

Programming Language Concepts: Lecture 22

Madhavan Mukund

Chennai Mathematical Institute

`madhavan@cmi.ac.in`

`http://www.cmi.ac.in/~madhavan/courses/pl2009`

PLC 2009, Lecture 22, 15 April 2009

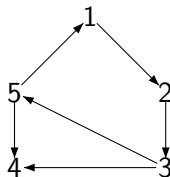
Logic programming

- ▶ Programming with relations
- ▶ Variables
 - ▶ Names starting with a capital letter
 - ▶ `X`, `Y`, `Name`, ...
- ▶ Constants
 - ▶ Names starting with a small letter
 - ▶ `ball`, `node`, `graph`, `a`, `b`,
 - ▶ Uninterpreted — no types like `Char`, `Bool` etc!
 - ▶ Exception: natural numbers, some arithmetic

Defining relations

A Prolog program describes a relation

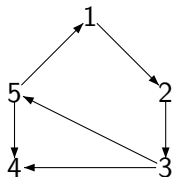
Example: A graph



Defining relations

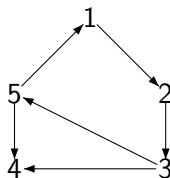
A Prolog program describes a relation

Example: A graph



- ▶ Want to define a relation `path(X,Y)`
- ▶ `path(X,Y)` holds if there is a path from `X` to `Y`

Facts and rules



Represent edge relation using the following facts.

`edge(3,4).`

`edge(5,4).`

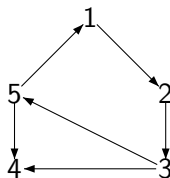
`edge(5,1).`

`edge(1,2).`

`edge(3,5).`

`edge(2,3).`

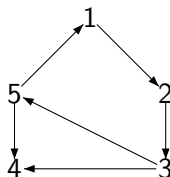
Facts and rules ...



Define **path** using the following **rules**.

```
path(X,Y) :- X = Y.  
path(X,Y) :- edge(X,Z), path(Z,Y).
```

Facts and rules ...



Define `path` using the following `rules`.

```
path(X,Y) :- X = Y.  
path(X,Y) :- edge(X,Z), path(Z,Y).
```

Read the rules read as follows:

Rule 1 For all `X`, $(X,X) \in \text{path}$.

Rule 2 For all `X,Y`, $(X,Y) \in \text{path}$ if there exists `Z` such that $(X,Z) \in \text{edge}$ and $(Z,Y) \in \text{path}$.

Facts and rules ...

```
path(X,Y) :- X = Y.  
path(X,Y) :- edge(X,Z), path(Z,Y).
```

- ▶ Each rule is of the form

Conclusion if Premise₁ and Premise₂ ... and Premise_n

- ▶ if is written :-
 - ▶ and is written ,
 - ▶ This type of logical formula is called a **Horn Clause**
-
- ▶ Quantification of variables
 - ▶ Variables in goal are universally quantified
 - ▶ X, Y above
 - ▶ Variables in premise are existentially quantified
 - ▶ Z above

Computing in Prolog

- ▶ Ask a question (a **query**)

?- path(3,1).

- ▶ Prolog scans facts and rules top-to-bottom

- ▶ **3** cannot be unified with **1**, skip Rule 1.
- ▶ Rule 2 generates two **subgoals**. Find **Z** such that
 - ▶ $(3,Z) \in \text{edge}$ and
 - ▶ $(Z,1) \in \text{path}$.

- ▶ Sub goals are tried depth-first

- ▶ $(3,Z) \in \text{edge}?$
 - ▶ $(3,4) \in \text{edge}$, set $Z = 4$
- ▶ $(4,1) \in \text{path}$? **4** cannot be unified with **1**, two subgoals, new **Z'**
 - ▶ $(4,Z') \in \text{edge}$
 - ▶ $(Z',1) \in \text{path}$
- ▶ Cannot find **Z'** such that $(4,Z') \in \text{edge}$!

Backtracking

- ▶ $(3, Z) \in \text{edge}$?
 - ▶ $\text{edge}(3, 4) \in \text{edge}$, set $Z = 4$
- ▶ $(4, 1) \in \text{path}$? 4 cannot be unified with 1, two subgoals, new Z'
 - ▶ $(4, Z') \in \text{edge}$
 - ▶ $(Z', 1) \in \text{path}$
- ▶ No Z' such that $(4, Z') \in \text{edge}$
- ▶ Backtrack and try another value for Z
 - ▶ $\text{edge}(3, 5) \in \text{edge}$, set $Z = 5$
- ▶ $(5, 1) \in \text{path}$? $(5, 1) \in \text{edge}$, ✓

Backtracking is sensitive to order of facts

- ▶ We had put $\text{edge}(3, 4)$ before $\text{edge}(3, 5)$

Reversing the question

- Consider the question

`?- edge(3,X).`

- Find all `X` such that $(3,X) \in \text{edge}$
- Prolog lists out all satisfying values, one by one

`X=4;`

`X=5;`

`X=2;`

`No.`

Unification and pattern matching

- ▶ A goal of the form $X = Y$ denotes unification.

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

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

Unification and pattern matching

- ▶ A goal of the form $X = Y$ denotes unification.

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

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

- ▶ Can implicitly represent such goals in the head

```
path(X,X).
```

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

Unification and pattern matching

- ▶ A goal of the form $X = Y$ denotes unification.

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

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

- ▶ Can implicitly represent such goals in the head

```
path(X,X).
```

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

- ▶ Unification provides a formal justification for **pattern matching** in rule definitions

Unification and pattern matching

- ▶ A goal of the form $X = Y$ denotes unification.

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

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

- ▶ Can implicitly represent such goals in the head

```
path(X,X).
```

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

- ▶ Unification provides a formal justification for **pattern matching** in rule definitions

- ▶ Unlike Haskell, a repeated variable in the pattern is meaningful
- ▶ In Haskell, we cannot write

```
path (x,x) = True
```

Complex data and terms

Represent arbitrary structures with nested terms

- ▶ A **record** or **struct** of the form

```
personal_data{  
  name : amit  
  date_of_birth{  
    year  : 1980  
    month : 5  
    day   : 30  
  }  
}
```


Complex data and terms

Represent arbitrary structures with nested terms

- ▶ A **record** or **struct** of the form

```
personal_data{  
  name : amit  
  date_of_birth{  
    year  : 1980  
    month : 5  
    day   : 30  
  }  
}
```

- ▶ ... can be represented by a term

```
personal_data(name(amit),  
              date_of_birth(year(1980),month(5),day(30)))
```

Lists

- ▶ Write `[Head | Tail]` for Haskell's `(head:tail)`

Lists

- ▶ Write `[Head | Tail]` for Haskell's `(head:tail)`
 - ▶ `[]` denotes the emptylist

Lists

- ▶ Write `[Head | Tail]` for Haskell's `(head:tail)`
 - ▶ `[]` denotes the emptylist
 - ▶ No types, so lists need not be homogeneous!

Lists

- ▶ Write `[Head | Tail]` for Haskell's `(head:tail)`
 - ▶ `[]` denotes the emptylist
 - ▶ No types, so lists need not be homogeneous!
- ▶ Checking membership in a list

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

Lists

- ▶ Write `[Head | Tail]` for Haskell's `(head:tail)`
 - ▶ `[]` denotes the emptylist
 - ▶ No types, so lists need not be homogeneous!
- ▶ Checking membership in a list

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

- ▶ Use patterns instead of explicit unification

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

Lists

- ▶ Write `[Head | Tail]` for Haskell's `(head:tail)`
 - ▶ `[]` denotes the emptylist
 - ▶ No types, so lists need not be homogeneous!

- ▶ Checking membership in a list

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

- ▶ Use patterns instead of explicit unification

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

- ▶ ...plus anonymous variables.

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

Lists ...

Appending two lists

Lists ...

Appending two lists

- ▶ `append(X,Y,[X|Y]).` will not work

Lists ...

Appending two lists

- ▶ `append(X,Y,[X|Y])` . will not work
 - ▶ `append([1,2],[a,b],Z)` yields `Z = [[1,2],a,b]`

Lists ...

Appending two lists

- ▶ `append(X,Y,[X|Y]).` will not work
 - ▶ `append([1,2],[a,b],Z)` yields `Z = [[1,2],a,b]`
- ▶ Inductive definition, like Haskell

```
append(Xs, Ys, Zs) :- Xs = [], Zs = Ys.  
append(Xs, Ys, Zs) :- Xs = [H | Ts], Zs = [H | Us],  
                        append(Ts, Ys, Us).
```

Lists ...

Appending two lists

- ▶ `append(X,Y,[X|Y]).` will not work
 - ▶ `append([1,2],[a,b],Z)` yields `Z = [[1,2],a,b]`
- ▶ Inductive definition, like Haskell

```
append(Xs, Ys, Zs) :- Xs = [], Zs = Ys.  
append(Xs, Ys, Zs) :- Xs = [H | Ts], Zs = [H | Us],  
                        append(Ts, Ys, Us).
```

- ▶ Again, eliminate explicit unification

```
append([], Ys, Ys).  
append([X | Xs], Ys, [X | Zs]) :- append(Xs, Ys, Zs).
```

Reversing the computation

```
?- append(Xs, Ys, [mon, wed, fri]).
```

Reversing the computation

```
?- append(Xs, Ys, [mon, wed, fri]).
```

All possible ways to split the list

Reversing the computation

```
?- append(Xs, Ys, [mon, wed, fri]).
```

All possible ways to split the list

```
Xs = []  
Ys = [mon, wed, fri] ;
```

```
Xs = [mon]  
Ys = [wed, fri] ;
```

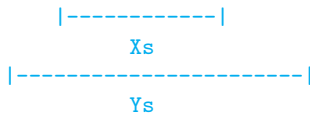
```
Xs = [mon, wed]  
Ys = [fri] ;
```

```
Xs = [mon, wed, fri]  
Ys = [] ;
```

```
no
```

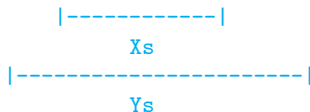
Reversing the computation ...

- Want to define a relation `sublist(Xs,Ys)`

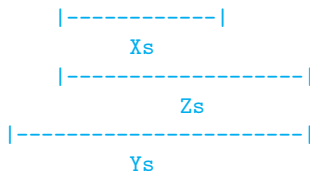


Reversing the computation ...

- ▶ Want to define a relation `sublist(Xs,Ys)`

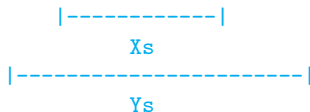


- ▶ Add an intermediate list `Zs`

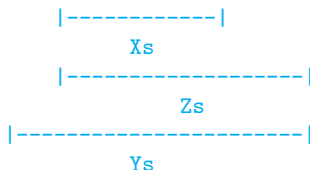


Reversing the computation ...

- ▶ Want to define a relation `sublist(Xs,Ys)`



- ▶ Add an intermediate list `Zs`

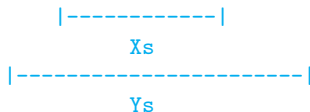


- ▶ Yields the rule

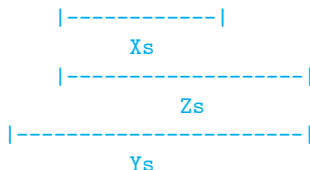
```
sublist(Xs, Ys) :- append(_, Zs, Ys), append(Xs, _, Zs).
```

Reversing the computation ...

- ▶ Want to define a relation `sublist(Xs,Ys)`



- ▶ Add an intermediate list `Zs`



- ▶ Yields the rule

```
sublist(Xs, Ys) :- append(_, Zs, Ys), append(Xs, _, Zs).
```

- ▶ Why won't the following work?

```
sublist(Xs, Ys) :- append(Xs, _, Zs), append(_, Zs, Ys).
```

Reversing the computation . . .

Type inference for simply typed lambda calculus

$$x \in Var \mid \lambda x.M \mid MN$$

Reversing the computation ...

Type inference for simply typed lambda calculus

$$x \in \text{Var} \mid \lambda x.M \mid MN$$

- ▶ Inference rules to derive type judgments of the form $A \vdash M : s$
 - ▶ A is list $\{x_i : t_i\}$ of type “assumptions” for variables
 - ▶ Under the assumptions in A the expression M has type s .

Reversing the computation ...

Type inference for simply typed lambda calculus

$$x \in \text{Var} \mid \lambda x.M \mid MN$$

- ▶ Inference rules to derive type judgments of the form $A \vdash M : s$
 - ▶ A is list $\{x_i : t_i\}$ of type “assumptions” for variables
 - ▶ Under the assumptions in A the expression M has type s .

$$\frac{x : t \in A}{A \vdash x : t}$$

$$\frac{A \vdash M : s \rightarrow t, \quad A \vdash N : s}{A \vdash (MN) : t}$$

$$\frac{A + x : s \vdash M : t}{A \vdash (\lambda x.M) : s \rightarrow t}$$

Reversing the computation ...

- ▶ Encoding λ -calculus and types in Prolog
 - ▶ `var(x)` for variable x (Note: x is a constant!)
 - ▶ `lambda(x,m)` for $\lambda x.M$
 - ▶ `apply(m,n)` for MN
 - ▶ `arrow(s,t)` for $s \rightarrow t$

Reversing the computation ...

► Encoding λ -calculus and types in Prolog

- `var(x)` for variable x (Note: x is a constant!)
- `lambda(x,m)` for $\lambda x.M$
- `apply(m,n)` for MN
- `arrow(s,t)` for $s \rightarrow t$

► Type inference in Prolog

```
% type(A, S, T):- lambda term S has type T in the environment A.  
type(A, var(X), T):- member([X, T], A).  
type(A, apply(M, N), T):- type(A, M, arrow(S,T), type(A, N, S).  
type(A, lambda(X, M), (arrow(S,T)):- type([X, S] | A), M, T).
```


Reversing the computation ...

► Encoding λ -calculus and types in Prolog

- `var(x)` for variable x (Note: x is a constant!)
- `lambda(x,m)` for $\lambda x.M$
- `apply(m,n)` for MN
- `arrow(s,t)` for $s \rightarrow t$

► Type inference in Prolog

```
% type(A, S, T):- lambda term S has type T in the environment A.  
type(A, var(X), T):- member([X, T], A).  
type(A, apply(M, N), T):- type(A, M, arrow(S,T), type(A, N, S).  
type(A, lambda(X, M), (arrow(S,T)):- type([X, S] | A), M, T).
```

► `?- type([], t, T).` asks if term t is typable.

```
?- type([], lambda(x, apply(var(x), var(x))), T).
```

Reversing the computation ...

- ▶ Encoding λ -calculus and types in Prolog
 - ▶ `var(x)` for variable x (Note: x is a constant!)
 - ▶ `lambda(x,m)` for $\lambda x.M$
 - ▶ `apply(m,n)` for MN
 - ▶ `arrow(s,t)` for $s \rightarrow t$

- ▶ Type inference in Prolog

```
% type(A, S, T):- lambda term S has type T in the environment A.  
type(A, var(X), T):- member([X, T], A).  
type(A, apply(M, N), T):- type(A, M, arrow(S,T), type(A, N, S).  
type(A, lambda(X, M), (arrow(S,T)):- type([X, S] | A), M, T).
```

- ▶ `?- type([],t,T).` asks if term t is typable.
 - `?- type([], lambda(x, apply(var(x), var(x))), T).`
 - `type([x, S], apply(var(x), var(x)), T)`

Reversing the computation ...

► Encoding λ -calculus and types in Prolog

- `var(x)` for variable x (Note: x is a constant!)
- `lambda(x,m)` for $\lambda x.M$
- `apply(m,n)` for MN
- `arrow(s,t)` for $s \rightarrow t$

► Type inference in Prolog

```
% type(A, S, T):- lambda term S has type T in the environment A.  
type(A, var(X), T):- member([X, T], A).  
type(A, apply(M, N), T):- type(A, M, arrow(S,T), type(A, N, S).  
type(A, lambda(X, M), (arrow(S,T)):- type([X, S] | A), M, T).
```

► `?- type([], t, T).` asks if term t is typable.

```
?- type([], lambda(x, apply(var(x), var(x))), T).  
type([x, S], apply(var(x), var(x)), T)  
type([x, S], var(x), arrow(S,T)).
```

Reversing the computation ...

► Encoding λ -calculus and types in Prolog

- `var(x)` for variable x (Note: x is a constant!)
- `lambda(x,m)` for $\lambda x.M$
- `apply(m,n)` for MN
- `arrow(s,t)` for $s \rightarrow t$

► Type inference in Prolog

```
% type(A, S, T):- lambda term S has type T in the environment A.  
type(A, var(X), T):- member([X, T], A).  
type(A, apply(M, N), T):- type(A, M, arrow(S,T), type(A, N, S).  
type(A, lambda(X, M), (arrow(S,T)):- type([[X, S] | A], M, T).
```

► `?- type([], t, T).` asks if term t is typable.

```
?- type([], lambda(x, apply(var(x), var(x))), T).  
type([[x, S]], apply(var(x), var(x)), T)  
type([[x, S]], var(x), arrow(S,T)).  
member([x, arrow(S,T)], [[x, S]])
```

Reversing the computation ...

► Encoding λ -calculus and types in Prolog

- `var(x)` for variable x (Note: x is a constant!)
- `lambda(x,m)` for $\lambda x.M$
- `apply(m,n)` for MN
- `arrow(s,t)` for $s \rightarrow t$

► Type inference in Prolog

```
% type(A, S, T):- lambda term S has type T in the environment A.  
type(A, var(X), T):- member([X, T], A).  
type(A, apply(M, N), T):- type(A, M, arrow(S,T), type(A, N, S).  
type(A, lambda(X, M), (arrow(S,T)):- type([X, S] | A), M, T).
```

► `?- type([], t, T).` asks if term t is typable.

```
?- type([], lambda(x, apply(var(x), var(x))), T).  
type([x, S], apply(var(x), var(x)), T)  
type([x, S], var(x), arrow(S,T)).  
member([x, arrow(S,T)], [[x, S]])
```

► Unification fails

Example: special sequence ...

Arrange three 1s, three 2s, ..., three 9s in sequence so that for all $i \in [1..9]$ there are exactly i numbers between successive occurrences of i

Example: special sequence . . .

Arrange three 1s, three 2s, ..., three 9s in sequence so that for all $i \in [1..9]$ there are exactly i numbers between successive occurrences of i

1, 9, 1, 2, 1, 8, 2, 4, 6, 2, 7, 9, 4, 5, 8, 6, 3, 4, 7, 5, 3, 9, 6, 8, 3, 5, 7.

Example: special sequence . . .

Arrange three 1s, three 2s, ..., three 9s in sequence so that for all $i \in [1..9]$ there are exactly i numbers between successive occurrences of i

1, 9, 1, 2, 1, 8, 2, 4, 6, 2, 7, 9, 4, 5, 8, 6, 3, 4, 7, 5, 3, 9, 6, 8, 3, 5, 7.

`% sequence(Xs) :- Xs is a list of 27 variables.`

`sequence([_,_]).`

Example: special sequence ...

Arrange three 1s, three 2s, ..., three 9s in sequence so that for all $i \in [1..9]$ there are exactly i numbers between successive occurrences of i

1, 9, 1, 2, 1, 8, 2, 4, 6, 2, 7, 9, 4, 5, 8, 6, 3, 4, 7, 5, 3, 9, 6, 8, 3, 5, 7.

```
% sequence(Xs) :- Xs is a list of 27 variables.
```

```
sequence([_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_]).
```

```
solution(Ss) :-
```

```
sequence(Ss),
```

```
sublist([9,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_], Ss),
```

```
sublist([8,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_], Ss),
```

```
sublist([7,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_], Ss),
```

```
sublist([6,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_], Ss),
```

```
sublist([5,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_], Ss),
```

```
sublist([4,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_], Ss),
```

```
sublist([3,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_], Ss),
```

```
sublist([2,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_], Ss),
```

```
sublist([1,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_], Ss).
```

Computing length of a list

```
length([],0).  
length([H,T],N) :- length(T,M), N = M+1.
```

Arithmetic

Computing length of a list

```
length([],0).  
length([H,T],N) :- length(T,M), N = M+1.
```

What does the following query yield?

```
?- length([1,2,3,4],N).
```

Computing length of a list

```
length([],0).  
length([H,T],N) :- length(T,M), N = M+1.
```

What does the following query yield?

```
?- length([1,2,3,4],N).
```

```
N=0+1+1+1+1
```

► $X = Y$ is unification

Arithmetic

Computing length of a list

```
length([],0).  
length([H,T],N) :- length(T,M), N = M+1.
```

What does the following query yield?

```
?- length([1,2,3,4],N).
```

```
N=0+1+1+1+1
```

- ▶ $X = Y$ is unification
- ▶ $X \text{ is } Y$ captures arithmetic equality

Arithmetic

Computing length of a list

```
length([],0).  
length([H,T],N) :- length(T,M), N = M+1.
```

What does the following query yield?

```
?- length([1,2,3,4],N).
```

```
N=0+1+1+1+1
```

- ▶ $X = Y$ is unification
- ▶ $X \text{ is } Y$ captures arithmetic equality

```
length([],0).  
length([H,T],N) :- length(T,M), N is M+1.
```

Arithmetic ...

Another approach

```
length(L,N) :- auxlength(L,0,N).  
auxlength([],N,N).  
auxlength([H|T],M,N) :- auxlength(T,M1,N), M1 is M+1.
```

Arithmetic ...

Another approach

```
length(L,N) :- auxlength(L,0,N).  
auxlength([],N,N).  
auxlength([H|T],M,N) :- auxlength(T,M1,N), M1 is M+1.
```

?- length([0,1,2],N) generates goals

```
auxlength([0,1,2],0,N)
```


Arithmetic ...

Another approach

```
length(L,N) :- auxlength(L,0,N).  
auxlength([],N,N).  
auxlength([H|T],M,N) :- auxlength(T,M1,N), M1 is M+1.
```

?- length([0,1,2],N) generates goals

```
auxlength([0,1,2],0,N)
```

```
auxlength([1,2],1,N)
```

Arithmetic ...

Another approach

```
length(L,N) :- auxlength(L,0,N).  
auxlength([],N,N).  
auxlength([H|T],M,N) :- auxlength(T,M1,N), M1 is M+1.
```

?- length([0,1,2],N) generates goals

```
auxlength([0,1,2],0,N)
```

```
auxlength([1,2],1,N)
```

```
auxlength([2],2,N)
```

Arithmetic ...

Another approach

```
length(L,N) :- auxlength(L,0,N).  
auxlength([],N,N).  
auxlength([H|T],M,N) :- auxlength(T,M1,N), M1 is M+1.
```

?- length([0,1,2],N) generates goals

```
auxlength([0,1,2],0,N)
```

```
auxlength([1,2],1,N)
```

```
auxlength([2],2,N)
```

```
auxlength([],3,N)
```

Another approach

```
length(L,N) :- auxlength(L,0,N).  
auxlength([],N,N).  
auxlength([H|T],M,N) :- auxlength(T,M1,N), M1 is M+1.
```

?- length([0,1,2],N) generates goals

```
auxlength([0,1,2],0,N)
```

```
auxlength([1,2],1,N)
```

```
auxlength([2],2,N)
```

```
auxlength([],3,N)
```

```
auxlength([],3,3)
```

Arithmetic ...

Another approach

```
length(L,N) :- auxlength(L,0,N).  
auxlength([],N,N).  
auxlength([H|T],M,N) :- auxlength(T,M1,N), M1 is M+1.
```

?- length([0,1,2],N) generates goals

```
auxlength([0,1,2],0,N)
```

```
auxlength([1,2],1,N)
```

```
auxlength([2],2,N)
```

```
auxlength([],3,N)
```

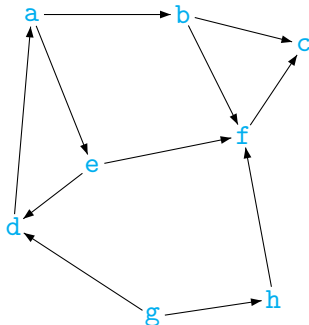
```
auxlength([],3,3)
```

Second argument to `auxlength` accumulates answer.

Coping with circular definitions

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

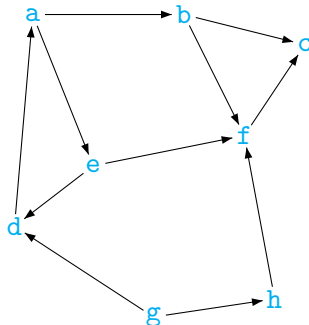
```
path(X,X).  
path(X,Y) :- edge(X,Z), path(Z,Y).
```



Coping with circular definitions

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

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



What does `?- path(a,b)` compute?

Coping with circularity . . .

Instead

```
path(X,X,T).  
path(X,Y,T) :- a(X,Z), legal(Z,T), path(Z,Y,[Z|T]).  
  
legal(Z, []).  
legal(Z,[H|T]) :- Z\==H, legal(Z,T).
```

- ▶ `path(X,Y,T)` succeeds if there is a path from `X` to `Y` that does not visit any nodes in `T`
- ▶ `T` is an accumulator