Principles of programming languages
Lecture 8

http://few.vu.nl/~nsilvis/PPL/2007

Natalia Silvis-Cividjian
e-mail: nsilvis@few.vu.nl

vrije Universiteit *amsterdam*

Announcements

- Exam 19 dec moved to 15:15-18:00.
- Guest lecture moved to 14 dec.
- Deadlines reports: 1 December
- Student presentations in 2 sessions:

30 nov and 7 december

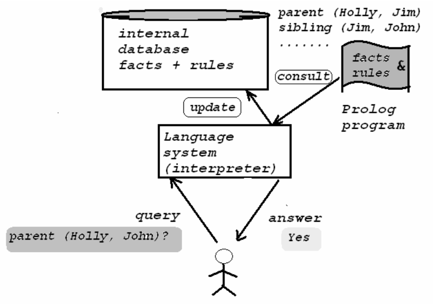
Logic programming

- ➡ First look at Prolog
- Declarative vs. Procedural programming
- Behind the scenes
- Second look at Prolog
- Example

The idea of logic programming

- A program is a collection of facts and rules (Horn clauses)
- The user can ask questions to the language system by giving a **goal** to prove. The language system tries to prove the goal by using the clauses given in the program.
- Goal - oriented

Logical programming



Facts about Prolog

- Prolog = Programming in Logic (University of Marseille, mid 70s)
- The most important logic language (other - Godel, Escher)
- A declarative language. Programmer only specifies the problem and the language system finds a solution.
- Used in AI applications (automated reasoning systems, expert systems)

Terms

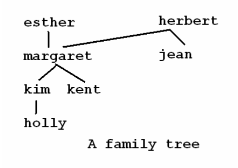
- All Prolog programs and data are built from terms:
  - Constants: **`1, 1.23, fred, *,=, []`**
  - Variables: **`X, Y, Fred, Child, _`**
  - Compound terms: **`parent(X,Y)`**

The Prolog Database

- A Prolog language system maintains a database of facts and rules of inference
- A Prolog program is a set of data for this database

Facts

```
parent(kim,holly).
parent(margaret,kim).
parent(margaret,kent).
parent(esther,margaret).
parent(herbert,margaret).
parent(herbert,jean).
```



- A fact is a term followed by a colon.
- This is a Prolog program of six facts
- parent is a predicate of arity 2
- Natural interpretation : facts about families: Kim is the parent of Holly , etc

SWI-Prolog

```
Welcome to SWI-Prolog (Version 3.4.2)
Copyright (c) 1990-2000 University of Amsterdam.
Copy policy: GPL-2 (see www.gnu.org)

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

?-
```

- Prompting for a query with **`?-`**
- To exit Prolog, use either **`?-halt.`** **`or Ctrl-D`**

Read a program

```
?- consult(relations).
% relations compiled 0.00 sec, 0 bytes

Yes
?-
```

- `Consult` is used to read a program from a file into the database
- File **`relations`** (or **`relations.pl`**) contains **`parent`** facts

Simple Queries

```
?- parent(margaret,kent).

Yes
?- parent(fred,pebbles).

No
?-
```

- Now we can ask the language system to prove something = make a query
- The answer will be **`Yes`** or **`No`**
- (Some queries, like **`consult`**, are executed only for their side-effects)

Queries With Variables

```
?- parent(P,jean).

P = herbert

Yes
?- parent(P,esther).

No
```

Here, it waits for input. We hit Enter to make it proceed.

- The Prolog system shows the bindings necessary to prove the query
- The binding makes a query term **unify** with a fact term from the program

Unification

- Pattern-matching in Prolog
- Two terms **unify** if there is some way of binding their variables that makes them identical

parent(P,jean)   **parent(herbert,jean**

Unification

- Two Prolog terms t_1 and t_2 *unify* if there is some substitution σ (their *unifier*) that makes them identical: $\sigma(t_1) = \sigma(t_2)$
  - **a** and **b** do not unify
  - **f(X,b)** and **f(a,Y)** unify: a unifier is {**X**→**a**, **Y**→**b**}
  - **f(X,b)** and **g(X,b)** do not unify
  - **a(X,X,b)** and **a(b,X,X)** unify: a unifier is {**X**→**b**}
  - **a(X,X,b)** and **a(c,X,X)** do not unify
  - **a(X,f)** and **a(X,f)** do unify: a unifier is {}

Substitutions

- A *substitution* is a function that maps variables to terms:
  σ = {**X**→**a**, P→herbert}
- This σ maps **X** to **a** and **P** to herbert
- The result of applying a substitution to a term is an *instance* of the term

Flexibility

- More flexible than Java
- Normally, variables can appear in any or all positions in a query:
  - **parent(Parent,jean)**
  - **parent(esther,Child)**
  - **parent(Parent,Child)**
  - **parent(Person,Person)**

Conjunctions

```
?- parent(margaret,X), parent(X,holly).

X = kim

Yes
```

- A conjunctive query has a list of query terms separated by commas
- The Prolog system tries prove them all (using a single set of bindings)

Multiple Solutions

```
?- parent(margaret,Child).

Child = kim ;

Child = kent ;

No
```

- There might be more than one way to prove the query
- By typing ; rather than Enter, you ask the Prolog system to find more

```
?- parent(Parent,kim), parent(Grandparent,Parent).

Parent = margaret
Grandparent = esther ;

Parent = margaret
Grandparent = herbert ;

No
?- parent(esther,Child),
|     parent(Child,Grandchild),
|     parent(Grandchild,GreatGrandchild).

Child = margaret
Grandchild = kim
GreatGrandchild = holly

Yes
```

Rules

head

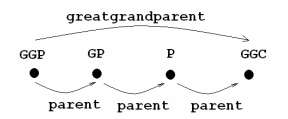
```
greatgrandparent(GGP,GGC) :-
   parent(GGP,GP),
   parent(GP,P),
   parent(P,GGC).
```

conditions

- A **rule** says how to prove something: to prove the head, prove the conditions
- To prove `greatgrandparent(GGP,GGC)`, find some `GP` and `P` for which you can prove `parent(GGP,GP)`, then `parent(GP,P)` and then finally `parent(P,GGC)`

Example

```
parent(kim,holly).
parent(margaret,kim).
parent(margaret,kent).
parent(esther,margaret).
parent(herbert,margaret).
parent(herbert,jean).
greatgrandparent(GGP,GGC) :-
  parent(GGP,GP), parent(GP,P), parent(P,GGC).
```



Example

```
?- greatgrandparent(esther,GreatGrandchild).

GreatGrandchild = holly

Yes
```

- This shows the initial query and final result
- Internally, there are intermediate *goals:*
  - The first goal is the initial query
  - The next is what remains to be proved after transforming the first goal using one of the clauses (in this case, the greatgrandparent rule)
  - And so on, until nothing remains to be proved

1. `parent(kim,holly).`
2. `parent(margaret,kim).`
3. `parent(margaret,kent).`
4. `parent(esther,margaret).`
5. `parent(herbert,margaret).`
6. `parent(herbert,jean).`
7. `greatgrandparent(GGP,GGC) :-`
   `parent(GGP,GP), parent(GP,P), parent(P,GGC).`

`greatgrandparent(esther,GreatGrandchild)`

⇓ Clause 7, binding `GGP` to `esther` and `GGC` to `GreatGrandchild`

`parent(esther,GP), parent(GP,P), parent(P,GreatGrandchild)`

⇓ Clause 4, binding `GP` to `margaret`

`parent(margaret,P), parent(P,GreatGrandchild)`

⇓ Clause 2, binding `P` to `kim`

`parent(kim,GreatGrandchild)`

⇓ Clause 1, binding `GreatGrandchild` to `holly`

Scope

```
grandparent(GP,GC) :-
  parent(GP,P), parent(P,GC).

greatgrandparent(GGP,GGC) :-
  grandparent(GGP,P), parent(P,GGC).
```

- Same relation, defined indirectly
- Note that both clauses use a variable **P**
- First occurrence of a variable serves as definition
- The scope of the definition of a variable is the clause that contains it

Recursion

- Recursion plays a central role in Prolog
- For example take the relation `parent(X,Y)`
- Such a relation can be generalized to X is `ancestor` of Y, to operate over as many generations as necessary.

Recursive Rules

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :-
  parent(Z,Y),
  ancestor(X,Z).
```

- **X** is an ancestor of **Y** if:
  - Base case: **X** is a parent of **Y**
  - Recursive case: there is some **Z** such that **Z** is a parent of **Y**, and **X** is an ancestor of **Z**
- Prolog tries rules in the order you give them, so put base-case rules and facts first

```
?- ancestor(jean,jean).

No
?- ancestor(kim,holly).

Yes
?- ancestor(A,holly).

A = kim ;

A = margaret ;

A = esther ;

A = herbert ;

No
```

Core Syntax Of Prolog

- You have seen the complete core syntax:

```
<clause> ::=  <fact> | <rule>
<fact> ::=  <term> .
<rule> ::=  <term> :- <termlist> .
<termlist> ::=  <term> | <term> , <termlist>
```

- There is not much more syntax for Prolog than this: it is a very simple language

Let’s practice with a joke

Monty Python and the Holy Grail (Scene 5, The Witch Scene)

Script available from :
http://www.calvin.edu/~rpruim/courses/m156/F99/prolog/duck.txt

The Prolog program

```
witch(X)  :- burns(X), female(X).
burns(X)  :- wooden(X).
wooden(X) :- floats(X).
floats(X) :- sameweight(duck, X).
female(girl). /* by observation */
sameweight(duck,girl). /*by experiment*/

? witch(girl).
```

Adapted from
http://www.csse.monash.edu.au/~lloyd/tildeLogic/Prolog.toy/Examples/.

```
$ pl
Welcome to SWI-Prolog (Multi-threaded, Version 5.6.32)
Copyright (c) 1990-2007 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).

?- consult(witch).
% witch compiled 0.00 sec, 1,428 bytes

Yes
?- witch(girl).

Yes
?- █
```

Logic programming

- First look at Prolog
- ➡ Declarative vs. Procedural programming
- Behind the scenes
- Second look at Prolog
- Example

The Procedural Side

```
greatgrandparent(GGP,GGC) :-
  parent(GGP,GP), parent(GP,P), parent(P,GGC).
```

- A rule says how to prove something:
  - To prove `greatgrandparent(GGP,GGC)`, find some `GP` and `P` for which you can prove `parent(GGP,GP)`, then `parent(GP,P)` and then finally `parent(P,GGC)`
- A Prolog program specifies proof procedures for queries

The Declarative Side

- A rule is a logical assertion:
  - For all bindings of `GGP`, `GP`, `P`, and `GGC`, if `parent(GGP,GP)` and `parent(GP,P)` and `parent(P,GGC)`, then `greatgrandparent(GGP,GGC)`
- Just a formula – it doesn't say how to *do* anything – it just makes an assertion:

$$\forall GGP, GP, P, GGC \,.\, \text{parent}(GGP, GP) \wedge \text{parent}(GP, P) \wedge \text{parent}(P, GGC) \Rightarrow \text{greatgrandparent}(GGP, GGC)$$

Declarative Languages

- Each piece of the program corresponds to a simple mathematical abstraction
  - Prolog clauses – formulae in first-order logic
  - ML fun definitions – functions
- Many people use *declarative* as the opposite of *imperative*, including both logic languages and functional languages

Declarative Advantages

- Imperative languages are doomed to subtle side-effects and interdependencies
- Simpler declarative semantics makes it easier to develop and maintain correct programs
- Higher-level, more like *automatic programming*: describe the problem and have the computer write the program

Prolog Has Both Aspects

- Partly declarative
  - A Prolog program has logical content
- Partly procedural
  - A Prolog program has procedural concerns: clause ordering, condition ordering, side-effecting predicates, etc.
- It is important to be aware of both

Logic programming

- First look at Prolog
- Declarative vs. Procedural programming
- ➡ Behind the scenes
- Second look at Prolog
- Example

Backtracking

- The language system has to look for a solution to a query and guarantee termination – difficult task.
- Prolog uses a simple backtracking strategy
- If efficiency is important, backtracking can be controlled.

Backtracking

- Simple backtracking: depth-first tree search with subgoal evaluation from left to right
- User gives a goal. Prolog starts to satisfy the list of subgoals and if failure occurs it backtracks and tries an alternative set of goals

Backtracking

- Prolog explores all possible targets of each call, until it finds as many successes as the caller requires or runs out of possibilities

Backtracking Example

```
1. loves(john,jane).
2. loves(john,hilda).
3. loves(bill,jane).
4. loves(james,jane).
5. loves(mary,bill).
6. loves(jane,james).
7. goodmatch(X,Y) :- loves(X,Y),loves(Y,X)
```

```
?- goodmatch(A,B)
```

Trace

```
?-
|    trace.

Yes
[trace]  ?- goodmatch(A,B).
   Call: (6) goodmatch(_G283, _G284) ? creep
   Call: (7) loves(_G283, _G284) ? creep
   Exit: (7) loves(john, jane) ? creep
   Call: (7) loves(jane, john) ? creep
   Fail: (7) loves(jane, john) ? creep
   Redo: (7) loves(_G283, _G284) ? creep
   Exit: (7) loves(john, hilda) ? creep
   Call: (7) loves(hilda, john) ? creep
   Fail: (7) loves(hilda, john) ? creep
   Redo: (7) loves(_G283, _G284) ? creep
   Exit: (7) loves(bill, jane) ? creep
   Call: (7) loves(jane, bill) ? creep
   Fail: (7) loves(jane, bill) ? creep
   Redo: (7) loves(_G283, _G284) ? creep
   Exit: (7) loves(james, jane) ? creep
   Call: (7) loves(jane, james) ? creep
   Exit: (7) loves(jane, james) ? creep
   Exit: (6) goodmatch(james, jane) ? creep

A = james
B = jane

Yes
[debug]  ?-
```

What is the disadvantage of Prolog automatic backtracking?

Controlled backtracking - cut

- The cut operation ! can stop backtracking.
- Prevents from trying alternatives that cannot succeed.
- Useful in mutually exclusive clauses

Same example, with cut

```
1. loves(john,jane).
2. loves(bill,jane).
3. loves(james,jane).
4. loves(mary,bill).
5. loves(jane,james).
6. goodmatch(X,Y) :- loves(X,Y),loves(Y,X),!.
```

Disadvantage of cuts

- Declarative and procedural meaning may differ
- Green and red cuts

Logic programming

- First look at Prolog
- Declarative vs. Procedural programming
- Behind the scenes
- ⇨ Second look at Prolog
- Example

Second look at Prolog

- Operators
- Lists
- Anonymous
- Negation

Operators

- Prolog has some predefined operators (and the ability to define new ones)
- An operator is just a predicate for which a special abbreviated syntax is supported

The = Predicate

- The goal **=(X,Y)** succeeds if and only if **X** and **Y** can be unified:

```
?- =(parent(adam,seth),parent(adam,X)).

X = seth

Yes
```

- Since = is an operator, it can be and usually is written like this:

```
?- parent(adam,seth)=parent(adam,X).

X = seth

Yes
```

Arithmetic Operators

- Predicates **+**, **-**, ***** and **/** are operators too, with the usual precedence and associativity

```
?- X = +(1,*(2,3)).

X = 1+2*3

Yes
?- X = 1+2*3.

X = 1+2*3

Yes
```

Prolog lets you use operator notation, and prints it out that way, but the underlying term is still **+(1,*(2,3))**

Terms are Not Evaluated

```
?- +(X,Y) = 1+2*3.

X = 1
Y = 2*3

Yes
?- 7 = 1+2*3.

No
```

- The term is still **+(1,*(2,3))**
- It is not evaluated

Lists in Prolog

- A bit like ML lists
- The atom `[]` represents the empty list
- A predicate `.` corresponds to ML's `::` operator

ML expression	Prolog term
`[]`	`[]`
`1::[]`	`.(1,[])`
`1::2::3::[]`	`.(1,.(2,.(3,[])))`
No equivalent.	`.(1,.(parent(X,Y),[]))`

List Notation

List notation	Term denoted
`[]`	`[]`
`[1]`	`.(1,[])`
`[1,2,3]`	`.(1,.(2,.(3,[])))`
`[1,parent(X,Y)]`	`.(1,.(parent(X,Y),[]))`

- ML-style notation for lists
- These are just abbreviations for the underlying term using the `.` predicate
- Prolog usually displays lists in this notation

Example

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

X = [1, 2, 3]

Yes
?- .(X,Y) = [1,2,3].

X = 1
Y = [2, 3]

Yes
```

List Notation With Tail

- Last in a list can be the symbol `|` followed by a final term for the tail of the list
- Useful in patterns: `[1,2|X]` unifies with any list that starts with `1,2` and binds `X` to the tail

```
?- [1,2|X] = [1,2,3,4,5].

X = [3, 4, 5]

Yes
```

The **append** Predicate

```
?- append([1,2],[3,4],Z).

Z = [1, 2, 3, 4]

Yes
```

- Predefined **append(X,Y,Z)** succeeds if and only if **Z** is the result of appending the list **Y** onto the end of the list **X**

append is Not Just A Function

```
?- append(X,[3,4],[1,2,3,4]).

X = [1, 2]

Yes
```

- **append** can be used with any pattern of instantiation (that is, with variables in any positions)

Not Just A Function

```
?- append(X,Y,[1,2,3]).

X = []
Y = [1, 2, 3] ;

X = [1]
Y = [2, 3] ;

X = [1, 2]
Y = [3] ;

X = [1, 2, 3]
Y = [] ;

No
```

Other Predefined List Predicates

Predicate	Description
`member(X,Y)`	Provable if the list **Y** contains the element **X**.
`select(X,Y,Z)`	Provable if the list **Y** contains the element **X**, and **Z** is the same as **Y** but with one instance of **X** removed.
`nth0(X,Y,Z)`	Provable if **X** is an integer, **Y** is a list, and **Z** is the **X**th element of **Y**, counting from 0.
`length(X,Y)`	Provable if **X** is a list of length **Y**.

- All flexible, like **`append`**
- Queries can contain variables anywhere

Using `select`

```
?- select(2,[1,2,3],Z).

Z = [1, 3] ;

No
?- select(2,Y,[1,3]).

Y = [2, 1, 3] ;

Y = [1, 2, 3] ;

Y = [1, 3, 2] ;

No
```

The `reverse` Predicate

```
?- reverse([1,2,3,4],Y).

Y = [4, 3, 2, 1] ;

No
```

- Predefined **`reverse(X,Y)`** unifies **`Y`** with the reverse of the list **`X`**
- **`Not flexible`**

The Anonymous Variable

- The variable _ is an anonymous variable
- Every occurrence is bound independently of every other occurrence
- In effect, much like ML's _: it matches any term without introducing bindings

Example

```
tailof(.(_,A),A).
```

- This **`tailof(X,Y)`** succeeds when **`X`** is a non-empty list and **`Y`** is the tail of that list
- Don't use this, even though it works:

```
tailof(.(Head,A),A).
```

The **not** Predicate

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

Yes
?- not(member(4,[1,2,3])).

Yes
```

- For simple applications, it often works quite a bit logical negation
- But it has an important procedural side…

Negation

- Prolog uses the **closed-world assumption=it assumes that all facts about the world are included in the model.**
- Ex: To prove **`not married (fred),`** Prolog attempts to prove **`married (fred).`**
- **`not married (fred)`** succeeds if the system fails to prove that **`fred`** is married.
- It can happen in 2 cases:
  - Bill is really not married
  - System cannot prove that bill is married
- Conclusion: take care in interpreting the answer to a negation question!!

Negation

- The two faces of Prolog:
  - Declarative: **`not(X)`** = ¬**X**
  - Procedural: **`not(X)`** succeeds if **`X`** fails, fails if **`X`** succeeds, and runs forever if **`X`** runs forever

Example

```
sibling(X,Y) :-
  parent(P,X),
  parent(P,Y),
  not(X=Y).
```

```
sibling(X,Y) :-
  not(X=Y),
  parent(P,X),
  parent(P,Y).
```

```
?- sibling(kim,kent).

Yes
?- sibling(kim,kim).

No
?- sibling(X,Y).

No
```

```
?- sibling(X,Y).

X = kim
Y = kent ;

X = kent
Y = kim ;

X = margaret
Y = jean ;

X = jean
Y = margaret ;

No
```

Logic programming

- First look at Prolog
- Declarative vs. Procedural programming
- Behind the scenes
- Second look at Prolog
- ⇨ Example

The problem: A Classic Riddle

- A man travels with wolf, goat and cabbage
- Wants to cross a river from west to east
- A rowboat is available, but only large enough for the man plus one possession
- Wolf eats goat if left alone together
- Goat eats cabbage if left alone together
- How can the man cross without loss?

- http://www.mathcats.com/explore/river/crossing.html

Configurations

- Represent a configuration of this system as a list showing which bank each thing is on in this order: man, wolf, goat, cabbage
- Initial configuration: **`[w,w,w,w]`**
- If man crosses with wolf, new state is **`[e,e,w,w]`** – but then goat eats cabbage, so we can't go through that state
- Desired final state: **`[e,e,e,e]`**

Moves

- In each move, man crosses with at most one of his possessions
- We will represent these four moves with four atoms: **`wolf`**, **`goat`**, **`cabbage`**, **`nothing`**
- (Here, **`nothing`** indicates that the man crosses alone in the boat)

Moves Transform Configurations

- Each move transforms one configuration to another
- In Prolog, we will write this as a predicate: **`move(Config,Move,NextConfig)`**
  - **`Config`** is a configuration (like **`[w,w,w,w]`**)
  - **`Move`** is a move (like **`wolf`**)
  - **`NextConfig`** is the resulting configuration (in this case, **`[e,e,w,w]`**)

The **`move`** Predicate

```
change(e,w).
change(w,e).

move([X,X,Goat,Cabbage],wolf,[Y,Y,Goat,Cabbage]) :-
  change(X,Y).
move([X,Wolf,X,Cabbage],goat,[Y,Wolf,Y,Cabbage]) :-
  change(X,Y).
move([X,Wolf,Goat,X],cabbage,[Y,Wolf,Goat,Y]) :-
  change(X,Y).
move([X,Wolf,Goat,C],nothing,[Y,Wolf,Goat,C]) :-
  change(X,Y).
```

Safe Configurations

- A configuration is safe if
  - At least one of the goat or the wolf is on the same side as the man, and
  - At least one of the goat or the cabbage is on the same side as the man

```
oneEq(X,X,_).
oneEq(X,_,X).

safe([Man,Wolf,Goat,Cabbage]) :-
  oneEq(Man,Goat,Wolf),
  oneEq(Man,Goat,Cabbage).
```

Solutions

- A solution is a starting configuration and a list of moves that takes you to **`[e,e,e,e]`**, where all the intermediate configurations are safe

```
solution([e,e,e,e],[]).
solution(Config,[Move|Rest]) :-
  move(Config,Move,NextConfig),
  safe(NextConfig),
  solution(NextConfig,Rest).
```

Prolog Finds A Solution

```
?- length(X,7), solution([w,w,w,w],X).

X = [goat, nothing, wolf, goat, cabbage, nothing, goat]

Yes
```

- Note: without the **`length(X,7)`** restriction, Prolog would not find a solution
- It gets lost looking at possible solutions like **`[goat,goat,goat,goat,goat…]`**

What Prolog Is Good For

- The program specified a problem logically
- It did not say how to search for a solution to the problem – Prolog took it from there
- That's one kind of problem Prolog is especially good for

Conclusion

- Logic programming is almost exclusively carried out by Prolog
- Prolog is a declarative language and is goal oriented
- A Prolog program consists of facts and rules
- The Prolog language system solves a query by matching. If the goal fails an alternative is tried by backtracking.
- Efficiency can be improved with cuts.
- Prolog is suited to solve logic problems in AI