Universitas Esa Unggul

Smart, Creative and Entrepreneurial

www.esaunggul.ac.id

CMC 101 TOPIK DALAM PEMROGRAMAN

PERTEMUAN 4

PROGRAM STUDI MAGISTER ILMU KOMPUTER

FAKULTAS ILMU KOMPUTER

TOPIK DALAM PEMROGRAMAN

PEMROGRAMAN DEKLARATIF

Pertemuan 4

TUJUAN PERKULIAHAN

- Mahasiswa mampu membuat design solusi persoalan dengan paradigma deklaratif dan mampu membuat program sederhana dengan bahasa Prolog

- Abstraksi dan “dekomposisi” dalam konteks fungsional: data (type bentukan), fungsi
- Ekspresi aritmatika, logika, dan kondisional
- Analisis rekurens
- Konsep list sebagai struktur rekursif
- Operasi dasar list dengan elemen tertentu: integer, character, type bentukan

SWI-Prolog

- SWI-Prolog is a good, standard Prolog for Windows and Linux
- It's licensed under GPL, therefore free
- Downloadable from: http://www.swi-prolog.org/

Syllogisms

- "Prolog" is all about programming in logic.
- Aristotle described syllogisms 2300 years ago
- Sample syllogism:
  - Socrates is a man.
  - All men are mortal.
  - Therefore, Socrates is mortal.
- This is logic. Can Prolog do it?

Forward and backward reasoning

- A syllogism gives two premises, then asks, "What can we conclude?"
  - This is forward reasoning -- from premises to conclusions
  - it's inefficient when you have lots of premises
- Instead, you ask Prolog specific questions
  - Prolog uses backward reasoning -- from (potential) conclusions to facts

Syllogisms in Prolog

Syllogism	Prolog
Socrates is a man.	man(socrates).
All men are mortal.	mortal(X) :- man(X).
Is Socrates mortal?	?- mortal(socrates).

Facts, rules, and queries

- Fact: Socrates is a man.
- `man(socrates).`
- Rule: All men are mortal.
- `mortal(X) :- man(X).`
- Query: Is Socrates mortal?
- `mortal(socrates).`
- Queries have the same form as facts

Running Prolog I

- Create your "database" (program) in any editor
- Save it as *text only,* with a **.pl** extension
- Here's the complete program:

```
man(socrates).
mortal(X) :- man(X).
```

Running Prolog II

- Prolog is *completely interactive.* Begin by
  - Double-clicking on your `.pl` file, *or*
  - Double-clicking on the Prolog application and consulting your file at the `?-` prompt:
    - `?- consult('C:\\My Programs\\adv.pl').`
- Then, ask your question at the prompt:
  - `?- mortal(socrates).`
- Prolog responds:
  - Yes

Prolog is a theorem prover

- Prolog's "Yes" means "I can prove it" --
  Prolog's "No" means "I can't prove it"
  - ?- mortal(plato).
    No
- This is the closed world assumption: the Prolog program knows everything it needs to know
- Prolog supplies values for variables when it can
  - ?- mortal(X).
    X = socrates

Syntax I: Structures

- A structure consists of a name and zero or more arguments.
- Omit the parentheses if there are no arguments
- Example structures:
  - sunshine
  - man(socrates)
  - path(garden, south, sundial)

Syntax II: Base Clauses

- A *base clause* is just a structure, terminated with a period.
- A base clause represents a simple fact.
- Example base clauses:
  - debug_on.
  - loves(john, mary).
  - loves(mary, bill).

Syntax III: Nonbase Clauses

- A nonbase clause is a structure, a turnstile :- (meaning "if"), and a list of structures.
- Example nonbase clauses:
  - mortal(X) :- man(X).
  - mortal(X) :- woman(X).
  - happy(X) :- healthy(X), wealthy(X), wise(X).
- The comma between structures means "and"

Syntax IV: Predicates

- A *predicate* is a collection of clauses with the same *functor* (name) and *arity* (number of arguments).
- ```
  loves(john, mary).
  loves(mary, bill).
  loves(chuck, X) :- female(X), rich(X).
  ```
  ```

Syntax V: Programs

- A *program* is a collection of predicates.
- Predicates can be in any order.
- Clauses within a predicate are used in the order in which they occur.

Syntax VI: Variables and atoms

- Variables begin with a capital letter:

  `X, Socrates, _result`

- Atoms do *not* begin with a capital letter:

  `x, socrates`

- Atoms containing special characters, or beginning with a capital letter, must be enclosed in single quotes:
  - `'C:\\My Documents\\examples.pl'`

Syntax VII: Strings are atoms

- In a quoted atom, a single quote must be doubled or backslashed:
  - 'Can''t, or won\'t?'
- Backslashes in file names must also be doubled:
  - 'C:\\My Documents\\examples.pl'

Common problems

- Capitalization is *meaningful*!
- No space is allowed between a functor and its argument list:

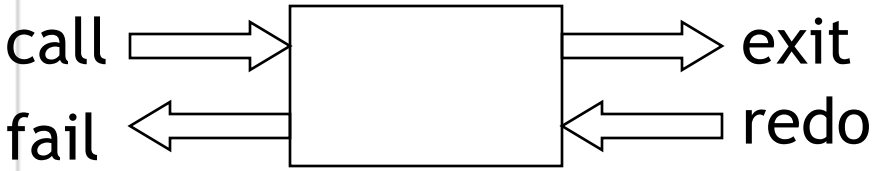
  `man(socrates)`, *not* `man (socrates).`
- Double quotes indicate a list of ASCII character values, *not* a string
- Don't forget the period! (But you can put it on the next line.)

Backtracking

- loves(chuck, X) :- female(X), rich(X).
- female(jane).
- female(mary).
- rich(mary).
- ---------- *Suppose we ask:* loves(chuck, X).
  - female(X) = female(jane), X = jane.
  - rich(jane) fails.
  - female(X) = female(mary), X = mary.
  - rich(mary) succeeds.

Backtracking and Beads

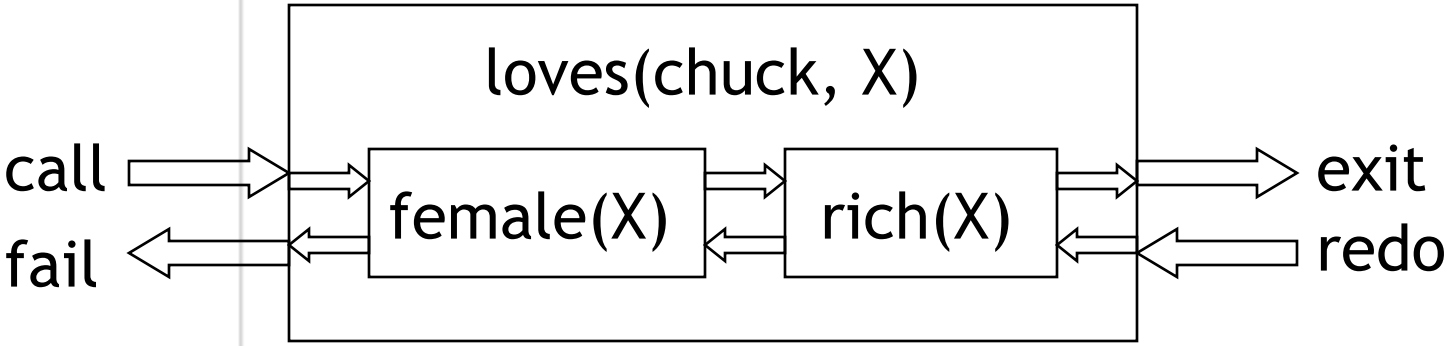
- Each Prolog call is like a "bead" in a string of beads:



- Each structure has four ports: call, exit, redo, fail
- Exit ports connect to call ports;
  fail ports connect to redo ports

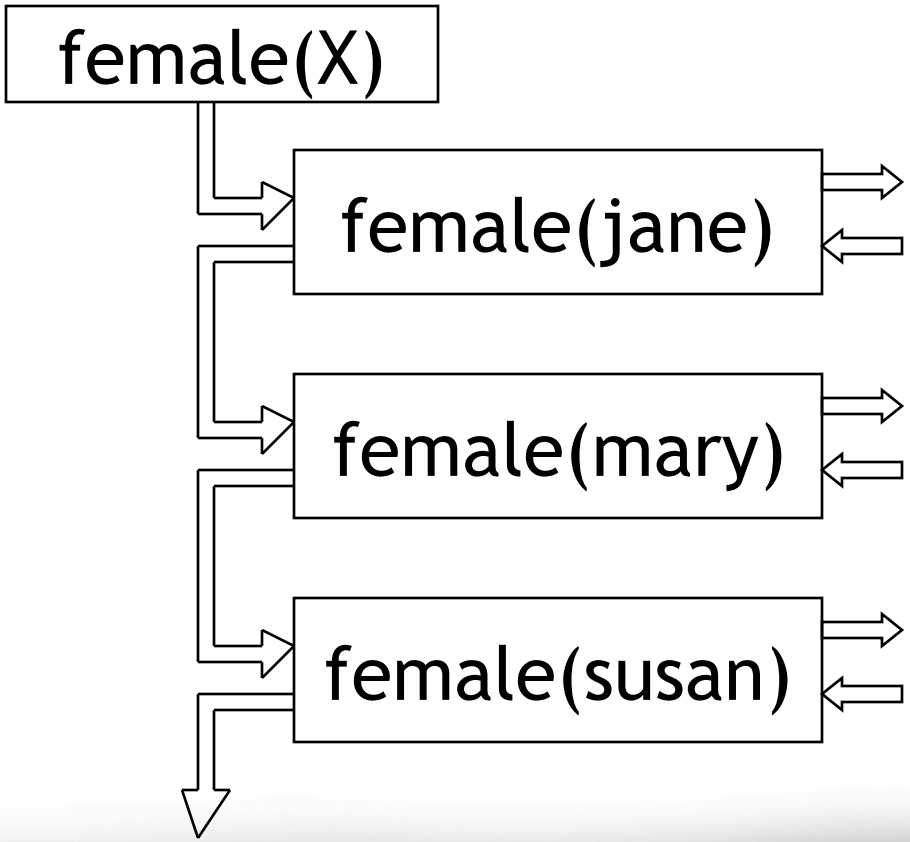
Calls as nested beads

loves(chuck, X) :- female(X), rich(X).



Additional answers

- female(jane).
  female(mary).
  female(susan).
- ?- female(X).
- X = jane ;
- X = mary
- Yes



Readings

- loves(chuck, X) :- female(X), rich(X).
- Declarative reading: Chuck loves X if X is female and rich.
- Approximate procedural reading: To find an X that Chuck loves, first find a female X, then check that X is rich.
- Declarative readings are almost always preferred.

Monotonic logic

- Standard logic is monotonic: once you prove something is true, it is true forever
- Logic isn't a good fit to reality
- If the wallet is in the purse, and the purse in is the car, we can conclude that the wallet is in the car
- But what if we take the purse out of the car?

Nonmonotonic logic

- Prolog uses nonmonotonic logic
- Facts and rules can be changed at any time
  - such facts and rules are said to be *dynamic*
- `assert(...)` adds a fact or rule
- `retract(...)` removes a fact or rule
- `assert` and `retract` are said to be *extralogical* predicates

Examples of assert and retract

- assert(man(plato)).
- assert((loves(chuck,X) :- female(X), rich(X))).
- retract(man(plato)).
- retract((loves(chuck,X) :- female(X), rich(X))).
- Notice that we use double parentheses for rules
  - this is to avoid a minor syntax problem
  - assert(foo :- bar, baz).
  - How many arguments did we give to assert?

Limitations of backtracking

- In Prolog, backtracking over something generally undoes it
- Output can't be undone by backtracking
- Neither can assert and retract be undone by backtracking
- Perform any necessary testing before you use write, nl, assert, or retract

Modeling "real life"

- Real life isn't monotonic; things change
- Prolog is superb for modeling change
- Games are often a model of real (or fantasy!) life
- Prolog is just about ideal for adventure games

Starting Prolog

- [Macintosh:~] dave% prolog
  % library(swi_hooks) compiled into pce_swi_hooks 0.00 sec, 3,928 bytes
  Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 5.10.1)
  Copyright (c) 1990-2010 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.

- ?- consult('C:_Prolog\\dragon.pl').

- % C:_Prolog\dragon.pl compiled 0.00 sec, 14,560 bytes
  Yes

Instructions

- ?- start.
- Enter commands using standard Prolog syntax.
  Available commands are:

start.	-- to start the game.
n. s. e. w.	-- to go in that direction.
take(Object).	-- to pick up an object.
drop(Object).	-- to put down an object.
use(Object).	-- to use an object.
attack.	-- to attack an enemy.
look.	-- to look around you again.
instructions.	-- to see this message again.
halt.	-- to end the game and quit.

Starting out

- You are in a meadow. To the north is the dark mouth of a cave; to the south is a small building. Your assignment, should you decide to accept it, is to recover the famed Bar-Abzad ruby and return it to this meadow.

  Yes

Going south

- ?- s.
- You are in a small building. The exit is to the north. The room is devoid of furniture, and the only feature seems to be a small door to the east.

  There is a flashlight here.

  Yes

Taking things, locked doors

- ?- take(flashlight).
- OK.

Yes

- ?- e.
- The door appears to be locked.
  You can't go that way.

Yes

Some time later...

- ?- use(key).
- The closet is no longer locked.

Yes

- Later still...
- ?- look.
- You are in a big, dark cave. The air is fetid.

There is a chest here.

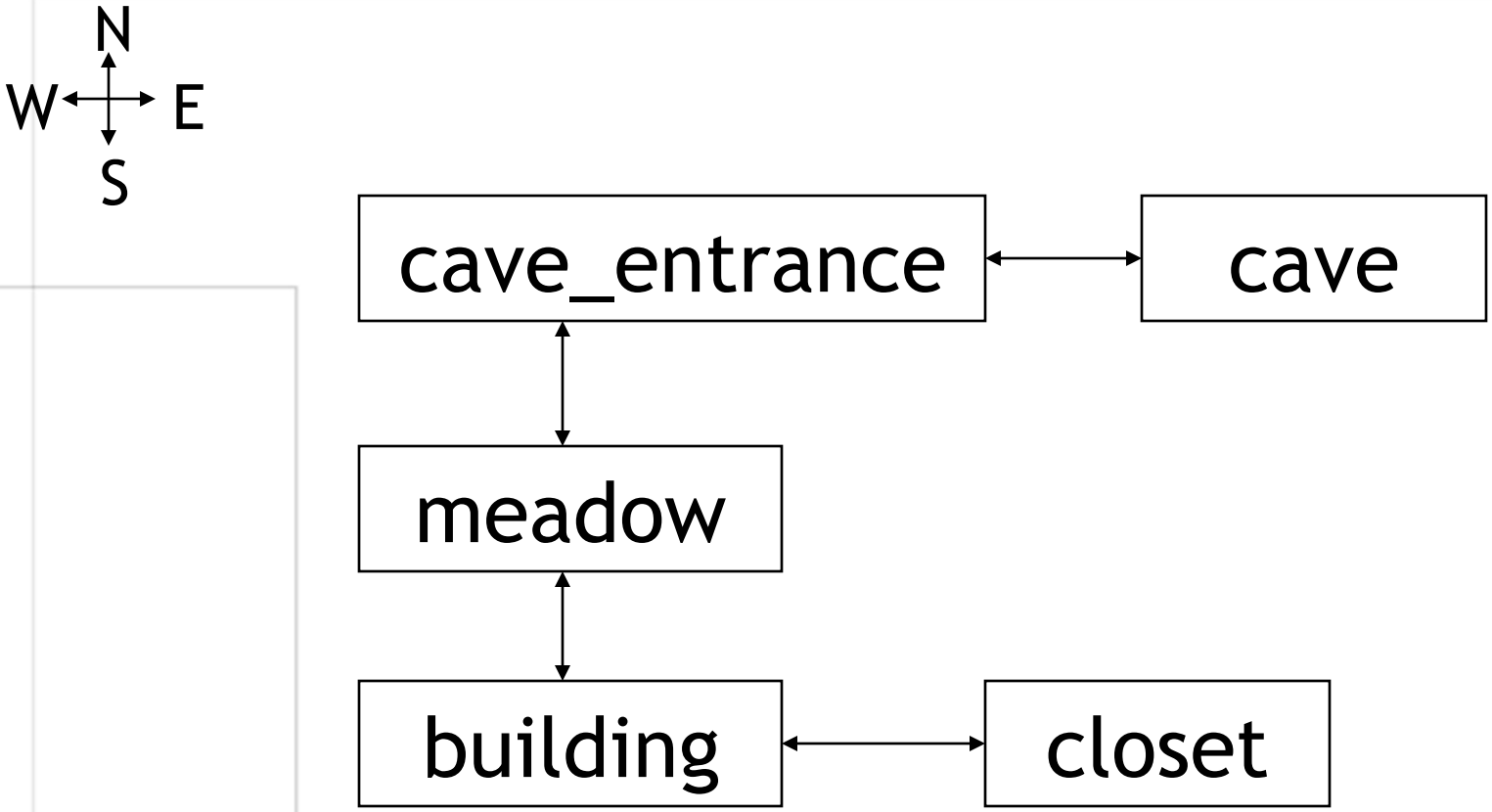
Essential facts

- Where I am at present:
  - i_am_at(meadow).
- Where other things are at:
  - at(flashlight, building).
- What I am holding:
  - holding(key).
- Which facts may be changed:
  - :- dynamic i_am_at/1, at/2, holding/1.

Input and output

- Input is unpleasant; we avoid it by giving commands (as questions) directly to Prolog
  - take(flashlight).
- write(...) outputs its *one* argument
- nl ends the line (writes a newline)
- describe(closet) :-
  write('You are in an old storage closet.'),
  nl.

The map



Implementing the map

- path(cave, w, cave_entrance).
  path(cave_entrance, e, cave).

- path(meadow, s, building).
  path(building, n, meadow).

- Could have done this instead:
  - path(cave, w, cave_entrance).
    path(X, e, Y) :- path(Y, w, X).

listing

- listing(predicate) is a good way to examine the current state of the program
- ?- listing(at).
  - at(key, cave_entrance).
    at(flashlight, building).
    at(sword, closet).

    Yes

North, south, east, west

- The commands n, s, e, w all call go.
- n :- go(n).

s :- go(s).

e :- go(e).

w :- go(w).

go

- go(Direction) :-
  i_am_at(Here),
  path(Here, Direction, There),
  retract(i_am_at(Here)),
  assert(i_am_at(There)),
  look.
- go(_) :-
  write('You can"t go that way.').

take

- take(X) :-
  i_am_at(Place),
  at(X, Place),
  retract(at(X, Place)),
  assert(holding(X)),
  write('OK.'),
  nl.

You can't always take

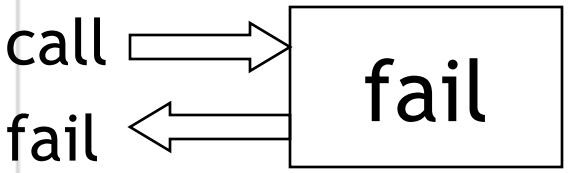
```
take(A) :-
        holding(A),
        write('You\'re already holding it!'), nl.

take(A) :- (actually take something, as before).

take(A) :-
        write('I don\'t see it here.'), nl.
```

Making things fail

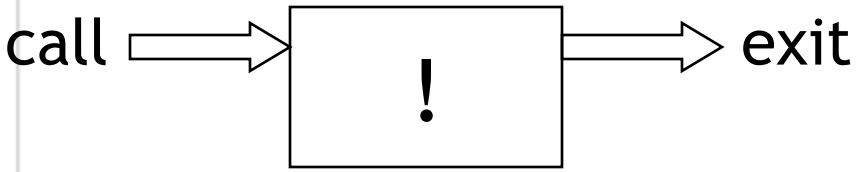
- A predicate will fail if it doesn't succeed
- You can explicitly use `fail`
- `fail` works like this:



- This often isn't strong enough; it doesn't force the entire predicate to fail

cut

- The "cut," written ! , is a *commit point*
  - It commits to the clause in which it occurs, and
  - everything before it in that clause
- Using cut says: Don't try any other clauses, and don't backtrack past the cut



cut-fail

- The cut-fail combination: `!, fail` means *really* fail
- It commits to this clause, then fails
- This means no other clauses of this predicate will be tried, so the predicate as a whole fails

A locked door

- path(building, e, closet) :-
  locked(closet),
  write('The door appears to be locked.'),
  nl,
  !, fail.
  path(building, e, closet).
- If the closet door isn't locked, the first clause fails "normally," and the second clause is used
- If the closet door *is* locked, the cut prevents the second clause from ever being reached

Dropping objects

```
drop(A) :-
        holding(A),
        i_am_at(B),
        retract(holding(A)),
        assert(at(A, B)),
        write('OK.'), nl.
drop(A) :-
        write('You aren\'t holding it!'), nl.
```

What else is Prolog good for?

- Prolog is primarily an AI (Artificial Intelligence) language
- It's second only to LISP in popularity
- It's more popular in Britain than in the U.S.
- Prolog is also a very enjoyable language in which to program (subjective opinion, obviously!)

Prolog vs. LISP

- Unlike LISP, Prolog provides:
  - built-in theorem proving
  - built in Definite Clause Grammars, good for parsing natural language
- If you just want to use these tools, Prolog is arguably better
- If you want to build your own theorem prover or parser, LISP is clearly better

The End