Logic Programming

Prolog: Arithmetic, Equalities, Operators, I/O. (PLP 11)
Databases: assert, retract. (CTM 9.6)

Carlos Varela
Rensselaer Polytechnic Institute

November 21, 2023

Arithmetic Goals

```
N>M
N<M
N=<M
N>=M
```

- `N` and `M` must be bound to numbers for these tests to *succeed* or *fail*.
- `X is 1+2` is used to *assign* numeric value of right-hand-side to variable in left-hand-side.

Loop Revisited

```
natural(1).
natural(N) :-  natural(M), N is M+1.
my_loop(N) :-  N>0,
               natural(I), %generate
               write(I), nl,
               I=N,        %test
               !.
my_loop(_).
```

Also called *generate-and-test*.

= is not equal to == or =:=

`X=Y` `X\=Y`

test whether `X` and `Y` **can be** or **cannot be** *unified*.

`X==Y` `X\==Y`

test whether `X` and `Y` are currently *co-bound*, i.e., have been bound to, or share the same value.

`X=:=Y` `X=\=Y`

test *arithmetic* equality and inequality.

Can take expressions and evaluates them to a numeric value before testing. Do not bind variables.

More equalities

`X=@=Y` `X\=@=Y`

test whether `X` and `Y` are *structurally identical*.

- `=@=` is weaker than `==` but stronger than `=`.
- Examples:

```
a=@=A               false
A=@=B               true
x(A,A)=@=x(B,C)     false
x(A,A)=@=x(B,B)     true
x(A,B)=@=x(C,D)     true
```

More on equalities

`X==Y` $\Rightarrow$ `X=@=Y` $\Rightarrow$ `X=Y`

but not the other way ($\Leftarrow$).

- If two terms are currently co-bound, they are structurally identical, and therefore they can unify.
- Examples:

`a=@=A`	**false**
`A=@=B`	**true**
`x(A,A)=@=x(B,C)`	**false**
`x(A,A)=@=x(B,B)`	**true**
`x(A,B)=@=x(C,D)`	**true**

Prolog Operators

```
:- op(P,T,O)
```

declares an operator symbol `O` with precedence `P` and type `T`.

- Example:

```
:- op(500,xfx,'has_color')
a has_color red.
b has_color blue.
```

then:

```
?- b has_color C.
C = blue.
?- What has_color red.
What = a.
```

Operator precedence/type

- Precedence **P** is an integer: the larger the number, the less the precedence (*ability to group*).
- Type **T** is one of:

T	**Position**	**Associativity**	**Examples**
xfx	Infix	Non-associative	is
xfy	Infix	Right-associative	, ;
yfx	Infix	Left-associative	+ - * /
fx	Prefix	Non-associative	?-
fy	Prefix	Right-associative	
xf	Postfix	Non-associative	
yf	Postfix	Left-associative	

Testing types

`atom``(X)`

tests whether `X` is an *atom*, e.g., `'foo'`, `bar`.

`integer``(X)`

tests whether `X` is an *integer*; it does not test for complex terms, e.g., `integer(4/2)` fails.

`float``(X)`

tests whether `X` is a *float*; it matches exact type.

`string``(X)`

tests whether `X` is a *string*, enclosed in `` ... ``.

Prolog Input

`seeing``(X)`

succeeds if `X` is (or can be) bound to *current read port*.

`X = user` is keyboard (standard input.)

`see``(X)`

opens port for input file bound to `X`, and makes it *current*.

`seen`

closes current port for input file, and makes `user` *current*.

`read``(X)`

reads Prolog type expression from *current* port, storing value in `X`.

`end-of-file`

is returned by **read** at *<end-of-file>*.

Prolog Output

`telling``(X)`

succeeds if `X` is (or can be) bound to *current output port*.

`X = user` is screen (standard output.)

`tell``(X)`

opens port for output file bound to `X`, and makes it *current*.

`told`

closes current output port, and reverses to screen output (makes `user` *current*.)

`write``(X)`

writes Prolog expression bound to `X` into *current* output port.

`nl`

new line (line feed).

`tab``(N)`

writes `N` spaces to current output port.

I/O Example

```
browse(File) :-
    seeing(Old),                  /* save for later */
    see(File),                    /* open this file */
    repeat,
    read(Data),                   /* read from File */
    process(Data),
    seen,                         /* close File */
    see(Old),                     /* prev read source */
    !.                            /* stop now */

process(end_of_file) :- !.
process(Data) :-  write(Data), nl, fail.
```

First-Class Terms Revisited

`call(P)`	Invoke predicate as a goal.
`assert(P)`	Adds predicate to database.
`retract(P)`	Removes predicate from database.
`functor(T,F,A)`	Succeeds if `T` is a *term* with *functor* `F` and *arity* `A`.
`findall(F,P,L)`	Returns a list `L` with all elements `F` satisfying predicate `P`
`clause(H,B)`	Succeeds if the clause `H :- B` can be found in the database.

Databases: assert and retract

- Prolog enables direct modification of its knowledge base using assert and retract.
- Let us consider a tic-tac-toe game:

1	**2**	**3**
4	5	6
7	8	9

- We can represent a board with facts x(n) and o(n), for n in {1..9} corresponding to each player's moves.
- As a player (or the computer) moves, a fact is dynamically added to Prolog's knowledge base.

Databases: assert and retract

```
% main goal:
play :- clear, repeat, getmove, respond.

getmove :-  repeat,
            write('Please enter a move: '),
            read(X), empty(X),
            assert(o(X)).

respond :- makemove, printboard, done.

makemove :- move(X), !, assert(x(X)).
makemove :- all_full.

clear :- retractall(x(_)), retractall(o(_)).
```

Human move

Computer move

Tic-tac-toe: Strategy

The strategy is to first try to win, then try to block a win, then try to create a split (forced win in the next move), then try to prevent opponent from building three in a row, and creating a split, finally choose center, corners, and other empty squares. The order of the rules is key to implementing the strategy.

```
move(A) :- good(A), empty(A), !.

good(A) :- win(A).
good(A) :- block_win(A).
good(A) :- split(A).
good(A) :- strong_build(A).
good(A) :- weak_build(A).

good(5).
good(1).  good(3).  good(7).  good(9).
good(2).  good(4).  good(6).  good(8).
```

Tic-tac-toe: Strategy(2)

O		
	X	O
		X

- Moving x(8) produces a split: x(2) or x(7) wins in next move.

```
win(A)        :- x(B), x(C), line(A,B,C).
block_win(A):- o(B), o(C), line(A,B,C).
split(A)      :- x(B), x(C), different(B,C),
   line(A,B,D), line(A,C,E), empty(D), empty(E).
strong_build(A) :- x(B), line(A,B,C), empty(C),
                   not(risky(C)).
weak_build(A)   :- x(B), line(A,B,C), empty(C),
                   not(double_risky(C)).

risky(C)      :- o(D), line(C,D,E), empty(E).
double_risky(C) :- o(D), o(E), different(D,E),
   line(C,D,F), line(C,E,G), empty(F), empty(G).
```

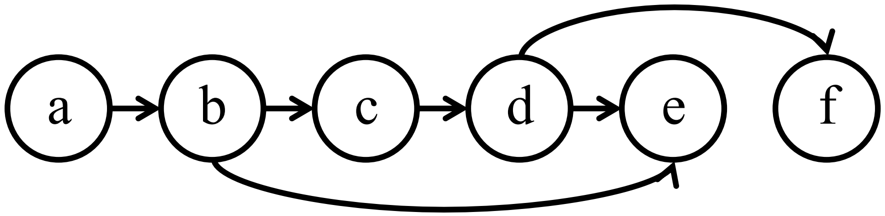
Databases in Oz: RelationClass

- Oz supports dynamic database modifications using a RelationClass. The initial relation is defined as follows:

  ```
  Rel = {New RelationClass init}
  ```

- Once we have a relation instance, the following operations are possible:
  - {Rel assert(T)} adds tuple T to Rel.
  - {Rel assertall(Ts)} adds the list of tuples Ts to Rel.
  - {Rel query(X)} binds X to one of the tuples in Rel. X can be any partial value. If more than one tuple is compatible with X, then search can enumerate all of them.

Databases in Oz: An example




```
GraphRel = {New RelationClass init}
{GraphRel assertall([edge(a b) edge(b c) edge(c d)
                     edge(d e) edge(b e) edge(d f)])}
proc {EdgeP A B} {GraphRel query(edge(A B))} end
{Browse {Search.base.all proc {$ X} {EdgeP b X} end}}
% displays all edges from node b: [c e]
```

Databases in Oz: An example(2)

```
proc {Path X Y}
  choice
    X = Y
  [] Z in
    {EdgeP Z Y}
    {Path X Z}
  end
end
{Browse {Search.base.all proc {$ X} {Path b X} end}}
% displays all nodes with a path from node b: [b c e e f d]
```

Exercises

87. The Prolog predicate my_loop/1 can succeed or fail as a goal. Explain why you may want a predicate to succeed, or to fail, depending on its expected calling context.
88. CTM Exercise 9.8.1 (page 671). Do it both in Prolog and Oz.
89. PLP Exercise 11.7 (page 571), in Oz.
90. Develop a tic-tac-toe game in Oz.