4: Logic paradigm

Programming Languages, Technologies and Paradigms

Escuela Técnica Superior de Ingeniería Informática etsinf

Summary

1 Introduction

2 Syntax of logic programs

3 Computational model of logic programming

4 Some practical issues

Objectives

- Understanding the logic programming computational model: inversion of definitions, logic variables, nondeterminism, etc.
- Understanding bidirectional parameter passing and its implementation by means of the unification mechanism.
- Understanding the resolution principle and the different computation rules and search strategies that can be applied.
- Solving small problems using the logic paradigm.

Example

- Monty Python's Knights of the round table (Monty Python and the Holy Grail) (1975)
  http://www.youtube.com/watch?v=yp_l5ntikaU

Example

- Prolog solution

```
witch(X) :- burns(X), woman(X) .
burns(X) :- wooden(X) .

wooden(X) :- floats(X) .
wooden(wooden_bridge) .
stone(stone_bridge) .

floats(bread) .
floats(apple) .
floats(green_sauce) .
floats(duck) .

floats(X) :- same_weight(duck,X) .

/* Observation */
same_weight(duck,woman-on-the-scene) .
woman(woman-on-the-scene) .
```

Terminal — swipl — 80×24

```
Last login: Mon Dec 10 16:08:00 on ttys000
millenium:~ mramirez$ cd /Users/mramirez/Documents/DOCENCIA/LTP/TEORIA/Tema\ 4/2
012-13
millenium:2012-13 mramirez$ swipl
% library(swi_hooks) compiled into pce_swi_hooks 0.00 sec, 2,284 bytes
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 5.10.4)
Copyright (c) 1990-2011 University of Amsterdam, VU 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).

?- [bruja].
% bruja compiled 0.00 sec, 2,248 bytes
true.

?- bruja(Quien).
Quien = la_mujer_de_la_escena .

?-
```

Some distinctive features

- Use of logic as a programming language
- Logical variables
  - Answer extraction
  - Invertible definitions
  - Non-determinism

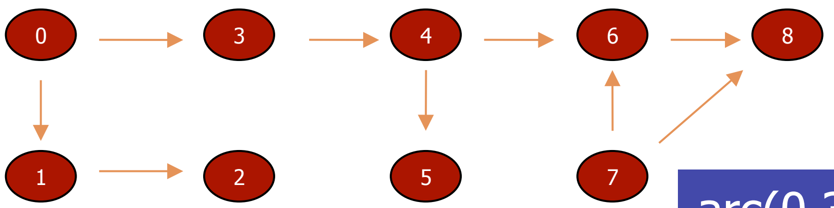
Use of logic as a programming language

- Logic programming implements the revolutionary idea of using ***logic as a programming language*.**
- Writing a logic program consists of expressing a relation (or set of relations) by using a logic notation based on the **predicate logic.**
- The essential idea of the logic paradigm is that of *COMPUTATION as DEDUCTION. This is in contrast to more standard notions like COMPUTATION as CALCULATION.*

Use of logic as a programming language

PROGRAM

Express the <u>knowledge of the problem</u> ⇒
WRITE LOGIC FORMULAS



arc(0,3). arc(3,4). arc(4,6). arc(6,8) .

arc(0,1). arc(1,2). arc(4,3).

arc(7,6). arc(7,8).

connected(X,Y) ← arc(X,Y).
connected(X,Y) ← arc(X,Z) ∧ connected(Z,Y).

Use of logic as a programming language

PROGRAM
Express the knowledge of the problem ⇒
WRITE LOGIC FORMULAS

PROGRAM EXECUTION
Express the problem to solve ⇒
GOAL FORMULA; DEDUCTION USING QUERIES

arc(0,3) .	arc(3,4) .	arc(4,6) .	arc(6,8) .
arc(0,1) .	arc(1,2) .	arc(4,3) .	
arc(7,6) .	arc(7,8) .		

connected(X,Y) ← arc(X,Y) .
connected(X,Y) ← arc(X,Z) ∧ connected(Z,Y) .

Are 0 and 8 connected?
Are 4 and 7 connected?

connected(0,8)?
yes
connected(4,7)?
no

Logical variables

- Program variables are unknowns (mathematical variables, like in an equation).
- Implicitly, logic formulas in programs are universally quantified.

connected(X,Y) ← arc(X,Z) ∧ connected(Z,Y)

$$\Downarrow$$

$\forall X,Y,Z(\text{connected}(X,Y) \leftarrow \text{arc}(X,Z) \wedge \text{connected}(Z,Y))$

$$\Downarrow$$

$\forall X,Y(\text{connected}(X,Y) \leftarrow \exists Z(\text{arc}(X,Z) \wedge \text{connected}(Z,Y)))$

Answer extraction

- Variables in queries are existentially quantified.

```
?-connected(X,Y) ?
X=0
Y=1
```

READ: Are there X and Y such that
connected(X,Y)
holds (w.r.t. the logic program)?

⇓

The mechanism that is used to prove the goal is constructive:
When succeeding, values for the unknowns X and Y are given

This is the outcome or ***answer*** to the query

Invertibility

- The predicate arguments can be both input or output arguments.

```
member(H,[H|L]) .
member(H,[X|L]) :- member(H,L) .
```

- Check for membership: member(2,[1,2])
- Return the elements of a list: member(X,[1,2])
- Generate the lists containing an element: member(1,L)

Non-determinism

- A query can deliver several answers that the interpreter obtains by exhaustively exploring the computation space.

```
?- member(X, [1,2,a]) .
```

Answer 1: X=1

Answer 2: X=2

Answer 3: X=a

2. Syntax of logic programs: Terms

- Data in logic programs are called terms and can be either:
  - Variables:
    - Prolog: variable identifiers begin with a **capital** letter. Anonymous variables are represented using “_”
    - Example: X, Y, SquareArea, Result
  - Constants:
    - Prolog: numeric and symbolic (with identifiers beginning with a **lower case** letter or written in quotes)
    - Example: 42, ‘a’, peter, ‘Peter’, ‘Hello World’, …
  - Structured ***data*** $f(t_1,\ldots,t_n)$ where f is a function symbol and $t_1,\ldots,t_n$ are terms
    - Prolog: f is a data constructor beginning with a lower case letter
    - Example: hour(h,m,s), name(‘Peter’)

Lists (Prolog notation)

- Lists are a particular kind of terms built out from:
  - the empty list: []
  - the list constructor symbol [_|_]

- Examples: [1|[2|[]]] (shortly: [1,2])
  [1|[2|X]] (equivalent to [1,2|X])
  [1|2] ERROR

- Similar to Haskell's [] and (_:_)

2. Syntax of logic programs: Atoms

- Atoms are expressions $p(t_1,\ldots,t_n)$ where
  - p is a predicate symbol of arity n (often written p/n), i.e., a sequence of characters beginning by a lower case letter
  - $t_1,\ldots,t_n$ are terms
- Atoms express properties or relations (p) concerning data represented by terms $t_1,\ldots,t_n$
- Example: arc(1,2)

Syntax of logic programs: Prolog programs

- A logic program is a set of **sentences/declarations** that can be of two types: facts or rules.

  - FACTS: single **atoms** followed by a dot **A .**

    Example: arc(0,1) .

    Note that ‘,’ is ‘∧’

  - RULES: having the form **A :- B_1,…, B_n .** where n>0

    Example: connected(X,Y) :- arc(X,Z), connected(Z,Y).

    head body

where A and each B_i are atoms.

NOTE: facts can be seen as rules with an empty body, as follows:

A :- true .

Syntax of logic programs: goals

- The 'call' that serves to execute a logic program is called the **goal** and is written as a clause without head, i.e.,

  ?- B_1,…, B_n with n>0

  Example: ?- connected(X,Y)

Note that, in sharp contrast to FP, terms are ***not evaluated*** because goals rather consist of atoms

- A clause without head nor body is called an **empty clause** and is represented as ?-

The empty clause witnesses that the computation was successfully finished.

From Haskell to Prolog

- Both Haskell and Prolog are **rule-based languages.** From a syntactic point of view, the main differences are that, in Prolog:
  - There is no function (only procedures)
  - Calls to such procedures cannot be nested
- Example:

```
fibonacci(0) = 0                                        /* Haskell */
fibonacci(1) = 1
fibonacci(n) | n>1 = fibonacci(n-1) + fibonacci(n-2)

fibonacci(0,0).                                         /* Prolog */
fibonacci(1,1).
fibonacci(N,M) :- N>      N-1, N2 is N-2, fibonacci(N1,F1), fibonacci(N2,F2), M is F1+F2.
```

Functions become procedures with an extra parameter which is used to return the result

From Haskell to Prolog

- Both Haskell and Prolog are **rule-based languages.** From a syntactic point of view, the main differences are that, in Prolog:
  - There is no function (only procedures)
  - Calls to such procedures cannot be nested
- Example:

```
fibonacci(0) = 0                                        /* Haskell */
fibonacci(1) = 1
fibonacci(n) | n>1 = fibonacci(n-1) + fibonacci(n-2)
```

```
fibonacci(0,0).                                         /* Prolog */
fibonacci(1,1).
fibonacci(N,M) :- N>1, N1 is N-1, N2 is N-2, fibonacci(N1,F1), fibonacci(N2,F2), M is F1+F2.
```

The guard is just another relation

From Haskell to Prolog

- Both Haskell and Prolog are **rule-based languages.** From a syntactic point of view, the main differences are that, in Prolog:
  - There is no function (only procedures)
  - Calls to such procedures cannot be nested
- Example:

```
fibonacci(0) = 0                                        /* Haskell */
fibonacci(1) = 1
fibonacci(n) | n>1 = fibonacci(n-1) + fibonacci(n-2)

fibonacci(0,0).                                         /* Prolog */
fibonacci(1,1).
fibonacci(N,M) :- N>1, N1 is N-1, N2 is N-2, fibonacci(N1,F1), fibonacci(N2,F2), M is F1+F2.
```

Calls to subtraction and fibonacci cannot be nested!
(essentially "X is E" evaluates expression E; its value is then bound to variable X)

Examples

Length of a list:

- Haskell code:

```
length [ ] = 0
length (x:xs) = length xs + 1
```

- Prolog code:

```
length ([], 0).
length ([_|T], N) :-  length(T, N1),
                      N is N1+1.
```

Examples

List concatenation

- Haskell:

```
[] ++ y = y
(x:xs) ++ y = x : (xs ++ y)
```

- Prolog:

```
append([], Y, Y) .
append([X|R], Y, Z) :- append(R, Y, RY), Z = [X|RY] .
```

Examples

List concatenation

- Haskell:

  [] ++ y = y

  (x:xs) ++ y = x : (xs ++ y)

- Prolog:

```
append([], Y, Y) .
append([X|R], Y, Z) :- append(R, Y, RY), Z = [X|RY] .
```

or better:

```
append([], Y, Y) .
append([X|R], Y, [X|RY]) :- append(R, Y, RY) .
```

The parameter that represents the outcome of the function is replaced by the returned expression

Examples

Last element of a list

- Haskell:

```
last [x] = x
last (x:y:xs) = last (y:xs)
```

- In Prolog:

```
last([X],X) .
last([X,Y|XS],Z):- last([Y|XS],Z) .
```

But also, using append, we have:

```
last(XS,Z):- append(YS,[Z],XS) .
```

Exercise

- Specify the relationship "ancestor" by using a logic program

X is an ancestor of Y if

X is the father of Y

X is the mother of Y

X is the father of Z and Z is an ancestor of Y

X is the mother of Z and Z is an ancestor of Y

Procedural interpretation

PROGRAM CLAUSE ≡ *DEFINITION OF A METHOD OR SUBPROGRAM*

m(t1,…,tn) :- A_1, …, A_n

m(t1,…,tn) {
call A_1
. . .
call A_n
}

ATOMS WITHIN A GOAL ≡ CALLS TO METHODS

?- C_1,…, C_k

call C_1
. . .
call C_k

RESOLUTION STEP ≡ AN EXECUTION STEP

UNIFICATION ≡ MECHANISM FOR:
Parameter passing
Data construction and selection

3. Computational model of LP

- The computational model of LP is based on the **Resolution** inference rule

- Basic idea: in order to execute a call **A** (an atom):

  - If the program contains a fact A_0 that *unifies with* A then we say that A succeeds (and conclude that it is ***true***).

  - If the program contains a clause A_0 :- A_1, .., A_n such that A_0 and A *unify*, then we have to further check A_1 to A_n as new independent goals.

¿How to deal with variables in queries?

Unification or bidirectional parameter passing

- **The unification** of two expressions A and B consists of finding the least (most general) substitution σ for their variables such that $\sigma(A)=\sigma(B)$.
- Informally:

	X	c	$f(t_1,\ldots,t_n)$
X'	Yes, {X/X'}	Yes, {X'/c}	Yes, {X'/ $f(t_1,\ldots,t_n)$}
c'	Yes, {X/c'}	Only if c=c'	No
$f'(t'_1,\ldots,t'_m)$	Yes, {X/ $f'(t'_1,\ldots,t'_m)$}	No	Only if f=f', n=m and t_i and t'_i unify for all i

1. expressions with different root symbol or different arity (i.e., number of arguments) do not unify
2. no binding for a variable can contain the same variable (otherwise, an infinite term would be created). This is know as the "occur check" problem.

Unification (bidirectional parameter passing)

- **Notation:** A substitution $\{x_1 \rightarrow t_1, \ldots, x_n \rightarrow t_n\}$ is traditionally denoted (in LP) as $\{x_1/t_1, \ldots, x_n/t_n\}$
- Example:

A unifies…	…with B…	…using θ
flies(theFly)	flies(theFly)	{ }
X	Y	{X/Y}
X	a	{X/a}
f(X,g(t))	f(m(h),g(M))	{X/m(h), M/t}
f(X,g(t))	f(m(h),t(M))	impossible (1)
f(X,X)	f(Y,h(Y))	impossible (2)

Lists unification

- Examples:

  [a,b] and [X|R] unify using {X/a, R/[b]}

  [a] and [X|R] unify using {X/a, R/[]}

  [a|X] and [Y,b,c] unify using {Y/a, X/[b,c]}

  [a] and [X,Y|R] do not unify

  [] and [X] do not unify

  - IMPORTANT: both lists must have a *uniform format* before any unification test!

MGU (most general unifier)

- During the program execution, we need to compute the **MGU** of the clause heads and the atoms in the goal

How to compute the mgu? (I)

- Given expressions t_1 and t_2, if one of them is a variable, for instance, t_1 is X:
  - Return $\{X/t_2\}$ as the mgu
  - exception 1: if $t_1 = t_2 = X$, then the mgu is { } (empty substitution)
  - exception 2: if t_2 is not a variable, and X occurs in t_2, failure! (there is no mgu)

Note: Dealing with different variables, e.g., X and Y, both {X/Y} and {Y/X} are valid MGUs.

MGU (most general unifier)

- During the program execution, we need to compute the **MGU** of the clause heads and the atoms in the goal

How to compute the mgu? (II)

- If the expressions are $p(t_1,\ldots,t_n)$ and $q(s_1,\ldots,s_m)$
  - Check that $p=q$ and $n=m$ (otherwise, failure)
  - Consider the terms t_i and s from left to right (i.e., i=1,…,n), and unify t_i and s_i using this algorithm for i=1,…,n
  - For each i, the computed unifier θ_i for t_i and s_i, must be applied to all $t_1,\ldots, t_n, s_1,\ldots, s_m$ and all terms of previously computed mgu's before attempting the unification of t_{i+1} and s_{i+1}
  - If some of the unifications fail we end with failure
  - If we reach the end without failure (both expressions are now identical), the union of all the θ_i is the MGU of the expressions

MGU (most general unifier): Example

- Which is the MGU of p([X,c], X) and p([f(Y)|R], f(a))?

1. Write the lists in the same format:

   p([X|[c]], X) and p([f(Y)|R], f(a))

2. Both predicate symbol and arity (num of arguments) coincide, so can compute the unifiers from left to right:

   p([X|[c]], X)

   p([f(Y)|R], f(a))

1st argument: does [X|[c]] and [f(Y)|R] unifiy? Yes: {X/f(Y),R/[c]}

MGU (most general unifier): Example

- Which is the MGU of p([X,c], X) and p([f(Y)|R], f(a))?

1. Write the lines in the same format:

   p([X|[c]], X) and p([f(Y)|R], f(a))

2. Both predicate symbol and arity (num of arguments) coincide, so can compute the unifiers from left to right:

   p([X|[c]], X)) => p([f(Y)|[c]], f(Y))

   p([f(Y)|R], f(a)) => p([f(Y)|[c]], f(a))

   {X/f(Y),R/[c]}

Now apply {X/f(Y),R/[c]} to all terms

MGU (most general unifier): Example

- Which is the MGU of p([X,c], X) and p([f(Y)|R], f(a))?

1. Write the lists in the same format:

   p([X|[c]], X) and p([f(Y)|R], f(a))

2. Both predicate symbol and arity (num of arguments) coincide, so can compute the unifiers from left to right:

   p([X|[c]], X) => p([f(Y)|[c]], f(Y))

   p([f(Y)|R], f(a)) => p([f(Y)|[c]], f(a))

   {X/f(Y),R/[c]}

2nd argument: does f(Y) and f(a) unify? Yes: {Y/a}

MGU (most general unifier): Example

- Which is the MGU of p([X,c], X) and p([f(Y)|R], f(a))?

1. Write the lists in the same format:

   p([X|[c]], X) and p([f(Y)|R], f(a))

2. Both predicate symbol and arity (num of arguments) coincide, so can compute the unifiers from left to right:

p([X|[c]], X)	=>	p([f(Y)|[c]], f(Y))	=>	p([f(a)|[c]], f(a))
p([f(Y)|R], f(a))	=>	p([f(Y)|[c]], f(a))	=>	p([f(a)|[c]], f(a))
		{X/f(a),R/[c]}		{Y/a}

Now apply {Y/a} to all terms

(including the previously computed mgu)

MGU (most general unifier): Example

- Which is the MGU of p([X,c], X) and p([f(Y)|R], f(a))?

1. Write the lists in the same format:

   p([X|[c]], X) and p([f(Y)|R], f(a))

2. Both predicate symbol and arity (num of arguments) coincide, so can compute the unifiers from left to right:

   p([X|[c]], X) => p([f(Y)|[c]], f(Y)) => p([f(a)|[c]], f(a))

   p([f(Y)|R], f(a)) => p([f(Y)|[c]], f(a)) => p([f(a)|[c]], f(a))

   {X/f(a),R/[c]} {Y/a}

The MGU is {X/f(a),R/[c]} U {Y/a} = {X/f(a), R/[c], Y/a}

Exercises MGU

- Which (if any) is the MGU of

  p(f(X, b), Z) y p(f(a, Y), g(c)) ?

- Which (if any) is the MGU of

  p([a,X], Y) y p([H|R], b) ?

- Which (if any) is the MGU of

  p(f(X),b,X) y p(f(a),Y,b) ?

3. The computational model of logic programming: Resolution

Given a logic program P and a goal ?-A_1,…,A_m ,

- **If** P contains a clause A :- B_1, .., B_n (with variables renamed to avoid unification clashes) and A and A_1 unify with mgu σ

  then the application of the resolution rule yields a new goal

$$
\frac{\begin{array}{l} \cancel{A} \text{ :- } B_1,\ldots, B_n \\ \text{?- } \cancel{A_1}, A_2,\ldots,A_m \end{array}}{\text{?- } (B_1,\ldots, B_n, A_2,\ldots, A_m)\sigma}
$$

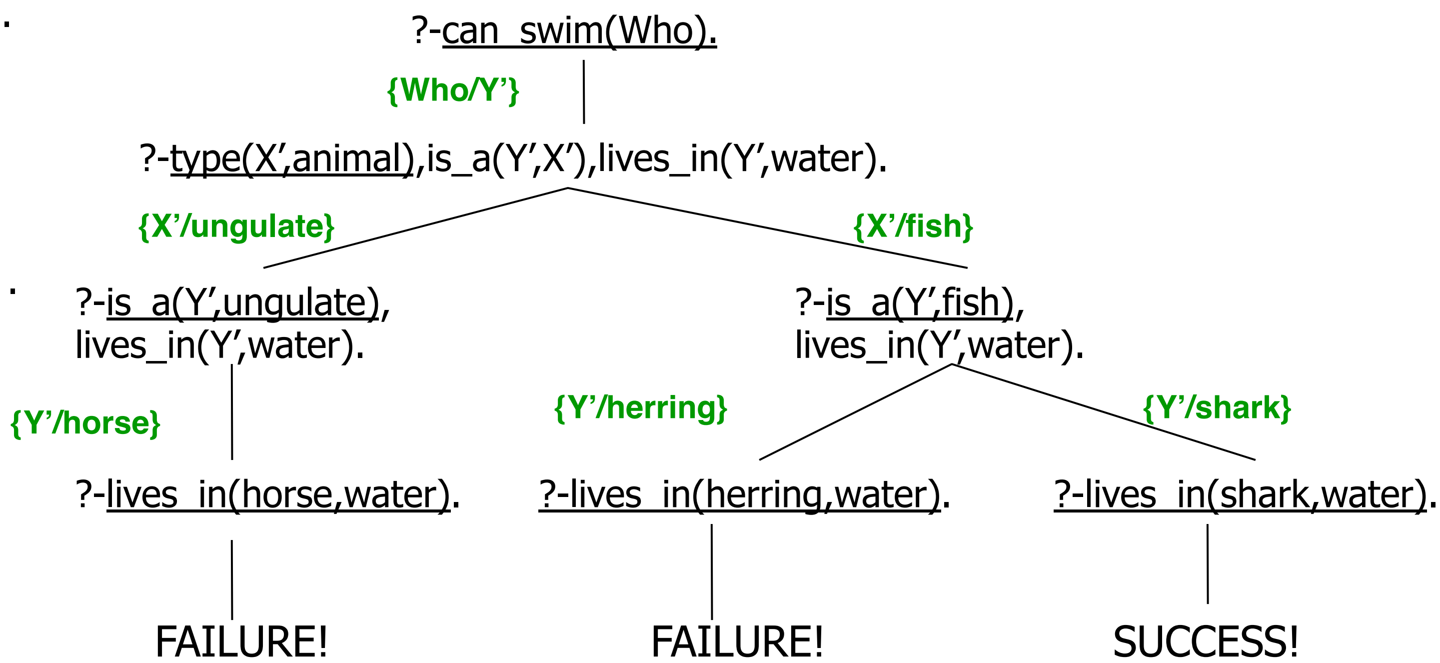
- The successive application of this rule generates a search tree.
- A computation or derivation is a sequence of resolution steps that corresponds to one of the branches in the tree.

Search tree

```
type(ungulate,animal) .
type(fish,animal) .
is_a(horse,ungulate) .
is_a(herring,fish) .
is_a(shark,fish) .

lives_in(horse,ground) .
lives_in(frog,ground) .
lives_in(frog,water) .
lives_in(shark,water) .

can_swim(Y):-
    type(X,animal),
    is_a(Y,X),
    lives_in(Y,water) .
```



The computational model of logic programming

Types of computation

- ***Finite***: the computation terminates in a finite number of steps, i.e., it is finite. Two kinds of finite computations are considered:
  - ***Failure:*** no clause unifies with the selected atom A_1
  - ***Successful***: an empty clause (?-) is obtained. This is also called a refutation.

> Each successful branch yields a **computed answer** which is obtained as the (restriction to the variables of the initial goal of the) composition $\theta_1\theta_2\cdots\theta_n$ of the sequence of mgu's that are obtained along the branch.

- ***Infinite***: in any goal of the sequence, the selected atom A_1 unifies with (a variant of) a program clause

Types of derivations

INFINITE

$\{p(f(X)) \leftarrow p(X)\}$

$?-\ p(X)$
$\Downarrow \{X/f(X')\}$
$?-\ p(X')$
$\Downarrow \{X'/f(X'')\}$
$?-\ p(X'')$
$\Downarrow \{X''/f(X''')\}$
$?-\ p(X''')$
$\Downarrow\ (\infty)$

FAILED

$\{p(0) \leftarrow q(X)\}$

$?-\ p(Z)$
$\Downarrow \{Z/0\}$
$?-\ q(X)$
$\Downarrow$ ***fail***

SUCCESSFUL

$\{p(0) \leftarrow q(X)$ }
$q(1) \leftarrow \}$

$?-\ p(Z)$
$\Downarrow \{Z/0\}$
$?-\ q(X)$
$\Downarrow \{X/1\}$
'?–' ***success!***

Renaming is important!

Example:

p(f(Z)) :- q(Z).	?- p(X)	
q(Y) :- r(X).	$\Downarrow$	{X/f(Z)}
r(a).	?- q(Z)	
	$\Downarrow$	{Z/Y}
	?- r(X)	
	$\Downarrow$	{X/a}
	?-	

wrong because X is bound to *two different terms* in the same derivation. The problem is that the second clause was used without the appropriate renaming: the variable X in the initial goal has nothing to do with X in the second clause.

Renaming is important!

Exercise: Which is the answer to the goal

?-p(X).

with respect to the following program?

```
r(0).
p(Y)    :- q.
 q      :- r(Y).
```

A. {X/0, Y'/0}
B. {X/Y'}
C. {X/0}
D. {Y'/0}

Predefined search

The search rule determines:

1) The order for trying the clause programs and,
2) The strategy for traversing the obtained tree.

   Two main strategies:

   * *depth-first*: completeness of SLD resolution gets lost.

   * *breadth-first*: the search tree is traversed from top to bottom, covering each level before changing to the next one. It is complete, but costly.

PROLOG: automatic predefined search

1) top-down,
2) depth-first search with backtracking.

Exercise

- Compute the search tree for the goal

```
?- pair(Person1,Person2).
```

using the previously proposed program and the following facts:

```
editor(zenspider, emacs).
editor(drbrain, vim).
editor(phiggins, vim).
editor(tenderlove, vim).

pair(Person1,Person2) :- editor(Person1, Editor),
                         editor(Person2, Editor),
                         Person1\==Person2.
```

Exercise

- Obtain the search tree for the goal

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

with respect to the logic program

```
length ([], 0).
length ([_|T], N) :- length(T, N1),
                     N is N1+1.
```

4. Some practical issues

Aplications of LP

- Software and hardware verification
- Program certification
- Automated prototyping
- Automated software engineering (automated debugging, program synthesis, program transformation,…)
- Modelling Information Systems and Data Bases
- Learning
- Robotics and scheduling
- Expert systems
- Natural language processing

LTP: References

BASIC

- Pascual Julián Iranzo, María Alpuente Frasnedo. *Programación lógica: teoría y práctica*. Pearson Educación, 2007.
- W.F. Clocksin, C.S. Mellish. *Programming in PROLOG,* 5th Edition. Springer-Verlag, 2003.

ADDITIONAL

- Krzysztof R. Apt. *From logic programming to Prolog*. Prentice Hall, 1997.
- Leon Sterling. *The art of Prolog : Advanced programming techniques*. MIT Press, 1997.