Overview: Representation Techniques

Week 6

- Representations for classical planning problems
  - deterministic environment; complete information

Week 7

- Logic programs for problem representations
  - including planning problems, games

Week 8

- First-order logic to describe dynamic environments
  - deterministic environment; (in-)complete information

Week 9

- State transition systems to describe dynamic environments
  - nondeterministic environment; (in-)complete information

KR for Reasoning About Actions

Many KR formalisms exist for representing knowledge of actions:

Situation calculus, event calculus, fluent calculus, dynamic logic, causal networks, …

- **Situation Calculus** (1967) – historically first formalism for
  - representing **and reasoning** about action knowledge in logic
  - planning; well before the first special-purpose planning languages
  - GOLOG – a language that combines programming + reasoning about actions

All existing formalisms for reasoning about actions, including GDL and last week's use of ASP for planning, have their roots in the Situation Calculus

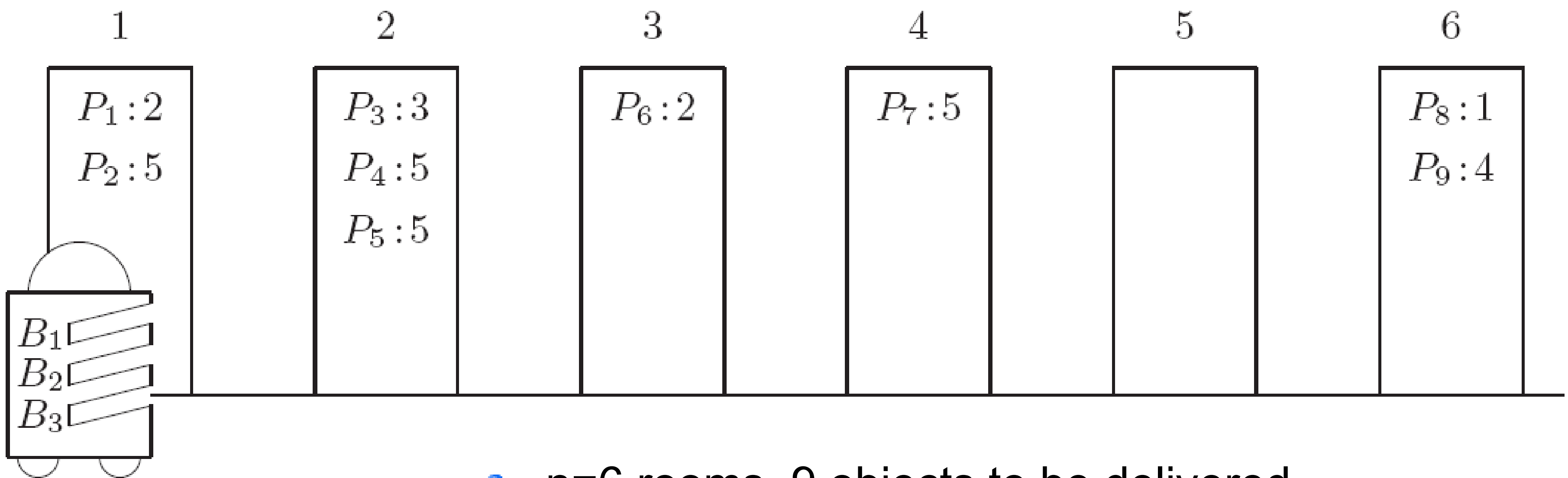
Reasoning About Actions: Overview

- Foundations: Situation calculus for representing actions in logic
- GOLOG – a Situation Calculus-based control language for agents/robots

Background reading

Knowledge in Action by Raymond Reiter, MIT Press 2001.
Chapters 3, 5, 6, 11

Example: A Delivery Robot



- n=6 rooms, 9 objects to be delivered
- Robot can carry at most k=3 objects at a time

State predicates
At(room)
Empty(b)
Carries(b,p,r)
Request(p,r,r')

$b = B_1, B_2, B_3$
$p = P_1, \ldots, P_9$
$r, r' = 1, \ldots, 6$

Actions
Go(up)
Go(down)
Pick(p,b)
Drop(b)

Situation Calculus

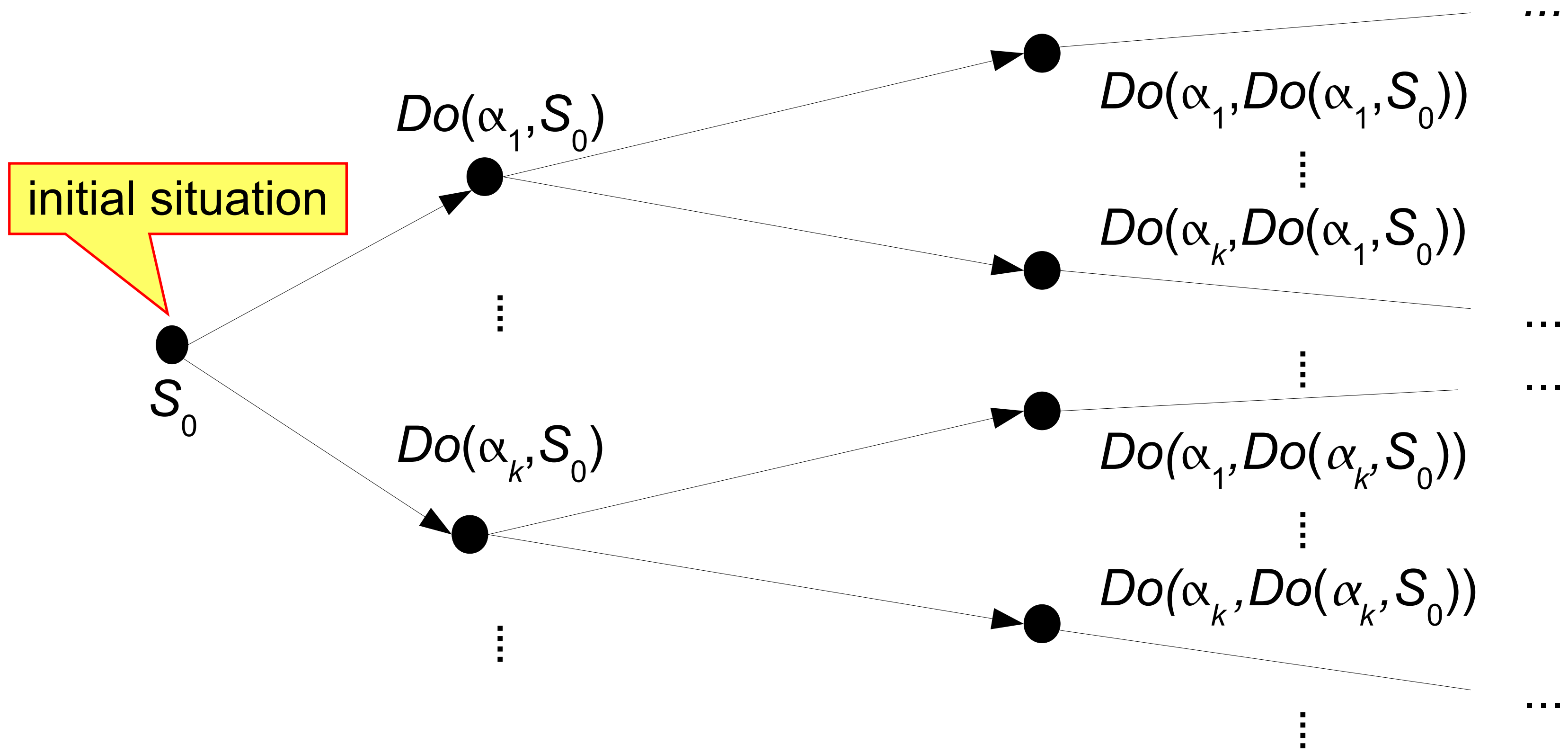
Situation Calculus: Logical Foundations

Situation = sequence of actions, written as a logical term

$Do(Pick(P_1,B_1),S_0)$

$Do(Pick(P_2,B_2),Do(Pick(P_1,B_1),S_0))$

$Do(Drop(B_1),Do(Go(Up),Do(Pick(P_2,B_2),Do(Pick(P_1,B_1),S_0))))$



Situation Calculus: Initial State Axioms

- Add **situation argument** to all state predicates (aka **fluents**)
- **Initial state axioms** specify properties of S_0 (initial situation)

$At(1,S_0) \wedge$

$Empty(B_1,S_0) \wedge Empty(B_2,S_0) \wedge Empty(B_3,S_0) \wedge$

$Request(P_1,1,2,S_0) \wedge \ldots \wedge Request(P_9,6,4,S_0)$

fluent, with situation argument

Situation Calculus: Precondition Axioms

- Special predicate: Poss(a,s) – read: "action a is possible in situation s"
- General form of **precondition axioms**:

$$Poss(a(\vec{y}),s) \leftrightarrow \gamma\,[a, s, \vec{y}\,]$$

All free variables are assumed to be universally quantified

$Poss(Go(d), s) \leftrightarrow (\exists r)(At\,(r, s) \wedge [d = Up \wedge r < 6 \vee d = Down \wedge r > 1])$

$Poss(Pick(p, b), s) \leftrightarrow (\exists r, r')(At\,(r, s) \wedge Request\,(p, r, r', s) \wedge Empty(b, s))$

$Poss(Drop(b), s) \leftrightarrow (\exists p, r)(At\,(r, s) \wedge Carries(b, p, r, s))$

- Can be used to reason logically about the executability of actions

Example: From $At(1,S_0) \wedge Empty(B_1,S_0) \wedge Empty(B_2,S_0) \wedge Empty(B_3,S_0)$
$\wedge\, Request\,(P_1,1,2,S_0) \wedge \ldots \wedge Request\,(P_9,6,4,S_0)$

it follows that $Poss(Pick(P_1,B_1),S_0)$ but not $Poss(Go(Down),S_0)$

Situation Calculus: The Frame Problem

- Effect formulas alone not enough to draw conclusions about what doesn't change

Example.

effect formula

$Poss(Go(Up),s) \rightarrow$
$At(r,s) \rightarrow \neg At(r,Do(Go(Up),s)) \wedge At(r+1,Do(Go(Up),s))$

Suppose given $At(1,S_0) \wedge Empty(B_1,S_0) \wedge Request(P_1,2,1,S_0)$

The effect axiom with $\{r / 1, s / S_0\}$ entails
$\neg At(1,Do(Go(Up),S_0)) \wedge At(2,Do(Go(Up),S_0))$
but **not** $Empty(B_1,Do(Go(Up),S_0)) \wedge Request(P_1,2,1,Do(Go(Up),S_0))$

- **Frame Problem (1969)**: How can we succinctly specify actions in logic so that both effects and non-effects follow?

Situation Calculus: Successor State Axioms

- Solution (1991) to the frame problem:
  There is a **successor state axiom** for all state propositions *F*

$$F(\vec{y}, Do(a, s)) \leftrightarrow \gamma_F^+[a, s, \vec{y}] \vee (F(\vec{y}, s) \wedge \neg\gamma_F^-[a, s, \vec{y}])$$

where

  - γ_F^+ describes the conditions under which $F(\vec{y})$ is positive effect
  - γ_F^- describes the conditions under which $F(\vec{y})$ is negative effect

- Example: $Empty(b, Do(a,s)) \leftrightarrow a = Drop(b) \vee (Empty(b,s) \wedge \neg(\exists p)\, a = Pick(p,b))$

Remember: all free variables are universally quantified

Successor State Axioms for the Delivery Robot

- 4 successor state axioms, one for each fluent

$At(r,Do(a,s)) \leftrightarrow (At(r-1, s) \wedge a = Go(Up)) \vee (At(r+1, s) \wedge a = Go(Down)) \vee$
$(At(r, s) \wedge \neg(\exists d)\ a = Go(d))$

$Request(p,r1,r2,Do(a,s)) \leftrightarrow Request(p,r1,r2,s) \wedge \neg(\exists b)\ a = Pick(p,b)$

$Empty(b,Do(a,s)) \leftrightarrow a = Drop(b) \vee$
$(Empty(b,s) \wedge \neg(\exists p)\ a = Pick(p,b))$

$Carries(b,p,Do(a,s)) \leftrightarrow a = Pick(p,b) \vee$
$(Carries(p,b,s) \wedge \neg(a = Drop(b))$

- Requires **unique-name assumption** for actions and objects (why?)
  e.g. $(\forall b,d,p)\ \neg(Go(d) = Pick(p,b))$ $\neg B_1{=}B_2$
- Axioms can be used to reason logically about the effects of actions

  Example. From $Empty(B_1,S_0)$ infer $Empty(B_1,Do(Go(Up),S_0))$

Sitcalc in Prolog: Precondition Axioms

```
at(1,s0).
empty(b1,s0).   …   empty(b3,s0).
request(p1,1,2,s0).   …   request(p9,6,4,s0).

poss(go(up),S)     :- at(R,S), R<6.
poss(go(down),S)   :- at(R,S), R>1.
poss(pick(P,B),S) :- at(R,S), request(P,R,R1,S), empty(B,S).
poss(drop(B),S)    :- at(R,S), carries(B,P,R,S).
```

```
?- poss(pick(Parcel,B),s0)
   Parcel = p1, B = b1    More (y/n)? y
   Parcel = p1, B = b2    …

?- poss(go(down),s0)
   no
```


Sitcalc in Prolog: Successor State Axioms

```
at(R,do(A,S)) :- at(R1,S), R is R1+1, A=go(up).
at(R,do(A,S)) :- at(R1,S), R is R1-1, A=go(down).
at(R,do(A,S)) :- at(R,S), not A=go(D).

request(P,R1,R2,do(A,S)) :- request(P,R1,R2,S),
                            not A=pick(P,B).

empty(B,do(A,S))          :- A=drop(B).
empty(B,do(A,S))          :- empty(B,S), not A=pick(P,B).

carries(B,P,R,do(A,S))  :- A=pick(P,B), request(P,R1,R,S).
carries(B,P,R,do(A,S))  :- carries(B,P,R,S), not A=drop(B).
```


Sitcalc in Prolog: The Regression Principle

- **Regression** means to answer a query with a fully instantiated action sequence by
  - rolling back the situation action-by-action
    - through the successor state axioms
      - to the initial situation

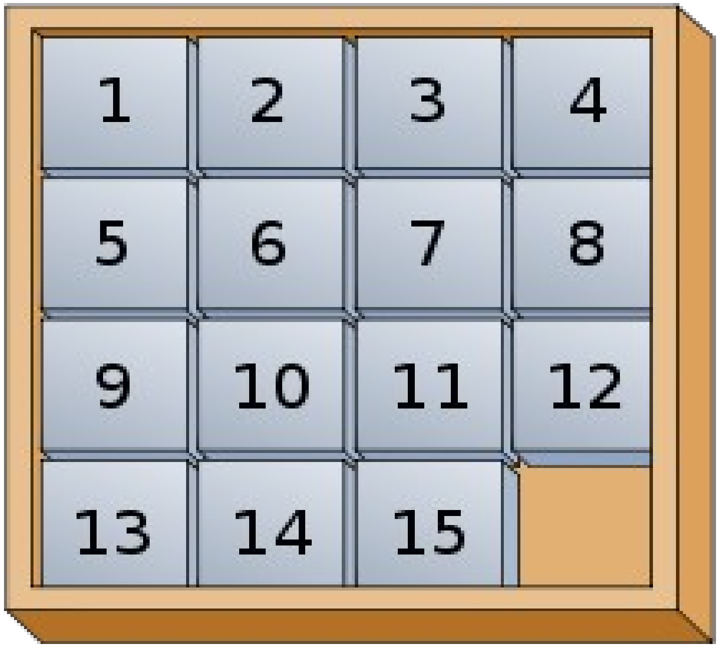
```
carries(B,P,R,do(A,S))  :- A=pick(P,B), request(P,R1,R,S).
carries(B,P,R,do(A,S))  :- carries(B,P,R,S), not A=drop(B).
```

```
   ?-  carries(b1,p1,2, do(go(up),do(pick(p1,b1),s0)))

⇒  ?-  carries(b1,p1,2, do(pick(p1,b1),s0)),
       not go(up)=drop(b1)

⇒  ?-  pick(p1,b1)=pick(p1,b1),
       request(p1,R1,2, s0),
       not go(up)=drop(b1)

⇒  R1=1
```

Exercise

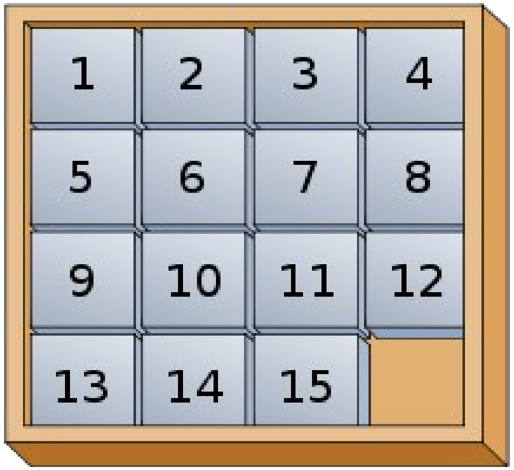
Exercise

- Constant symbols:
  - The tiles: 1, 2, 3, …, 15, blank
  - The coordinates: 1, 2, 3, 4
- Fluent:
  - *Cell*(coord,coord,tile,s)
- Action:
  - *Move*(x-from,y-from,x-to,y-to)



Precondition axiom for Move(x,y,x',y')?

Exercise

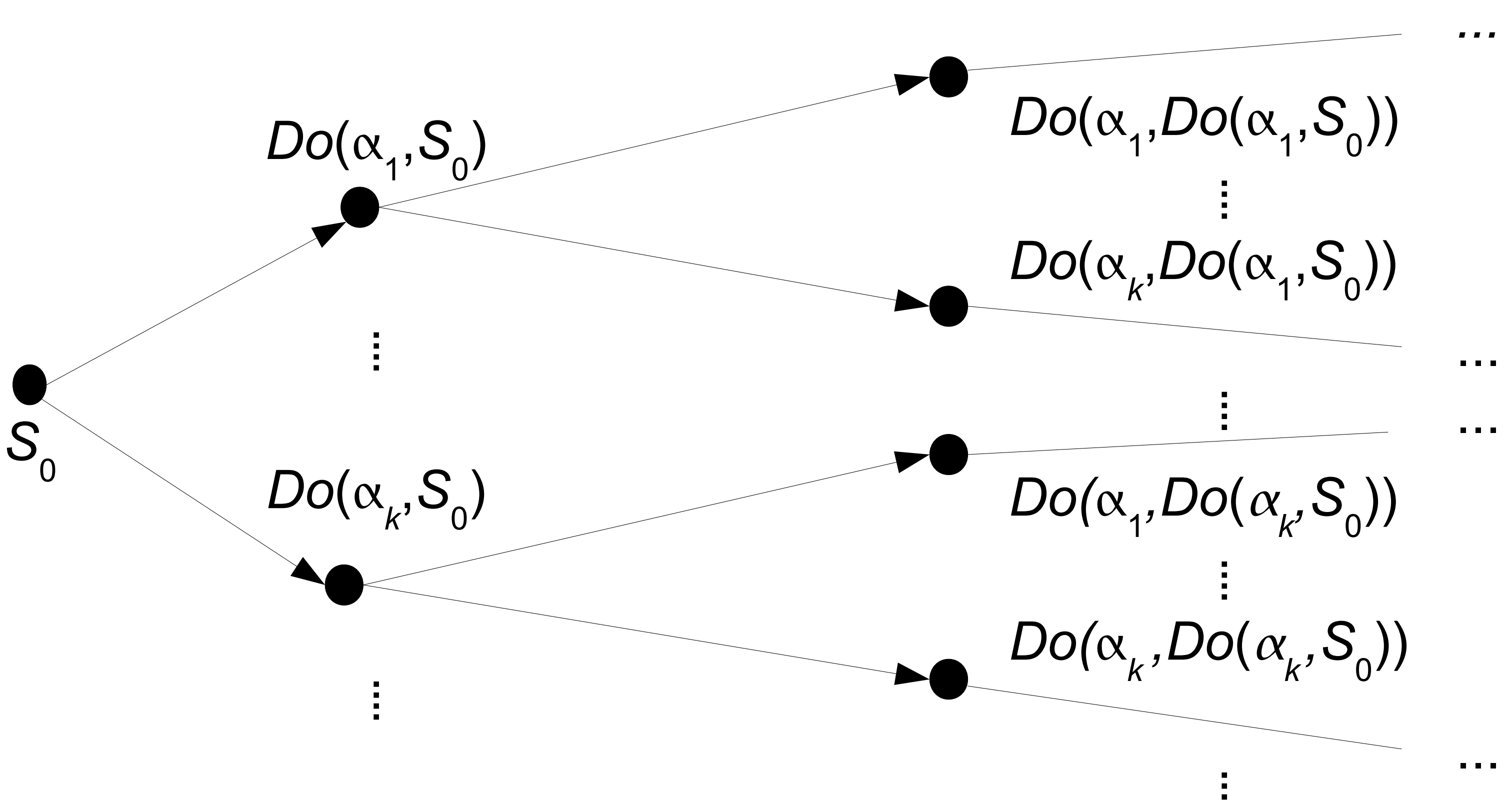


Successor state axiom for Cell(x,y,z,s)?

GOLOG —
agent/robot programming language based on Situation Calclus

GOLOG and the Situation Tree

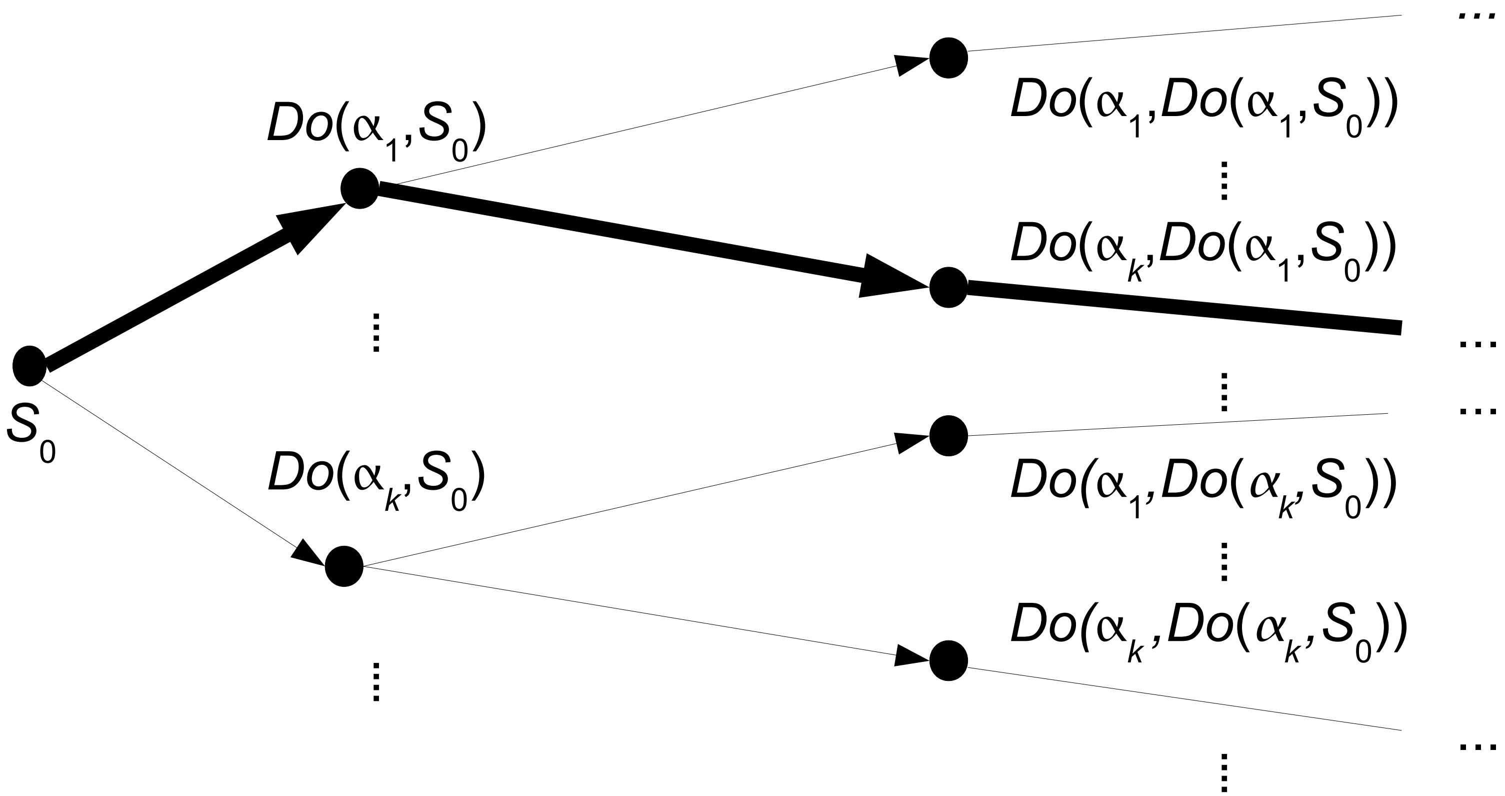
- GOLOG = Al<u>gol</u> in <u>Logic</u>
- Purpose of GOLOG programs:
  Define a **strategy** (= a tree **skeleton**) to reduce the search space



Extreme Case 1: Fixed Sequence

- Sample GOLOG program:

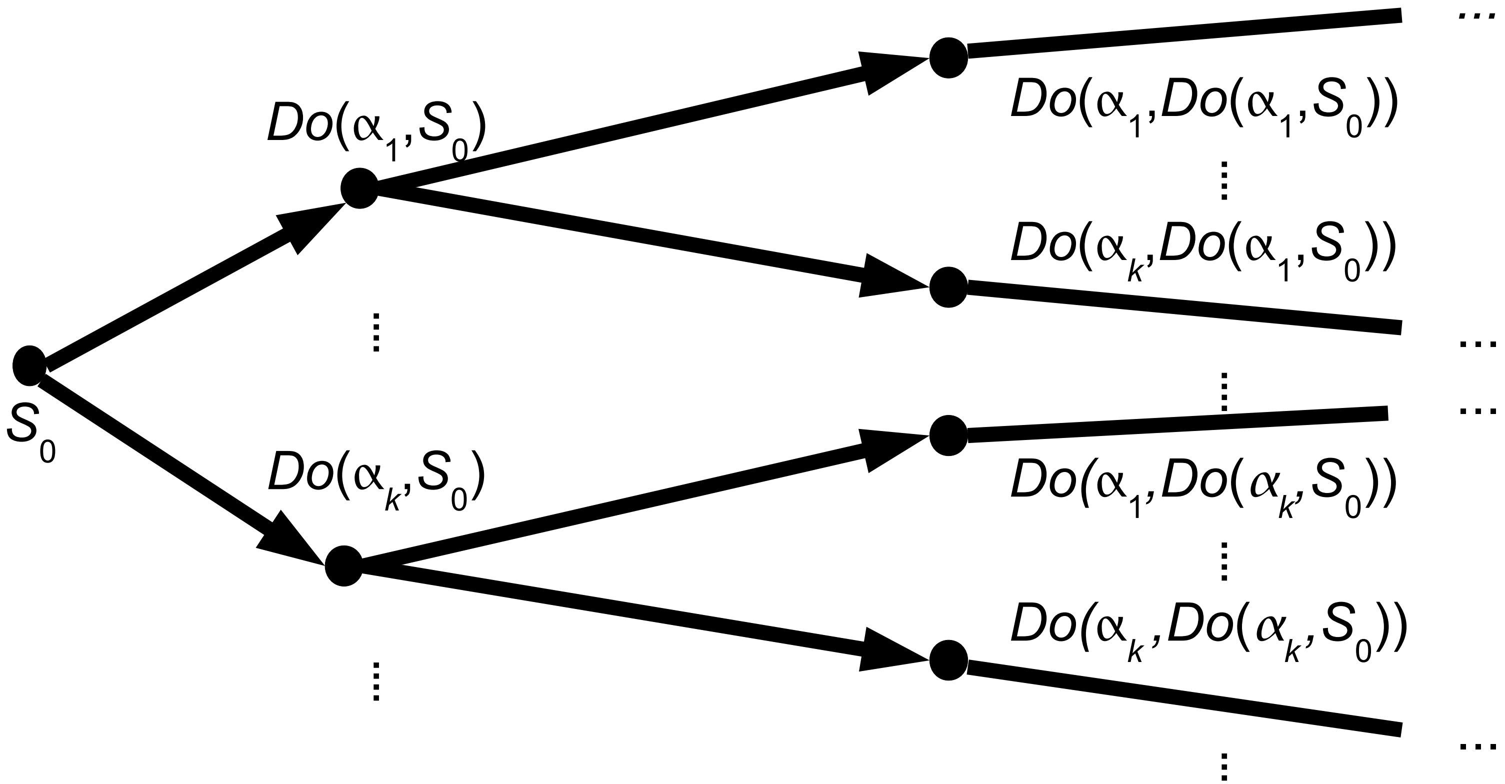
$\alpha_1; \alpha_k; \alpha_2; \ldots$



Extreme Case 2: Entire Tree

- Sample GOLOG program:

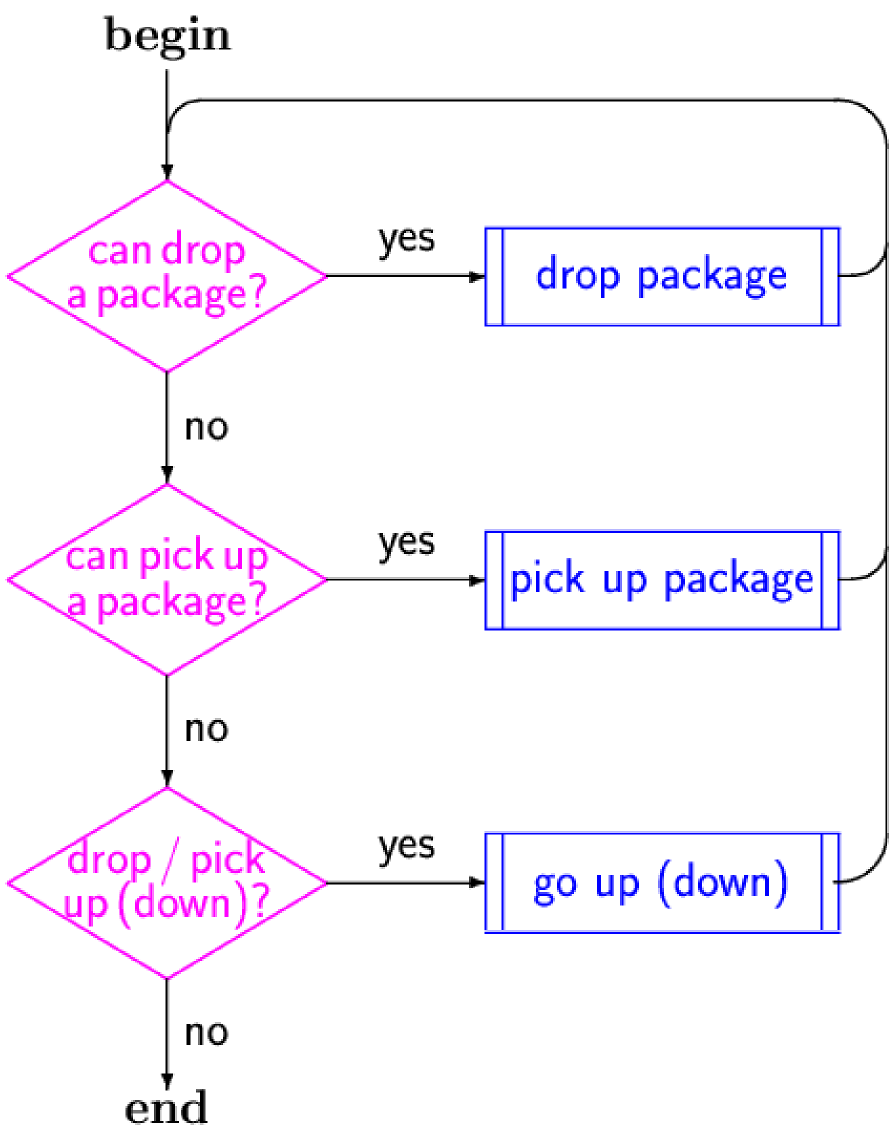
  $(\; \alpha_1 \mid \alpha_2 \mid \; \ldots \; \mid \alpha_k \;)^*$



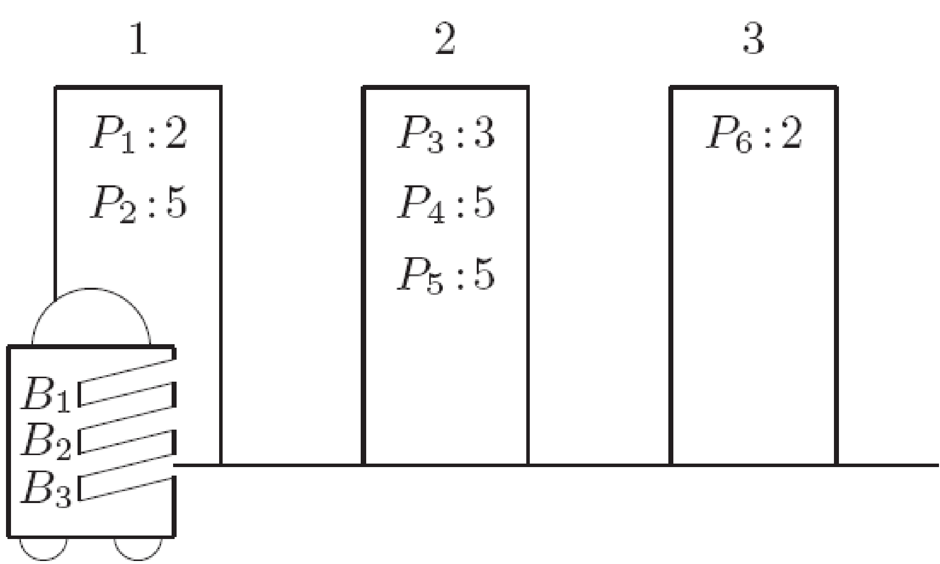
Example: iCinema's Scenario

- GOLOG used to program the behaviour of the virtual characters

A Simple Algorithm for the Mailbot



- Easier than trying to solve optimally as planning problem
- Especially in an environment
  - with >10 rooms and 100s of objects
  - where requests are dynamically added or cancelled



GOLOG

- **Mathematical** programming language to implement **high-level** control algorithms for agents/robots
- Program execution requires agents/robots to **reason** about their actions

- Standard programming constructs but
  - primitive statements are actions

    Example. **proc** A

    Go(Up); Pick(P_1,B_1); Go(Down); Drop(B_1)

    endProc;

  - primitive tests refer to the environment of the agent/robot

    Example. **proc** B

    if *At(6)* **then** *Go(Down)* **endIf**

    endProc;

Reasoning About Actions

- Executing a GOLOG program requires reasoning about actions
  - to verify that an action is executable in the current situation
  - to evaluate a conditional statement in the current situation

Example. $Go(Up);$

$Pick(P_1,B_1);$

$\textbf{if}\ \neg At(6)\ \textbf{then}\ Go(Down)\ \textbf{endIf};$

- $Go(Up)$ possible in S_0?
- $Pick(P_1,B_1)$ possible in $Do(Go(Up,S_0))$?
- $\neg At(6)$ true in $Do(Pick(P_1,B_1),Do(Go(Up,S_0)))$?

The Other GOLOG Programming Constructs

- Primitive **test**

  $\neg At(6)\,?$

- **Nondeterministic choice** of subprogram

  $Go(Up) \mid Go(Down)$

- **Nondeterministic choice** of argument

  $\pi p.\ Pick(p, B_1)$

- **Nondeterministic iteration**

  $Go(Up)^*$

- **Loop**

  while $\neg At(6)$ ***do*** $Go(Up)$ ***endWhile***

  note: this is equivalent to $(\neg At(6)?\ ;\ Go(Up))^*\ ;\ At(6)?$

GOLOG Control Structures (Summary)

nil	empty program
a	primitive action
$\phi?$	test
$\delta_1 ; \delta_2$	sequential composition
$\delta_1 \mid \delta_2$	nondeterministic choice of sub-program
$\pi x.\delta(x)$	nondeterministic choice of argument
δ^*	nondeterministic iteration
$p(\vec{t})$	procedure call
if ϕ **then** δ_1 **else** δ_2 **endIf**	conditional
while ϕ **do** δ **endWhile**	loop

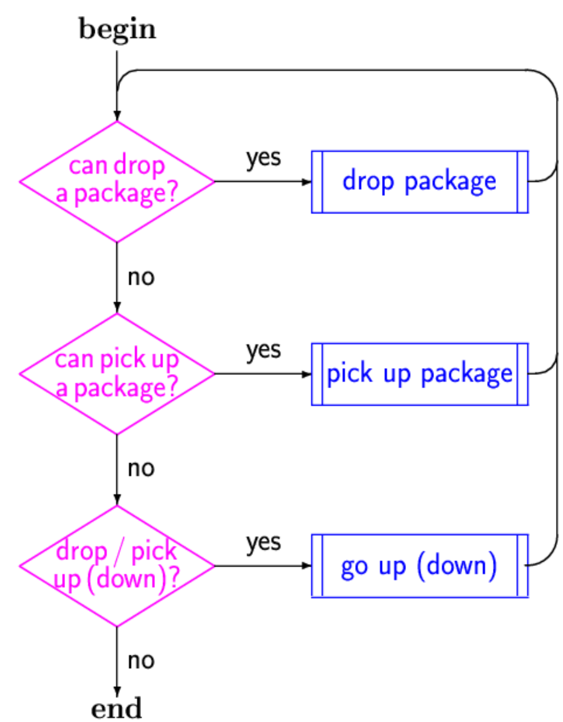
- General structure of a GOLOG program

proc $p_1(\vec{v}_1)$ δ_1 **endProc**;
...
proc $p_n(\vec{v}_n)$ δ_n **endProc**;
δ

GOLOG Program for the Delivery Robot (Part 1)

- Main loop

proc *Control*
  while $(\exists b, p, r)\ Carries(b, p, r) \lor (\exists p, r_1, r_2)\ Request\ (p, r_1, r_2)$ **do**
    if $(\exists b, p, r)\ (Carries(b, p, r) \land At(r))$ **then**
      Deliver
    else
      if $(\exists b, p, r_1, r_2)\ (Empty(b) \land Request\ (p, r_1, r_2) \land At(r_1))$ **then**
        Collect
      else
        Continue
      endIf
    endIf
  endWhile
endProc;



nondeterministic choice

- Auxiliary procedures

proc *Deliver* $\pi b.\ Drop(b)$ **endProc;**

proc *Collect* $\pi b.\ \pi p.\ Pick(p, b)$ **endProc;**

A GOLOG Program for the Mailbot (Part 2)

- Auxiliary procedure

go to where a parcel can be delivered

proc *Continue*

if $(\exists b, p, r)\ Carries(b, p, r)$ **then**

$\pi b.\ \pi p.\ \pi r.\ \pi r'.\ (At(r)\ ?\ ;\ Carries(b, p, r')\ ?\ ;$

if $r < r'$ **then** $Go(Up)$ **else** $Go(Down)$ **endIf**)

else

$\pi p. \pi r.\ \pi r_1.\ \pi r_2.\ (At(r)\ ?\ ;\ Request(p, r_1, r_2)\ ?\ ;$

if $r < r_1$ **then** $Go(Up)$ **else** $Go(Down)$ **endIf**)

endIf

endProc;

go to where a parcel can be picked up

- Main program body

Control

GOLOG in Prolog

GOLOG in Prolog

`www.cse.unsw.edu.au/~cs4418/2015/code/gologinterpreter.swi`

- **Simple** Prolog program to simulate execution of GOLOG programs
- Uses specific syntax for GOLOG programming constructs
  - Sequence

    a ; b ; c → `a : b : c`
  - Test

    p? → `?(p)`
  - Nondeterministic choice 1

    a | b | c → `a # b # c`
  - Nondeterministic choice 2

    $\pi x.\, a$ → `pi(x,a)`
  - Nondeterministic iteration

    a* → `star(a)`

GOLOG in Prolog (Part 1)

- Implementation via predicate `do(δ,s,s')`, which means
  - δ can be executed in situation s
  - s' is the result of **a possible execution** of δ in s

user-defined

```
do(E,S,do(E,S))  :- primitive_action(E), poss(E,S).
do(E1:E2,S,S1)   :- do(E1,S,S2), do(E2,S2,S1).    % sequence
do(?(P),S,S)     :- holds(P,S).                    % test
do(E1#E2,S,S1)       :- ...                        % choice
do(pi(V,E),S,S1)     :- ...                        % choice
do(star(E),S,S1)     :- ...                        % iteration
do(if(P,E),S,S1)     :- ...
do(if(P,E1,E2),S,S1) :- ...
do(while(P,E),S,S1)  :- ...
do(E,S,S1)           :- proc(E,E1), do(E1,S,S1).
```

see next slide

user-defined procedures

GOLOG in Prolog (Part 2)

- Recursive evaluation of test conditions
- Uses special syntax for logical connectives

```
holds(P & Q,S) :- holds(P,S), holds(Q,S).    % and
holds(P v Q,S) :- ...                         % or
holds(P => Q,S) :- ...                        % implication
holds(P <=> Q,S) :- ...                       % iff
holds(all(V,P),S) :- ...                      % ∀x
holds(some(V,P),S) :- ...                     % ∃x
holds(-P,S) :- ...                            % not


holds(A,S) :- restoreSitArg(A,S,F), F.        % fluents
```

user-defined

Users Guide to the Prolog Implementation (1)

- Define all your actions thus:

```
primitive_action(bake(_)).
primitive_action(eat(_)).
```

- Define all your fluents thus:

```
restoreSitArg(cake(C),S,cake(C,S)).
restoreSitArg(have(C),S,have(C,S)).
restoreSitArg(eaten(C),S,eaten(C,S)).
```

Users Guide to the Prolog Implementation (2)

- Implement action preconditions:

```
poss(bake(C),S) :- cake(C,S), not have(C,S).
poss(eat(C),S)  :- cake(C,S), have(C,S).
```

- Implement successor state axioms:

```
cake(C,do(A,S)) :- cake(C,S).

have(C,do(A,S)) :- A = bake(C).
have(C,do(A,S)) :- have(C,S), not A = eat(C).

eaten(C,do(A,S)) :- A = eat(C).
eaten(C,do(A,S)) :- eaten(C,S).
```


Users Guide to the Prolog Implementation (3)

- Define initial state:

```
cake(cheesecake,s0).
cake(pavlova,s0).
cake(friand,s0).
have(cheesecake,s0).
```

- Optional: Define additional predicates, e.g. goal conditions

```
restoreSitArg(goal,S,goal(S)).
goal(S) :- eaten(pavlova,S), eaten(cheesecake,S).
```

- Implement GOLOG program thus:

```
proc(devour(C), if(-have(C),bake(C)) : eat(C)).

proc(main, while(-goal, pi(c, ?(cake(c) & -eaten(c))
                                    : devour(c))
                )
     ).
```

Encoding the Delivery Robot Program in Prolog

- The Prolog-GOLOG interpreter requires us to define the actions & fluents in our domain thus:

```
% 3 actions
primitive_action(go(_)).
primitive_action(pick(_,_)).
primitive_action(drop(_)).

% 4 fluents
restoreSitArg(at(R),S,at(R,S)).
restoreSitArg(empty(B),S,empty(B,S)).
restoreSitArg(carries(B,P,R),S,carries(B,P,R,S)).
restoreSitArg(request(P,R1,R2),S,request(P,R1,R2,S)).
```

GOLOG Program for the Delivery Robot in Prolog (1)

```
proc(deliver, pi(b,drop(b))).

proc(collect, pi(b,pi(p,pick(p,b)))).

proc(continue, if(some(b,some(p,some(r,carries(b,p,r)))),
      pi(b,pi(p,pi(r,pi(r1,?(at(r)) : ?(carries(b,p,r1)) :
                 if(r<r1,go(up),go(down)))))),
      pi(p,pi(r,pi(r1,pi(r2,?(at(r)) : ?(request(p,r1,r2)) :
                 if(r<r1,go(up),go(down))))))
                  )).
```

proc *Continue*
if $(\exists b, p, r)$ *Carries*(b, p, r) **then**
$\pi b.\ \pi p.\ \pi r.\ \pi r'.$ (*At*(r) ? ; *Carries* (b, p, r') ? ;
if $r < r'$ **then** *Go*(*Up*) **else** *Go*(*Down*) **endIf**)
else
$\pi p.\ \pi r.\ \pi r_1.\ \pi r_2.$ (*At*(r) ? ; *Request* (p, r_1, r_2) ? ;
if $r < r_1$ **then** *Go*(*Up*) **else** *Go*(*Down*) **endIf**)
endIf
endProc

GOLOG Program for the Delivery Robot in Prolog (2)

```
proc(control, while(some(b,some(p,some(r,carries(b,p,r))))
                     v
                 some(p,some(r1,some(r2,request(p,r1,r2)))),
   if(some(b,some(p,some(r,carries(b,p,r) & at(r)))),
      deliver,
      if(some(b,some(p,some(r1,some(r2,
                 empty(b) & request(p,r1,r2) & at(r1))))),
         collect,
         continue)))).
```

proc *Control* **while** $(\exists b, p, r)$ $Carries(b, p, r) \vee (\exists p, r_1, r_2)$ $Request\,(p, r_1, r_2)$ **do**
if $(\exists b, p, r)$ $(Carries(b, p, r) \wedge At(r))$ **then**
Deliver
else
if $(\exists b, p, r_1, r_2)$ $(Empty(b) \wedge Request\,(p, r_1, r_2) \wedge At(r_1))$ **then**
Collect
else
Continue
endIf endIf endWhile endProc;

Example Initial Situation and Queries

```
at(1,s0).
empty(b1,s0).
request(p1,1,5,s0).
request(p2,1,2,s0).
```

```
?- do(deliver,s0,S).
no

?- do(collect,s0,S).
S = do(pick(p1,b1),s0) More?
S = do(pick(p2,b1),s0) More?
no

?- do(control,s0,S).
S = do(drop(b1),do(go(up),do(pick(p2,b1),...
```

Users Guide to the Prolog Implementation (4)

Be careful

- "Variables" in
  - tests using quantifiers
  - π – statements

  need to be **lowercase**

```
proc(gorge, while(some(c,have(c)), pi(c,eat(c)))).

proc(deliver, pi(b,drop(b))).
```

- Other variables in upper case, as usual

```
proc(devour(C), if(-have(C),bake(C)) : eat(C)).
```

- No built-in loop check; hence, the following may loop indefinitely

```
proc(main, while(-goal, pi(c, ?(cake(c)) : devour(c)))).
```

Online- vs. Offline-Execution

Online-execution

- Actions are performed immediately as program is executed step by step

→ Agent commits to every nondeterministic choice (don't care-nondeterminism)

→ Robot delivery and cake eating programs are examples

→ Note, however, that in general a program may not be completed successfully even though a successful run may exist

Offline-execution

- Program is run in simulation first

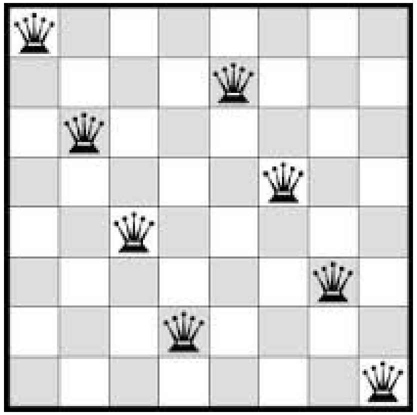
→ Allows to find a successful run if it exists; compare different runs to find the shortest; etc. (don't know-nondeterminism)

→ Can be used for problem solving (planning); see following exercise

→ But practically infeasible for large programs with many nondeterministic operations

Exercise

Exercise



- The 8-Queens problem is to
  - place 8 queens on a chess board
  - such that no two queens attack each other
- Fluents:
  - *Placed(i,s)* – i queens have been placed on the board
  - *Queen(row,col,s)* – queen placed at position (row,col) $\in$ {0..7} x {0..7}
- Action:
  - *Put(row,col)* – place a queen at (row,col) $\in$ {0..7} x {0..7} so that no existing queen attacks it
- Implement in Prolog the following GOLOG program for offline execution

while $\neg$*Placed(8)* **do**
  π *row.* π *col.* *Put(row,col)*
endWhile

What would happen if we executed the program **online**?

GOLOG: Extensions

- **ConGOLOG**: concurrency, interrupts, exogenous actions
  - allows for dynamic environments
- **IndiGOLOG**
  - interleaves on-line and off-line execution
  - agents/robots use **limited** lookahead when executing a program
  - sensing, exogenous events

Summary

- Situation calculus to represent actions and to reason about them
- GOLOG programs
- GOLOG in Prolog