KRR5: Logic programming: PROLOG

Yannick Pencolé
Yannick.Pencole@anu.edu.au

17 Mar 2005

What is Prolog?

Prolog is a programming language

Declarative programming

The programmer declares a knowledge base (KB) and asks a question. Prolog does the rest.

How does it work

The KB declared in Prolog is based on Horn's Clauses. To answer the question, Prolog uses Backward Chaining.

Constants and Variables

Definition

A Constant is

1. Number: 12,3.5
2. Atoms:
   - any string that begins with a small letter
   - any string between " "
   - empty lists symbol []
3. Variables:
   - any string that begins with a capital letter
   - any string that begins with _
   - wildcard pattern _

Three kinds of knowledge

Definition

A Fact is a predicate.
`p(...).` (i.e. $p(...)$).
A fact can be seen as the Head of a Horn's clause.

Definition

A Rule is a complete Horn clause:
`p(..)  :- q(..), ..., r(..).`
(i.e. $q(...) \wedge \cdots \wedge r(...) \Rightarrow p(...)$)

Definition

A Query is a set of predicates:
`s(..),..,t(..).`
A query can be seen as the Body of a Horn's clause.

My first program

KB

Here is the KB to program:

$father(charlie, david)$

$father(henri, charlie)$

$father(X, Z) \wedge father(Z, Y) \Rightarrow grandfather(X, Y)$

In Prolog

```
father(charlie,david).
father(henri,charlie).
grandfather(X,Y) :- father(X,Z) , father(Z,Y).
```

My first program

```
pencole@chef$ swiprolog
The binary name `swiprolog' is deprecated in favour of `swipl'.
Please use the new name instead.

Welcome to SWI-Prolog (Multi-threaded, Version 5.2.13)
Copyright (c) 1990-2003 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

?- [father].
% father compiled 0.00 sec, 1,148 bytes

Yes
```

My first program

```
Yes
?- father(charlie,david).

Yes
?- father(charlie,henri).

No
?- father(X,Y).

X = charlie
Y = david ;

X = henri
Y = charlie ;

No
?-
```

My first program

```
?- grandfather(x,y).

No
?- grandfather(X,Y).

X = henri
Y = david ;

No
?- grandfather(henri,X).

X = david ;

No
```

Order of the answers

```
?- listing.


mother(sophie, charlie).
mother(anne, david).


parents(A, B, C) :-
        father(B, A),
        mother(C, A).

%   Foreign: rl_read_init_file/1


father(charlie, david).
father(henri, charlie).
father(david, luc).
```

```
?- parents(X,Y,Z).

X = david
Y = charlie
Z = anne ;

X = charlie
Y = henri
Z = sophie ;

No
?-
```

Prolog “reads” clauses from the top to the bottom and “explores” from the left to the right.

Functions

Function

In prolog, we can also declare a function of FOL. A function has not result, it is just a functional relation.

Example

John'wife: `wife(john)`

Such a term is always included in a predicate in prolog: `name(wife(john),marie).`

Be careful about the confusion between the function `wife(john)` which represents the wife of John and the predicate `wife(john)` which says that John is a wife!

Arithmetic

How to play with arithmetic

- Comparisons: `>`, `<`, `>=`, `=<`, `=:=`, `=\=`
- Assignation: `is`
  - `?- X is 3+2.`
  - X=5
- Predefined functions: $-$, $+$, $*$, $/$, ^, `mod`, `abs`, `min`, `max`, `sign`, `random`, `sqrt`, `sin`, `cos`, `tan`, `log`, `exp`...

Recursive programming

Depth-first search

Depth-first search from a start state X:

```
dfs(X) :- goal(X).
dfs(X) :- successor(X,S) dfs(S).
```

Factorial

Factorial:

```
fact(A,B) :- fact(A,1,B).
fact(A,B,C) :- A > 1, D is B*A, E is A-1,
fact(E,D,C).
fact(1,A,A).
```

Redundant inference and infinite loops

Find a path: version 1

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


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

Find a path: version 2

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


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

What is the difference between version 1 and version 2?

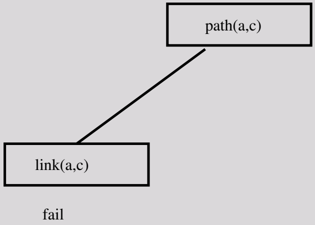
Proof tree: version 1

Example

path(a,c)

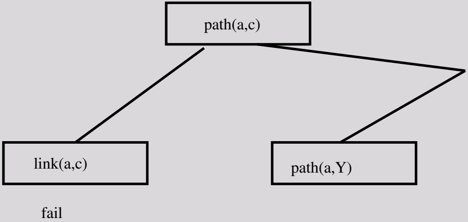
Proof tree: version 1

Example



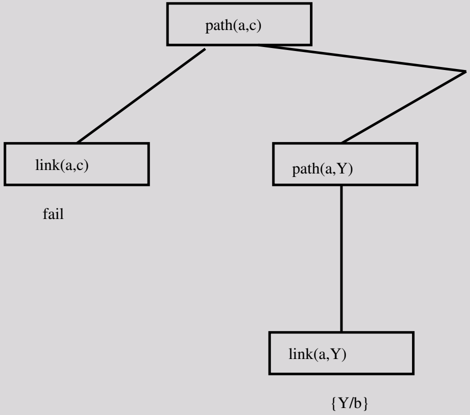
Proof tree: version 1

Example



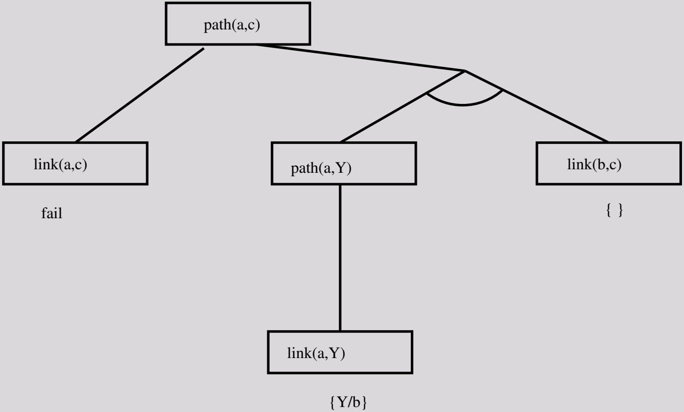
Proof tree: version 1

Example



Proof tree: version 1

Example



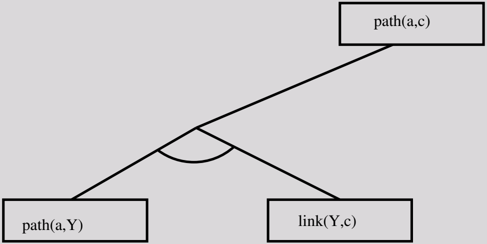
Proof tree: version 2

Example

path(a,c)

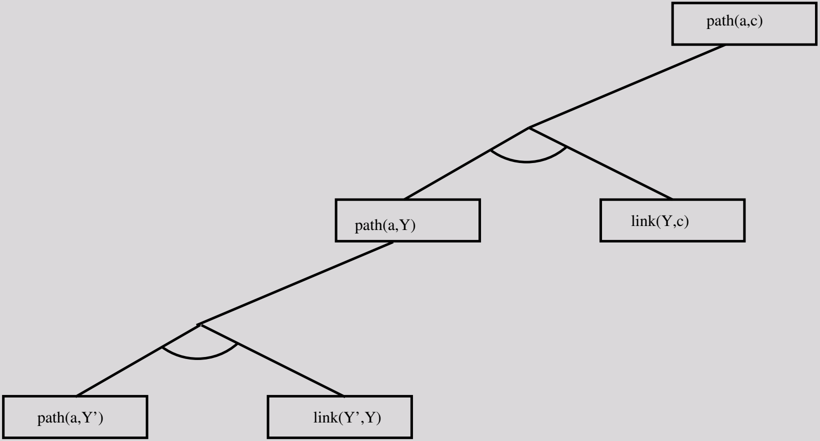
Proof tree: version 2

Example



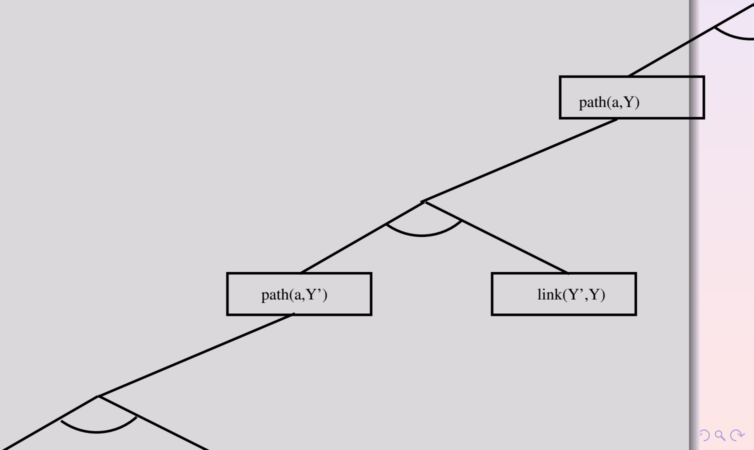
Proof tree: version 2

Example



Proof tree: version 2

Example



Term comparison and unification

Comparison

- T1 == T2 succeeds if T1 and T2 are identical (equality of FOL)
- T1 \ == T2 succeeds if T1 and T2 are not identical

Unification

- T1 = T2 is the Unification of T1 and T2 (i.e. UNIFY(T1,T2) is called)
- T1 \= T2 succeeds if (i.e. UNIFY(T1,T2) has no solution)

Lists

Empty list

The empty list is represented by: [ ]

General case

A list has a Head and a Tail: `[ Head | Tail ]`

Example

The list a, b, c is denoted in Prolog: [a |[b |[c |[ ]]]]

Lists: examples

Example

1. `[ X | L ] = [ a, b, c ]` $\longrightarrow$

Lists: examples

Example

1. `[ X | L ] = [ a, b, c ]` $\longrightarrow$ X = a, L = [b,c]
2. `[ X | L ] = [a]` $\longrightarrow$

Lists: examples

Example

1. [X | L] = [a, b, c] → X = a, L = [b,c]
2. [X | L] = [a] → X = a, L =[ ]
3. [X | L] = [ ] →

Lists: examples

Example

1. [X | L] = [a, b, c] → X = a, L = [b,c]
2. [X | L] = [a] → X = a, L =[ ]
3. [X | L] = [ ] → fail
4. [X , Y] = [a, b, c] →

Lists: examples

Example

1. [X | L] = [a, b, c] → X = a, L = [b,c]
2. [X | L] = [a] → X = a, L =[ ]
3. [X | L] = [ ] → fail
4. [X , Y] = [a,b,c] → fail
5. [X, Y | L] = [a,b,c] →

Lists: examples

Example

1. [X | L] = [a, b, c] $\rightarrow$ X = a, L = [b,c]
2. [X | L] = [a] $\rightarrow$ X = a, L =[ ]
3. [X | L] = [ ] $\rightarrow$ fail
4. [X , Y] = [a,b,c] $\rightarrow$ fail
5. [X, Y | L] = [a,b,c] $\rightarrow$ X = a, Y = b, L = [c]
6. [X | L] = [X,Y | L2] $\rightarrow$

Lists: examples

Example

1. [X | L] = [a, b, c] → X = a, L = [b,c]
2. [X | L] = [a] → X = a, L =[ ]
3. [X | L] = [ ] → fail
4. [X , Y] = [a,b,c] → fail
5. [X, Y | L] = [a,b,c] → X = a, Y = b, L = [c]
6. [X | L] = [X,Y | L2] → L = [Y|L2]

Sum of elements

Example

```
sumElements([  ],0).
sumElements([ A | B ], C) :-
                      sumElements(B,D),
                      C is D+A.
```

Query:

```
?- sumElements([1,2,3,5],N).
N = 11 ;
No
```

Wildcard pattern: ith

Example

```
ith([ X | _],1,X).
ith([ _ | L ], R, Y) :-
Rm1 is R-1, ith(L,Rm1,Y).
```

Query:

```
?- ith([a,b,c,d],2,N).
N = b ;
No
```

Predicate append

Append

append is a predefined predicate to append lists

Example

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

L = [ a,b,c,d ]
```

How to find the last element of a list?

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

X= d
```

How to create sub-lists from lists?

```
?- append(L2,L3,[ b,c,a,d,e ]), append(L1, [ a ], L2).
L2 = [b,c,a]
L3 = [d,e]
L1 = [ b , c ]
```

Sort

Example

Given two sorted lists L1, L2 the predicat `merge` merges the lists to build a new sorted list:

```
merge([ ], L, L).

merge( L, [ ], L).

merge( [X|L1], [Y | L2 ], [ X | L ]) :- X=<Y, merge(L1, [ Y |
L2 ], L).

merge( [X|L1], [Y | L2 ], [ Y | L ]) :- X>Y, merge([X | L1],L2,
L).
```

Negation as failure

not

Prolog allows a “kind of” negation called negation as failure. If Prolog is not able to prove P then $notP$ is proved!

Example

```
alive(X) :- not dead(X).
```

means: “Everyone is alive if not provably dead”.

Be careful the `not` is NOT the $\neg$ of FOL. If we are not able to prove $dead(X)$, we cannot say anything about $\neg dead(X)$

The cut

Normal behaviour

Imagine the following rules:

```
R1:  belong(X, [X | _ ]).
R2:  belong(X, [_| L ]) :- belong(X,L).
```

and the query

```
belong(X,[a,b,c]).
```

Solution: X = a, X = b, X = c

Proof tree: at each node of the tree, we choose R1 and THEN R2.

Cut

```
R1:  belong(X, [X | _ ]):- !   .
R2:  belong(X, [_| L ]) :- belong(X,L).
```

and the query

```
belong(X,[a,b,c]).
```

Solution: X = a

Proof tree: We cut the complete proof tree. At each node of the tree, we choose only the rule that are before “!” (i.e. R1)

Last example :-)

Teaching at the ANU

```
person(yannick).
study(people,anu).
have(people,m1).
goodlectureslogic(m1).
students(X) :- study(X,anu).
gives(yannick,X,people) :- goodlectureslogic(X) ,
have(people,X).
goodteacher(X) :- person(X), gives(X,Y,Z),
goodlecturesfol(Y) , students(Z).
goodlecturesfol(X) :- goodlectureslogic(X).
```

Query:

```
?- goodteacher(Yannick).
Yes
?- goodteacher(Z).
Z = Yannick
```