September
1st Session 2014/2015 (A141)

Wan Hussain Wan Ishak

School of Computing
UUM College of Arts and Sciences
Universiti Utara Malaysia

(P) 04-9285150
(E) hussain@uum.edu.my
(U) http://wanhussain.com
Lecture notes

Expressing Procedural Algorithm

- Conditional execution
- “if-then-else” structure
- repetition through backtracking
- recursion
Procedural Prolog

- Prolog combines procedural and non-procedural programming techniques.

- Prolog’s control strategy – based on simple depth-first search.
Conditional Execution

- Prolog procedures can have multiple definitions (clauses) – each applying under different conditions.

- Conditional execution (*if* or *case* statements) – expressed with alternative definitions of procedures.
Conditional Execution

- Example - Java switch/case statement

```java
public static void printNum(int X) {
    switch(X) {
        case 1:
            System.out.println(" One");
        case 2:
            System.out.println(" Two");
        case 3:
            System.out.println(" Three");
    }
}
```
Conditional Execution

In Prolog, `printNum` has three definitions:

```
printNum(1):- write('One').
printNum(2):- write('Two').
printNum(3):- write('Three').
```
Conditional Execution

Common mistakes - inefficient:

- printNum(X):- X=1, write(‘One’).
- printNum(X):- X=2, write(‘Two’).
- printNum(X):- X=3, write(‘Three’).

Gives correct results but waste time - Execute each clause, perform test, and backtrack out.
Conditional Execution

- Effective programming in Prolog:
  - Make each logical unit of the program into a separate procedure.
  - Each **if** or **case** statement should become a procedure call – decisions are made by procedure-calling process – choosing the right clause.
Conditional Execution

Example

Pascal

```pascal
procedure a(X:integer);
begin
  b;
  if X=0 then c else d;
  e
end;
a(X):- b,
    cd(X),
e.
cd(0):- c.
cd(X):- \+ X = 0, d.
```

Prolog
The “IF-THEN-ELSE” Structure

- Can be implemented in Prolog as:

  \[ \text{Goal1} \rightarrow \text{Goal2} ; \text{Goal3} \]

  “if Goal1 then Goal2 else Goal 3”

- Meaning:
  Test whether Goal1 succeeds, and if so, execute Goal2, otherwise execute Goal3.
The “IF-THEN-ELSE” Structure

Example (simple if-then-else):

```
writeNum(X):- X=1 -> write(one) ; write(‘Not one’).
```

Meaning:
If $X = 1$ the ‘One’ will be written, if not (else) ‘Not one’ will be written.
The “IF-THEN-ELSE” Structure

Example (nested if-then-else):

\[\text{writeNum}(X) :-
\]
\[
\begin{align*}
( & X=1 \rightarrow \text{write(one)} \\
& ; X=2 \rightarrow \text{write(two)} \\
& ; X=3 \rightarrow \text{write(three)} \\
& ; \text{write(‘out of range’)}
\end{align*}
\]
The “IF-THEN-ELSE” Structure

- If-then-else structure – for making decision without calling procedures.

- Discouraged
  - Looks like ordinary structured programming
  - Prolog clauses are supposed to be logical formulas.
Backtracking

- Prolog will automatically backtrack – for satisfying a goal.

- In console, Prolog will backtrack automatically after we press "\".

- To force backtracking use fail/0.
Controlling Backtracking

- Uncontrolled backtracking may cause inefficiency in a program.
- Control using ‘cut’ facility.
- The symbol is ‘!’.
- Function – prevent backtracking.
- Useful – relieves the programmer of the burden of programming backtracking explicitly.
Controlling Backtracking

Example (Pascal)

```pascal
procedure writename(X:integer);
begin
  case X of
    1: write('one');
    2: write('two');
    3: write('three');
    else
      write('out of range')
  end
end;
```

```prolog
writename(1):- write('one').
writename(2):- write('two').
writename(3):- write('three').
writename(X):- X < 1, write('out of range').
writename(X):- X > 3, write('out of range').
```

Correct but lack of conciseness
Controlling Backtracking

\[
\begin{align*}
\text{writename}(1): & \quad \text{write(‘one’).} \\
\text{writename}(2): & \quad \text{write(‘two’).} \\
\text{writename}(3): & \quad \text{write(‘three’).} \\
\text{writename}(X): & \quad X < 1, \\
& \quad \text{write(‘out of range’).} \\
\text{writename}(X): & \quad X > 3, \\
& \quad \text{write(‘out of range’).}
\end{align*}
\]

Can be re-written as

\[
\begin{align*}
\text{writename}(1): & \quad \text{write(‘one’).} \\
\text{writename}(2): & \quad \text{write(‘two’).} \\
\text{writename}(3): & \quad \text{write(‘three’).} \\
\text{writename}(X): & \quad X < 1, \\
& \quad \text{write(‘out of range’).} \\
\text{writename}(X): & \quad X > 3, \\
& \quad \text{write(‘out of range’).} \\
\text{writename}(\_): & \quad \text{write(‘out of range’).}
\end{align*}
\]
Controlling Backtracking

Example: (Prolog)

```prolog
writename(1):- write('one').
writename(2):- write('two').
writename(3):- write('three').
writename(_):- write('out of range').
```

Because, anonymous variable is used in the last clause. This variable will match with any value.
Excellent…. So clever….

Controlling Backtracking

Because of backtracking!

Ok… referring to our problem. We want to display only one solution, right!

So, we have to state that when any clauses has succeed then the computer need to stop looking for other alternatives.

How is that?

We can use a special operator called cut.

The symbol is '!' symbol “!”

Cut “!” operator will tell the computer to ignore other alternatives.

Meaning that, it will prevent backtracking …
Controlling Backtracking

- Example: (Prolog)

\[ \text{b:- c, d, !, e, f.} \]
\[ \text{b:- g, h.} \]

I don’t understand...

Now, look at this example
Controlling Backtracking

Example: (Prolog)

\[
\begin{align*}
\text{b} & : - \ c, \ d, \ !, \ e, \ f. \\
\text{b} & : - \ g, \ h. \\
\end{align*}
\]

Why??? Prolog will automatically backtrack to the second rule, right?

Correct... but in this case its different. Cut “!” will prevent Prolog from backtrack. So, there is no attempt to look for the second alternative.
Controlling Backtracking

So, I guess this example can be modified this way, right?

```prolog
writename(1):- write('one').
writename(2):- write('two').
writename(3):- write('three').
writename(_):- write('out of range').
```

```prolog
writename(1):- !, write('one').
writename(2):- !, write('two').
writename(3):- !, write('three').
writename(_):- write('out of range').
```

Fast learner... excellent...
Controlling Backtracking - Discussion

Given that:

\[ \text{max}(X, Y, \text{Max}). \]

Where Max = X if X is greater than or equal to Y, and Max = Y is X is less than Y.

\[ \text{max}(X, Y, X) :- \ X \geq Y. \]
\[ \text{max}(X, Y, Y) :- \ X \lt Y. \]
Controlling Backtracking - Discussion

max(X,Y,X):- X>=Y.
max(X,Y,Y):- X<Y.

These rules are mutually exclusive.

- If the first one succeeds then the second one will fail.
- If the first one fails then the second must succeed.
Controlling Backtracking - Discussion

More economical formulation

Instead of:

if \( X \geq Y \) then \( \text{Max} = X \)
if \( X < Y \) then \( \text{Max} = Y \)

max(\( X,Y,X \)):-\( X \geq Y \).
max(\( X,Y,Y \)):-\( X < Y \).
Making a goal deterministic without cuts

Instead of creating deterministic predicates, we can define nondeterministic predicates in the ordinary manner and then block backtracking when we call them.

Special built-in predicate once/1.

To define once/1 as:

\[
\text{once(Goal):- call(Goal), !.}
\]
Making a goal deterministic without cuts

Example:

?- food(A).
A = rice ;
A = ice_cream ;
A = banana

?- once(food(A)).
A = rice

Facts:

food(rice).
food(ice_cream).
food(banana).

All alternatives were taken out

Only the first fact return.
In order to control the program flow, there is a need:

- to guarantee that a goal will succeed - regardless of the results.
- to guarantee that a goal will always fail.
Goal Always Succeed or Always Fail

- Always succeed:
  - Used true/0.
  - Example:
    ```prolog
    ?- eat(ahmad,fish); true.
    ```

- Always fail:
  - Used fail/0.
  - Example:
    ```prolog
    writeNum(X):- X> 0, write('more than 0'), fail.
    ```
Recursion

- A procedure that calling itself to perform the tasks inside its tasks until the stopping condition is reached.

- Must have at least two clauses:
  - Basic clause – to stop the recursion.
  - Recursive clause – the one that call and reference to itself.
Recursion

Example

display_num(0).

display_num(X):-
    write(X),
    NewX is X - 1,
    display_num(NewX).

Basic clause

Recursive clause
Rule

Example

\[
\text{is_in}(X,Y) :- \\
\text{in}(X,T), \\
\text{in}(T,Y).
\]

\[
\text{in}\text{(city_plaza, alor_star)}. \\
in\text{(alor_star, kedah)}. \\
? \rightarrow \text{is_in}\text{(city_plaza, kedah)}. \\
is\text{_in}(X,Y) :- \\
in\text{(X,T)}, \\
in\text{(T,Y)}. \\
? \rightarrow \text{is_in}\text{(city_plaza, malaysia)}. \\
is\text{_in}(X,Y) :- \\
in\text{(X,T)}, \\
in\text{(T,T2)}, \\
in\text{(T2,Y)}.
\]
Rule

Example

\[
\text{in(city_plaza, alor_star).} \\
\text{in(alor_star, kedah).} \\
\text{in(kedah, malaysia).} \\
\text{in(malaysia, south_east).} \\
\text{in(south_east, asia).} \\
\text{in(asia, world).} \\
\]

? – is_in(city_plaza, world).

is_in(X,Y):-
  \text{in(X,Y).}

is_in(X,Y):-
  \text{in(X,T), in(T,Y).}

is_in(X,Y):-
  \text{in(X,T), in(T,T2), in(T2,T3).} \\
  \text{in(T3, T4), ...}

Recursive rule
Recursive Rule
Recursive Rule

Example

\[
\text{is\_in}(X,Y) :\ - \ 
\begin{align*}
\text{in}(X,Y) & . \\
\text{is\_in}(X,Y) : & - \ 
\begin{align*}
\text{in}(X,T) & , \\
\text{is\_in}(T,Y) & .
\end{align*}
\end{align*}
\]

\[
\begin{align*}
\text{in}(\text{city\_plaza}, \text{alar\_star}) & . \\
\text{in}(\text{alar\_star}, \text{kedah}) & . \\
\text{in}(\text{kedah}, \text{malaysia}) & . \\
\text{in}(\text{malaysia}, \text{south\_east}) & . \\
\text{in}(\text{asia\_south\_east}, \text{asia}) & . \\
\text{in}(\text{asia}, \text{world}) & .
\end{align*}
\]
Recursive Rule

Example (step-by-step)

? – is_in(city_plaza, world).

```prolog
is_in(X,Y):-
   in(X,Y).

is_in(X,Y):-
   in(X,T),
   is_in(T,Y).
```

Working memory:

X = city_plaza
Y = world

* Fact in(city_plaza, world) not exist
Recursive Rule

Example (step-by-step)

? – is_in(city_plaza, world).

\[
\begin{align*}
is_{\text{in}}(X,Y) : & - \\
\text{in}(X,Y).
\end{align*}
\]

\[
\begin{align*}
is_{\text{in}}(X,Y) : & - \\
\text{in}(X,T), & \text{is}_{\text{in}}(T,Y).
\end{align*}
\]

\[
\begin{align*}
in(\text{city}_\text{plaza}, \text{alar}_\text{star}). \\
in(\text{alar}_\text{star}, \text{kedah}). \\
in(\text{kedah}, \text{malaysia}). \\
in(\text{malaysia}, \text{south}_\text{east}). \\
in(\text{south}_\text{east}, \text{asia}). \\
in(\text{asia}, \text{world}).
\end{align*}
\]

Working memory:

\[
\begin{align*}
X & = \text{city}_\text{plaza} \\
Y & = \text{world} \\
T & = \text{alar}_\text{star}
\end{align*}
\]

Call is_in(alar_star, world)
Recursive Rule

Example (step-by-step)

? – is_in(city_plaza, world).

is_in(X,Y):-
    in(X,Y).

is_in(X,Y):-
    in(X,T),
    is_in(T,Y).

in(city_plaza, alor_star).
in(alor_star, kedah).
in(kedah, malaysia).
in(malaysia, south_east).
in(south_east, asia).
in(asia, world).

Working memory:

X = alor_star
Y = world

* Fact in(alor_star, world) not exist
Recursive Rule

Example (step-by-step)

\(? - is\_in(city\_plaza, world).\)

```prolog
is\_in(X,Y):-
    in(X,Y).
```

```prolog
is\_in(X,Y):-
    in(X,T),
    is\_in(T,Y).
```

```
in(city\_plaza, alor\_star).
in(alor\_star, kedah).
in(kedah, malaysia).
in(malaysia, south\_east).
in(south\_east, asia).
in(asia, world).
```

**Working memory:**

- X = alor\_star
- Y = world
- T = kedah

Call `is\_in(kedah, world)`