F28PL1 Programming Languages

Lecture 16: Prolog 1

Overview

- logic programming language
- roots in predicate logic
- developed by Alan Colmerauer & collaborators in Marseilles, in early 1970s
- ISO standard derived from University of Edinburgh
- adapted for Japanese 5th Generation programme, 1980s
- now widely used for Artificial Intelligence research & education

Overview

- based on *logic programming*
  - use of predicate logic as a specification language
  - an implementation of predicate logic would enable the use of specifications directly as programs
- concentrate on describing a problem solution as an input/output *relation*
  - not an input/output *process*
  - i.e. in a *descriptive* rather than a *prescriptive* manner

Overview

- enables a high degree of abstraction and of implementation independence
- emphasis is on *what* is to be done rather than *how* it is to be done
- predicate logic has a well developed proof theory
  - use formal techniques to manipulate/verify specifications
- specification can be used to :
  - check that outputs correspond to inputs
  - find outputs from inputs
  - find inputs from outputs

Overview

- not a pure logic programming language
- known evaluation order for predicate manipulation
  - implementation considerations are used by programmers
- many predicates can only be used for checking or for finding outputs from inputs but not both
- quantification must be made explicit

Overview

- differences with imperative languages:
  - no necessary distinction between programs and data
  - there is no concept of a statement as a state change, for example, through assignment
  - like functional languages
  - evaluation order is not necessarily linear

Prolog resources

- we will use SICStus Prolog interpreter
  - from the Swedish Institute of Computer Science
  - licenses cost real money – don’t buy one!
- SICStus documentation from:
  - http://www.sics.se/isl/sicstuswww/site/documentation.html
- free Prologs from:
  - http://www.gprolog.org/
  - http://www.swi-prolog.org/
- W. F. Clocksin & C. S. Mellish, Programming in Prolog: Using the ISO Standard, (5th edition), Springer, 2003

Running Prolog

- to run Prolog on Linux:

```
$ sicstus
SICStus 4.2.1 (x86_64-linux-glibc2.7):
 Wed Feb  1 01:15:06 CET 2012
Licensed to SP4macs.hw.ac.uk
| ?-
```

- `| ?-` - Prolog prompt
- system commands are Prolog *terms*
  - end with a .

Running Prolog

- system does not support interactive editing
  - use separate windows for program & interpreter
- to load a program

```
| ?- [file name].
```

- `file name` is any valid Linux path
- if not a single word then in `'...'`
- file name convention
  - Prolog files end with `.pl`

Running Prolog

- to turn on tracing:

```
| ? – trace.
```

to turn off tracing:

```
| ? – notrace.
```

- to leave Prolog

```
| ?- ^D
```

Prolog summary

- weak types
  - i.e. can change type associated with variable
- dynamic typing
  - i.e. types checked at run time
- ad-hoc polymorphism
  - variable can be bound to different types as program runs
- non-linear evaluation
  - programs may backtrack to unbind and rebind variables

Memory model

- database
  - holds asserted *fact*s and *rule*s
  - searched and manipulated to answer *questions*
  - may change arbitrarily during program
- stack
  - variable bindings
  - information about current position(s) in database
- heap
  - space allocated to data structures

Programs

- Prolog program consists of series of *clauses* specifying:
  - *facts*
  - *rules*
  - *questions*
- load program from file
- system will:
  - remember *fact*s & *rule*s in database
  - attempt to satisfy *question*s using *fact*s & *rule*s in database

Terms

- *clauses* made up of *terms*
- *atom*
  - words or symbols
- sequence of lower case letters, digits & _s
  - starting with a lower case letter
- sequence of characters in `‘...’`

e.g. `size top45 -- +++ fish_finger`

   `‘one hundred'`

Terms

- *integer*
- `e.g. 0 777 42 -199`
- *variable*
  - sequence of letters, digits and _s
  - starting with an upper case letter or a _
- e.g. `Cost X_11 _Property`

Terms

- *structure*
  - constructs for program and data structures

functor(*arguments*)

- *functor* – atom
- *arguments* – one or more terms separated by `,`s
- e.g. `wet(water) cost(milk,95)`
- recursive definition
  - nested structures as arguments
- e.g.
```
parents(mark,mother(margaret),
             father(dennis))
```

Terms

- *infix* structures

term atom term

- *atom* usually a symbol
- used for infix operations
- e.g. 7*8 X=99
- NB these are structures not expressions
  - * and = are symbols

Facts

- a *fact* is a *structure*
- e.g. `fly(pigs)`
- e.g.`ordered(1,3,2)`
- NB facts have no necessary meanings

Questions 1

- suppose `l16.pl` holds:

```
wet(water).

cost(milk,95).

| ?- [‘l16.pl’].

…

yes
```

- *facts* now in database

Question matching

- *question* is a *structure*
- if no *variable*s in *question* then system:
  - looks for a database *clause*
  - with the same *functor* and *arguments* as the *question*
  - displays yes or no

Question matching

- is `wet(water)` a fact?

```
| ?- wet(water).

yes
```

- try `wet(water)`
  - `water` matches `water`

Question matching

- does `milk cost 85`?

```
| ?- cost(milk,85).
no
```

- try `cost(milk,95)`
  - `milk` matches `milk`
  - `85` doesn’t match `95`

Questions with variables

- if *variable*s in *question* then system:
  - looks for a *clause* in the database with:
  - same *functor* as *question*
  - *atom*s in same *argument* positions as in *question*
  - instantiates *variable*s in *question* to *term*s in same positions in assertion
  - displays *question* variable instantiations
- use this form to search database for values in *clauses* satisfying query

Questions with variables

- for what `X` is `wet` true?

```
| ?- wet(X).
```

`X = water ?` – press `Return`

```
yes
```

- match `wet(X)`
- try `wet(water)`
  - `X` instantiated to `water`

Questions with variables

- what `X` has cost `95`?

```
| ?- cost(X,95).
X = milk ? – press Return
yes
```

- try `cost(milk,95)`
  - `cost` matches `cost`
  - `95` matches `95`
  - `X` instantiated to `milk`

Questions with variables

- what `X` has `cost Y`?

```
| ?- cost(X,Y).
X = milk
Y = 95 ? – press Return
yes
```

- try `cost(milk,95)`
  - `cost` matches `cost`
  - `X` instantiated to `milk`
  - `Y` instantiated to `95`

Multiple facts

- can have multiple *fact*s with same *functor* and different *arguments*
- *e.g.*

```
wet(water).

wet(milk).

wet(orange_juice).
```

Multiple facts

- multiple *fact*s with the same *functor* are logical *disjunctions*

functor (*argument1*).

functor (*argument2*).

...

≡

functor (*argument1*) or *functor* (*argument2*) or ...

Backtracking

- when system offers solution

1. press `Return`
   - accept solution
   - system displays `yes`
2. enter `no`
   - reject solution
   - system will *backtrack*
   - uninstantiate any *variable*s in *question*
   - try to find another *clause* matching *question*

Backtracking

`| -? wet(X)`

- match `wet(X)`
- try `wet(water)`
  - `X` instantiated to `water`

`X = water ? no`

- uninstantiate `X` from `water`
- try `wet(milk)`
  - `X` instantiated to `milk`

`X = milk ?`

Backtracking

`X = milk ? no`

- uninstantiate `X` from `milk`
- try `wet(orange_juice)`
  - `X` instantiated to `orange_juice`

`X = orange_juice ? no`

- uninstantiate `X` from `orange_juice`
- no more matches

`no`

Terms and variables

- all occurrences of a variable in a *term* are the same instance
- whenever one occurrence of a variable is bound to a value
  - all occurrences now reference that value
- e.g. `same(X,X).`
  - both `X`’s are the same variable

Matching variables

- when a *question* term with variables matches a database *term* with variables
  - variables in the same position *share*

```
| ?- same(fish,Y).
```

- match `same(X,X)`
- try `same(fish,Y)`
  - `X` instantiated to `fish`
  - `X` shares with `Y`

```
Y = fish ?
```

Rules

- *rule*s are superficially similar to methods or functions
  - fundamental differences…
- a rule has the form:

head :- *body*

- means:
  1. the *head* is true if the *body* is true
  2. the *body* implies the *head*
- *head* – *term*, usually an *atom* or *structure*
- *body* – *term*, often a *conjunction* of *term*s separated by , i.e. , behaves like logical and

Variables in rules

- all occurrences of a variable in a *term* are the same instance
- so occurrences of variables in the *head* are the same instance as occurrences in the *body*
- whenever an occurrence of a variable in the *body* is bound to a value
  - all other occurrences reference that value
  - including occurrences in the *head*
- use this to get results back from *body* of rule to *head*

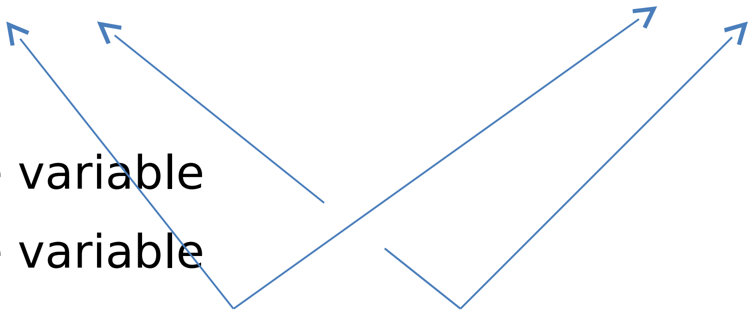
Rules

- suppose we have the *facts*:

```
mother(betty,ann).
mother(delia,betty).
```

- `X` is `Y`’s parent if they are `Y`’s mother

```
parent(X,Y) :- mother(X,Y).
```



- X’s are same variable
- Y’s are same variable

Rule matching

- to match a *rule*, try to match the *body*
- to match the *body*, try all *body* options in turn
- if matching the *body* fails:
  - backtrack, undoing any variable instantiations
  - try the next *rule* option

Rule matching

`| ?- parent(delia,P).`

- try `parent(X,Y) :- mother(X,Y)`
- `X` instantiated to `delia`
- `Y` and `P` share
  - match `mother(delia,Y)`
  - try `mother(betty,ann)`
  - `delia` does not match `betty`

Rule matching

  - backtrack
  - match `mother(delia,Y)`
  - try `mother(delia,betty)`
  - `delia` matches `delia`
  - bind `Y` to `betty`
- `P` shares with `Y` so:

```
P = betty ?
```

Multiple rules

- multiple *rule*s with the same *functor* are like logical disjunctions

functor (*arguments1*) :- *body1*.

functor (*arguments2*) :- *body2*.

...

ㅍ

functor (*arguments1*) :- *body1* or

functor (*arguments2*) :- *body2 or …*

Rules

- suppose we have the *fact*s:

```
mother(betty,ann).
mother(delia,betty).
father(chris,ann).
father(eric,betty).
```

- `X` is `Y`'s parent if they are `Y`'s mother or `Y`'s father

```
parent(X,Y) :- mother(X,Y).
parent(X,Y) :- father(X,Y).
```

Rules

`| ?- parent(P,Q).`

- try `parent(X,Y) :- mother(X,Y)`
  - `P` shares with `X`
  - `Q` shares with `Y`
  - match `mother(X,Y)`
  - try `mother(betty,ann)`
  - `X` instantiated to `betty` (shares with `P`)
  - `Y` instantiated to `ann` (shares with `Y`)

```
P = betty

Q = ann? no
```

Rules

- backtrack
- match `mother(X,Y)`
- try `mother(delia,betty)`
- `X` insantiated to `delia` (shared with `P`)
- `Y` instantiated to `betty` (shared with `Q`)

```
P = delia
Q = betty ? no
```

- backtrack

Rules

- try `parent(X,Y) :- father(X,Y)`
  - `P` shares with `X`
  - `Q` shares with `Y`
  - match `father(X,Y)`
  - try `father(chris,ann)`
  - `X` instantiated to `chris` (shared with `P`)
  - `Y` instantiated to `ann` (shared with `Q`)

```
P = chris
Q = ann? no
...
```

Rules

- if the *body* is a conjunction:

functor(*arguments*) :- *term1*,*term2*...

- *body* is equivalent to: *term1* and *term2* and ...
- to match conjunctive *body*, match each *termi* in turn
- if matching *termi* fails then backtrack to *termi-1* and try again
- NB system must remember how far each *termi* has progressed
- NB *termi* will also involve nested terms for nested rules

Rule example

- consider the train from Dundee to Aberdeen:

Dundee->Arbroath->Montrose->Stonehaven->Aberdeen

```
next(dundee,arbroath).
next(arbroath,montrose).
next(montrose,stonehaven).
next(stonehaven,aberdeen).
```

- `X` is before `Y` if `X` is next to `Y` or

  `X` is next to `W` and `W` is before `Y`

```
before(X,Y) :- next(X,Y).
before(X,Y) :- before(X,W),next(W,Y).
```

Rule example

```
| ?- before(arbroath,aberdeen).
yes
```

- try `before(arbroath,aberdeen) :- next(arbroath,aberdeen)`
  - try `next(arbroath,aberdeen)`
  - fail & backtrack
- try `before(arbroath,aberdeen) :-`

  `before(arbroath,W),next(W,aberdeen)`
  - try `before(arbroath,W),next(W,aberdeen)`
  - try `before(arbroath,W) :- next(arbroath,W)`
    - `next(arbroath,W)`
    - matches `next(arbroath,montrose)`
  - `before(arbroath,W)` succeeds with `W` instantiated to `montrose`
  - try `next(montrose,aberdeen)`
  - fail & backtrack

Rule example

- try `before(arbroath,W) :-`
- `before(arbroath,W’),next(W’,W)`
- - where `W’` is a new variable

  try `before(arbroath,W’) :- next(arbroath,W’)`
  - try `next(arbroath,W’)`
  - matches `next(arbroath,montrose)`

  `before(arbroath,W’)` succeeds with `W’` instantiated to `montrose`

  try `next(montrose,W)`
  - matches `next(montrose,stonehaven)`
- `before(arbroath,W)` succeeds with `W` instantiated to `stonehaven`
- try `next(stonehaven,aberdeen)`
  - matches `next(stonehaven,aberdeen)`
- `before(arbroath,aberdeen)` succeeds