, c]]`	list of lists

Examples of Lists

A list is simply a collection of Prolog data objects.

The empty list is a special and very useful list.

`[a, b, c, d]`	list of atoms
`[a, B, f(c), 42, 'list']`	list of anything
`[a, [a, b], [a, b, c]]`	list of lists
`[]`	empty list

List Matching

- A list only matches with another list.

```
?- [1, 2, 3] = [1, 2, 3].
true.
```

- **Order** and **number** of elements must be the same.

```
?- [1, 2, 3] = [1, 3, 2].
false.
```

- A variable can only match a **single** list element.

```
?- [1, 2, X] = [1, 2, 3].
X = 3.
?- [1, X] = [1, 2, 3].
false.
```

List Matching

- A variable can only match a **single** list element.

```
?- [1, 2, X] = [1, 2, 3].
X = 3.
?- [1, X] = [1, 2, 3].
false.
```

  But that element can be itself a list.

```
?- [1, X] = [1, [2, 3]].
X = [2, 3].
```

Heads and Tails

Head is the first element, tail is a list with the remaining ones

We use | to separate head and tail in a list.

```
?- [Head | Tail] = [a, b, c, d].
```

Heads and Tails

Head is the first element, tail is a list with the remaining ones

We use | to separate head and tail in a list.

```
?- [Head | Tail] = [a, b, c, d].
Head = a.
Tail = [b, c, d].
```

Heads and Tails

Head is the first element, tail is a list with the remaining ones

`Head` and `Tail` are just arbitrary variable names.

```
?- [Head | Tail] = [a, b, c, d].
Head = a.
Tail = [b, c, d].
?- [X | Y] = [].
```

Heads and Tails

Head is the first element, tail is a list with the remaining ones

The empty list has no head or tail.

```
?- [Head | Tail] = [a, b, c, d].
Head = a.
Tail = [b, c, d].
?- [X | Y] = [].
false.
```

Heads and Tails

Head is the first element, tail is a list with the remaining ones

We can also use | to extract the first *n* elements.

```
?- [Head | Tail] = [a, b, c, d].
Head = a.
Tail = [b, c, d].
?- [X | Y] = [].
false.
?- [X, Y | Z] = [a, b, c, d].
```

Heads and Tails

Head is the first element, tail is a list with the remaining ones

We can also use | to extract the first *n* elements.

```
?- [Head | Tail] = [a, b, c, d].
Head = a.
Tail = [b, c, d].
?- [X | Y] = [].
false.
?- [X, Y | Z] = [a, b, c, d].
X = a.
Y = b.
Z = [c,d].
```

Heads and Tails

Head

The first element of a list. It is a **single element**.

```
?- [H|_] = [1, 2, 3].
H = 1.
```

Tail

The remaining elements of a list. It is itself a **list**.

```
?- [_|T] = [1, 2, 3].
T = [2, 3].
```

Going Through a List

How do we print out the elements of a list?

Assume the following facts about fruit prices. How can we query the price of multiple fruits at once?

```
price(apple, 0.5).
price(banana, 0.25).
price(lime, 1.0).
```

Going Through a List

How do we print out the elements of a list?

1. Define our base case: the empty list.

```
price(apple, 0.5).
price(banana, 0.25).
price(lime, 1.0).
getPrice([]).
```

Going Through a List

How do we print out the elements of a list?

1. Define our base case: the empty list.
2. Get the value of the head.

```
price(apple, 0.5).
price(banana, 0.25).
price(lime, 1.0).
getPrice([]).
getPrice([H|T]):-
    price(H, X),
    writeln(X),
```

Going Through a List

How do we print out the elements of a list?

1. Define our base case: the empty list.
2. Get the value of the head.
3. Call the same predicate recursively on the tail.

```
price(apple, 0.5).
price(banana, 0.25).
price(lime, 1.0).
getPrice([]).
getPrice([H|T]):-
    price(H, X),
    writeln(X),
    getPrice(T).
```

Going Through a List

How do we print out the elements of a list?

1. Define our base case: the empty list.
2. Get the value of the head.
3. Call the same predicate recursively on the tail.

```
price(apple, 0.5).
price(banana, 0.25).
price(lime, 1.0).
getPrice([]).
getPrice([H|T]):-
    price(H, X),
    writeln(X),
    getPrice(T).
```

```
?-getPrice([apple, banana, lime]).
0.5
0.25
1.0
```

Returning a List

We can also return lists.

1. Our base case now compares two empty lists.

```
price(apple, 0.5).
price(banana, 0.25).
price(lime, 1.0).
getPrice2([], []).
```

Returning a List

We can also return lists.

1. Our base case now compares two empty lists.
2. Match the values of both heads.

```
price(apple, 0.5).
price(banana, 0.25).
price(lime, 1.0).
getPrice2([], []).
getPrice2([H|T], [H2|T2]):-
    price(H, H2),
```

Returning a List

We can also return lists.

1. Our base case now compares two empty lists.
2. Match the values of both heads.
3. Call the same predicate recursively on the tails.

```
price(apple, 0.5).
price(banana, 0.25).
price(lime, 1.0).
getPrice2([], []).
getPrice2([H|T], [H2|T2]):-
    price(H, H2),
    getPrice2(T, T2).
```

Returning a List

We can also return lists.

1. Our base case now compares two empty lists.
2. Match the values of both heads.
3. Call the same predicate recursively on the tails.

```
price(apple, 0.5).
price(banana, 0.25).
price(lime, 1.0).
getPrice2([], []).
getPrice2([H|T], [H2|T2]):-
    price(H, H2),
    getPrice2(T, T2).
```

```
?-getPrice2([apple, lime], L).
L = [0.5, 1.0].
```

member

Checking whether an element belongs to a list.

SWI-Prolog has a built-in member/2 predicate.

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

member

Checking whether an element belongs to a list.

`member(?Elem, ?List)`: True if `Elem` is a member of `List`

```
?- member(a, [a, b, c, d]).
true.
```

member

Checking whether an element belongs to a list.

We can also query the members of a list.

```
?- member(a, [a, b, c, d]).
true.
?- member(X, [a, b, c, d]).
```

member

Checking whether an element belongs to a list.

We can also query the members of a list.

```
?- member(a, [a, b, c, d]).
true.
?- member(X, [a, b, c, d]).
X=a ;
…
X=d .
```

member

Checking whether an element belongs to a list.

What happens in this query?

```
?- member(a, [a, b, c, d]).
true.
?- member(X, [a, b, c, d]).
X=a ;
…
X=d .
?- member(b, [a, [b], c, d]).
```

member

Checking whether an element belongs to a list.

`b` is not a member of the list, but `[b] is`.

```
?- member(a, [a, b, c, d]).
true.
?- member(X, [a, b, c, d]).
X=a ;
…
X=d .
?- member(b, [a, [b], c, d]).
false.
?- member([b], [a, [b], c, d]).
true.
```

Implementing the member Predicate

We start with a base case.

The smallest possible example of the problem.

Implementing the member Predicate

We start with a base case.

If an element is the head of a list, it is a member of that list.

```
mem(X, [X|T]).
```

Implementing the member Predicate

We start with a base case.

This already works for some simple examples.

```
mem(X, [X|T]).
```

```
?- mem(a, [a,b,c]).
true.
?- mem(a, [c,b,a]).
false.
```

Implementing the member Predicate

Recurse down the list towards the base case.

We keep splitting the problem until we get to the base case.

```
mem(X, [X|T]).
mem(X, [_|T]) :-
    mem(X, T).
```

Implementing the member Predicate

Recurse down the list towards the base case.

This works for every list.

```
mem(X, [X|T]).
mem(X, [_|T]) :-
    mem(X, T).
```

```
?- mem(a, [a,b,c]).
true.
?- mem(a, [c,b,a]).
true.
?- mem(b, [a, [b], c, d]).
false.
```

length

Getting the length of a list.

SWI-Prolog has a built-in length/2 predicate.

```
?- length([a, b, c, d], 4).
```

length

Getting the length of a list.

`length(?List, ?Int)`: True if `Int` is the number of elements in `List`

```
?- length([a, b, c, d], 4).
true.
```

length

Getting the length of a list.

We can also query the length of a list using a variable.

```
?- length([a, b, c, d], 4).
true.
?- length([a, b, c, d], X).
```

length

Getting the length of a list.

We can also query the length of a list using a variable.

```
?- length([a, b, c, d], 4).
true.
?- length([a, b, c, d], X).
X=4.
```

length

Getting the length of a list.

We can also query the length of a list using a variable.

```
?- length([a, b, c, d], 4).
true.
?- length([a, b, c, d], X).
X=4.
?- length([a, [b, c], [d, e]], X).
```

length

Getting the length of a list.

We count only the elements inside the the outermost brackets.

```
?- length([a, b, c, d], 4).
true.
?- length([a, b, c, d], X).
X=4.
?- length([a, [b, c], [d, e]], X).
X=3.
```

Implementing the length Predicate

Getting the length of a list.

We start with a base case: the empty list.

```
len([], 0).
```

Implementing the length Predicate

Getting the length of a list.

The length of any list is the length of its tail plus one.

```
len([], 0).
len([_|T], L) :-
    len(T, Lx),
    L is Lx + 1.
```

Implementing the length Predicate

Getting the length of a list.

And it works.

```
len([], 0).
len([_|T], L) :-
    len(T, Lx),
    L is Lx + 1.
```

```
?- len([a, [b, c], d], X) :-
X = 3.
?- len([a, b, c], 3).
X = true.
```

append/2

Appending one list to another.

SWI-Prolog has a built-in append/2 predicate.

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

append/2

Appending one list to another.

`append(+ListOfLists, ?List)`:

True if `List` is the concatenation of the lists in `ListOfLists`

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

append/2

Appending one list to another.

We can also query the concatenation of two lists.

```
?- append([[a,b], [c, d]], [a, b, c, d]).
true.
?- append([[a,b], [c, d]], L).
```

append/2

Appending one list to another.

We can also query the concatenation of two lists.

```
?- append([[a,b], [c, d]], [a, b, c, d]).
true.
?- append([[a,b], [c, d]], L).
L = [a, b, c, d].
```

append/2

Appending one list to another.

Or even the possible splits of a list into two.

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

append/3

Combining two lists into one.

SWI-Prolog also has a built-in append/3 predicate.

append/3

Combining two lists into one.

`append(?List1, ?List2, ?List1AndList2)`:

True if `List1AndList2` is the concatenation of the lists in `List1` and `List2`

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

sort/2

Sorting a list.

SWI-Prolog has a built-in sort/2 predicate.

```
?- sort([3,1,2], [1, 2, 3]).
true.
```

sort/2

Sorting a list.

`sort(+List, -Sorted)`:

True if `Sorted` has the same elements of `List` in a sorted order.

```
?- sort([3,1,2], [1, 2, 3]).
true.
?- sort([brasil, colombia, argentina], L).
L = [argentina, brasil, colombia].
```

sort/2

Sorting a list.

Standard order of terms.

```
?- sort([3,1,2], [1, 2, 3]).
true.
?- sort([brasil, colombia, argentina], L).
L = [argentina, brasil, colombia].
?- sort([arc(a, b), 3, a, [c, 4], 1+1], L).
L = [3, a, 1+1, [c, 4], arc(a, b)].
```

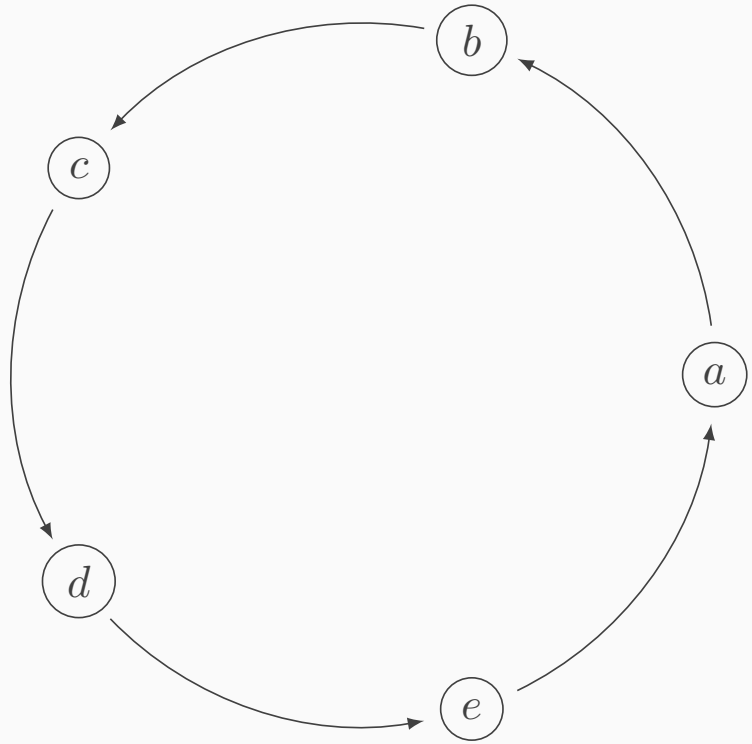
Returning Unique Solutions

Returning Unique Solutions

We keep track of previous paths using a list.

We start by redefining the `arcs` predicate

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

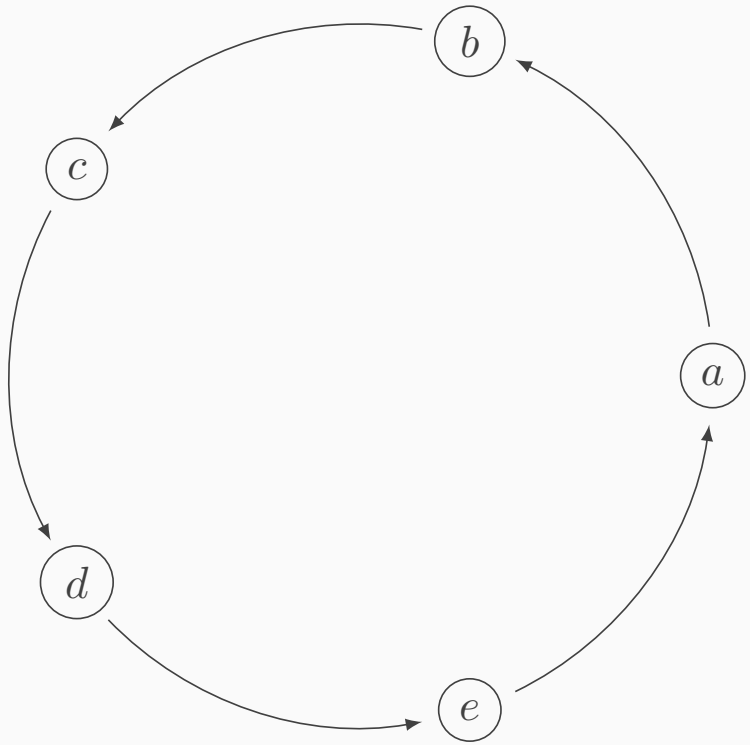


Returning Unique Solutions

We keep track of previous paths using a list.

path/3 has an extra element: the list of nodes between X and Y.

```
arcs(a, [b]).
arcs(b, [c]).
arcs(c, [d]).
arcs(d, [e]).
arcs(e, [a]).
path( X, Y, P ) :-
    path( X, Y, [X], P ),
    X\=Y.
```

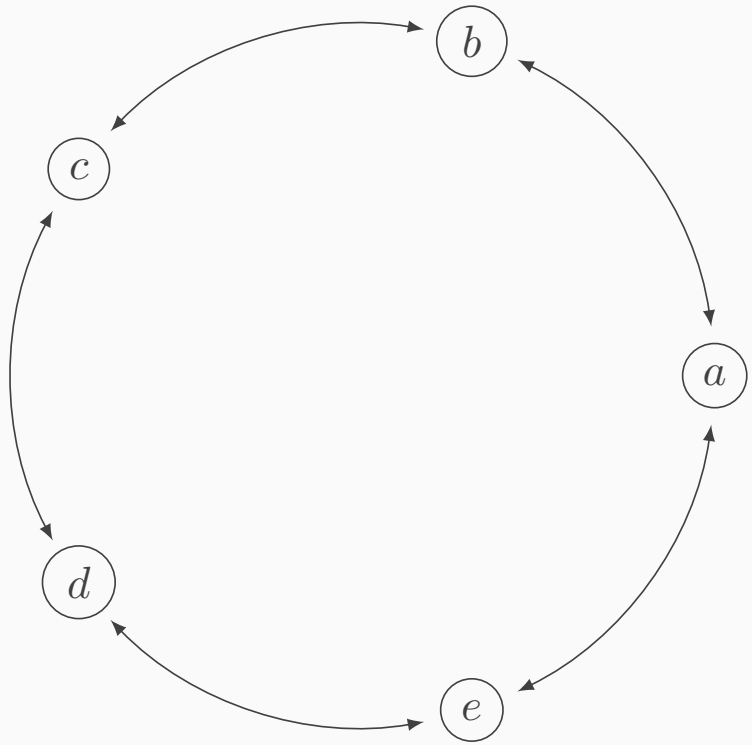


Returning Unique Solutions

We keep track of previous paths using a list.

We add the symmetric closure of path/4.

```
arcs(a, [b]).
arcs(b, [c]).
arcs(c, [d]).
arcs(d, [e]).
arcs(e, [a]).
path( X, Y, P ) :-
    path( X, Y, [X], P ),
    X\=Y.
path( X, Y, P ) :-
    path( Y, X, [Y], P ),
    X\=Y.
```



Returning Unique Solutions

We keep track of previous paths using a list.

path/4 is the transitive closure. `PIn` is the path so far, `POut` is the final path.

```
arcs(a, [b]).
arcs(b, [c]).
arcs(c, [d]).
arcs(d, [e]).
arcs(e, [a]).
path( X, Y, P ) :-
    path( X, Y, [X], P ),
    X\=Y.
path( X, Y, P ) :-
    path( Y, X, [Y], P ),
    X\=Y.
```

```
path( X, X, Path, Path ).
path( X, Y, PIn, POut ):-
    arcs( X, L ),
    member( Z, L ),
    not(member( Z, PIn )),
    path( Z, Y, [Z|PIn], POut ).
```

Returning Unique Solutions

We keep track of previous paths using a list.

Now we get all possible paths without repetitions.

```
?- path(b, X, L).
X = c, L = [b, c] ;
X = d, L = [b, c, d] ;
X = e, L = [b, c, d, e] ;
X = a, L = [b, c, d, e, a] ;
X = a, L = [b, a] ;
X = c, L = [b, a, e, d, c] ;
X = d, L = [b, a, e, d] ;
X = e, L = [b, a, e] ;
```

Summary

Recursion

There is no explicit loop in Prolog.

No while, until or for.

Repetition by recursion.

- Call the same predicate as condition.
- Stop when a special or base case is reached.

Recursion

There is no explicit loop in Prolog.

No while, until or for.

Repetition by recursion.

- Call the same predicate as condition.
- Stop when a special or base case is reached.

Base and special cases should come first!

Procedural vs. **Declarative** meanings

Special care when defining symmetric and transitive relations!

Create new predicate name to avoid infinite loops.

Closures

Symmetric Closure

If it holds for `(X, Y)`, then it must hold for `(Y, X)`.

```
path(X, Y) :- arc(X, Y).
path(X, Y) :- arc(Y, X).
```

Transitive Closure

If it holds for `(X, Z)` and `(Z, Y)`, then it must hold for `(X, Y)`.

```
path(X, Y) :- arc(X, Y).
path(X, Y) :- arc(X, Z), path(Z, Y).
```

Manipulating lists

Going through a list

1. Define our base case: the empty list.
2. Get the value of the head.
3. Call the same predicate recursively on the tail.

```
printList([]).
printList([H|T]):-
    writeln(H),
    printList(T).
```

Returning a list

1. The base case compares two empty lists.
2. Match the values of both heads.
3. Call the same predicate recursively on the tails.

```
returnList([], []).
returnList([H|T], [H2|T2]):-
    doSomething(H, H2)
    returnList(T, T2).
```

Manipulating lists

Returning a list

1. The base case compares two empty lists.
2. Match the values of both heads.
3. Call the same predicate recursively on the tails.

For instance, we could copy a list.

```
returnList([], []).
returnList([H|T], [H2|T2]):-
    H = H2,
    returnList(T, T2).
```

```
?-returnList([1,2,3], L).
L= [1,2,3].
```

Manipulating lists

Built-in Predicates

member/2

```
?- member(X, [1,2,3]).
X = 1;
```

append/3

```
?- append([1], [2,3], L).
L = [1,2,3].
```

length/2

```
?- length([1,2,3], L).
L = 3.
```

sort/2

```
?- sort([2, 1, 3], L).
L = [1,2,3].
```