Prolog

Prolog: ‘Programmation en Logique’

- First-order predicate logic for the specification of data and relationships
- Computation = logical deduction

1972 - A. Colmerauer, P. Roussel: first Prolog-like interpreter
Logic Programming (LP)

- R.A. Kowalski (Imperial College):
  
  Algorithm = Logic + Control

- Imperative languages (C++, Java):
  
  - data (what)
  - operations on data (how)
  - no separation between ‘what’ and ‘how’
LP: Logic + Control

what

Logical Formulas

∀x (P(x) → Q(x))

how

Logical Deduction

¬ψ, φ →ψ

¬φ
What: Problem Description

- Horn clause: $A \leftarrow B_1, B_2, \ldots, B_n$

- Equivalent to: $A \lor \neg B_1 \lor \neg B_2 \lor \ldots \lor \neg B_n$

- Meaning: $A$ is true if
  - $B_1$ is true, and
  - $B_2$ is true, ...., and
  - $B_n$ is true
What: Problem Description

\[ A \leftarrow B_1, \ldots, B_n \]

- specification of **facts** concerning **objects** and **relations** between objects
- specification of **rules** concerning **objects** and **relations** between objects
- specification of **queries** concerning **objects** and **relations**
Problem Description

- **Facts:** $A \leftarrow$

- **Rules:** $A \leftarrow B_1,\ldots,B_n$

- **Queries:** $\leftarrow B_1,\ldots,B_n$
Meet the Royal Family
Example: Family Relations

- **Facts:** mother(juliana, beatrix) ← constant

- **Rules:**
  
  parent(X, Y) ← mother(X, Y)

  parent(X, Y) ← father(X, Y)

- **Query:** ← parent(juliana, beatrix)
Logic Program

mother(juliana, beatrix) ←
mother(beatrix, alexander) ←
father(claus, alexander) ←

parent(X, Y) ← mother(X, Y)
parent(X, Y) ← father(X, Y)

Queries:

← parent(claus, alexander)
← parent(beatrix, juliana)
Prolog

- Prolog: practical realisation of LP
- Prolog clause:
  - Example:
    - mother(juliana, beatrix).
    - parent(X, Y) :- mother(X,Y).
    - :- parent(juliana, beatrix).
Why is Prolog so Handy?

Hotel suite design:

front door

Living Room

window

Bed Room

door

window
1. Living-room window opposite the front door
2. Bed-room door at right angle with front door
3. Bed-room window adjacent to wall with bed-room door
4. Bed-room window should face East
C-like

```c

// Define directions
typedef enum {north, south, east, west} dir;

// Function to check if a living room is adjacent to a bedroom
bool livrm(dir fd, dir lw, dir bd) {
    return opposite(fd, lw) && adjacent(fd, bd);
}

// Function to check if a bedroom is adjacent to another bedroom
bool bedrm(dir bd, dir bw) {
    return adjacent(bd, bw) && (bw == east);
}

// Function to check if a suite is adjacent to a living room and a bedroom
bool suite(dir fd, dir lw, dir bd, dir bw) {
    return livrm(fd, lw, bd) && bedrm(bd, bw);
}

```
for fd = north to west do
    for lw = north to west do
        for bd = north to west do
            for bw = north to west do
                if suite(fd, lw, bd, bw) then
                    print(fd, lw, bd, bw)
In Prolog

\begin{align*}
\text{livrm}(\text{Fd}, \text{Bd}, \text{Lw}) & : - \text{opposite}(\text{Fd}, \text{Lw}), \text{adjacent}(\text{Fd}, \text{Bd}). \\
\text{bedrm}(\text{Bd}, \text{Bw}) & : - \text{adjacent}(\text{Bd}, \text{Bw}), \text{Bw} = \text{east}. \\
\text{suite}(\text{Fd}, \text{Lw}, \text{Bd}, \text{Be}) & : - \text{livrm}(\text{Fd}, \text{Lw}, \text{Bd}), \text{bedrm}(\text{Bd}, \text{Bw}). \\
\text{:- suite}(\text{Fd}, \text{Lw}, \text{Bd}, \text{Bw}).
\end{align*}
Declarative Semantics

- Prolog clause: \( A :- B_1, B_2, \ldots, B_n \).

- **Meaning:** \( A \) is true if
  - \( B_1 \) is true, and
  - \( B_2 \) is true, \ldots, and
  - \( B_n \) is true
Procedural Semantics

- Prolog as a procedural language
- Prolog clause = procedure
  
  \[
  A : - B_1, B_2, \ldots, B_n. 
  \]

- Query = procedure call
  
  \[
  : - B_1, B_2, \ldots, B_n. 
  \]
More General Programs

- Use often lists:
  \[ [a, b, c, d] = [a | [b, c, d]] \]

- Element is first element (fact):
  \[ \text{member}(a, [a | [b, c, d]]) \].

- In general:
  \[ \text{member}(X, [X|\_]). \]
Set Membership

member(X, [X|_]).
member(X, [_|Y]) :- member(X, Y)

Queries:

:- member(a,[b,a,c])

:- member(d,[b,a,c])
Example 1

/*1*/ member(X, [X|_]).  procedure entry
/*2*/ member(X, [_|Y]) :- procedure entry
    member(X, Y).
/*3*/ :- member(a, [a, b, c]).  call

Step 1

    :- member(a, [a, b, c]).
/*1*/  member(X, [X|_]).
/*1*/  member(X, [X|_]).

Instantiation: X = a match with /*1*/
Example 2

/*1*/ member(X, [X|_]).  procedure entry
/*2*/ member(X, [_|Y]) :- procedure entry
member(X, Y).
/*3*/ :- member(a, [b, a, c]).  call

Step 1

:- member(a, [b, a, c]).
/*1*/ member(X, [X|_]).

Instantiation: X = a \textit{no match with} /*1*/
Example 2 (continued)

/*1*/ member(X, [X|_]).  procedure entry
/*2*/ member(X, [_|Y]) :- procedure entry
    member(X, Y).
/*3*/ :- member(a, [b,a,c]).  call

Step 2

    :- member(a, [b, a, c]).
/*2*/  member(X, [_|Y]) :- member(X, Y).

Match: X = a;  Y = [a, c]
/*1*/ member(X, [X|_]).  procedure entry
/*2*/ member(X, [_|Y]) :- procedure entry
    member(X, Y).
/*3*/ :- member(a, [b,a,c]).  call

Step 3

:- member(a, [a, c]).  subcall

/*1*/ member(X, [X|_]).

Match: X = a
Matching

- A call and procedure head match if:
  - predicate symbols are equal
  - arguments in corresponding positions are equal

- Example:

  ```prolog
  :- member(a, [a, c]).
  /*1*/ member(a, [a|_]).
  ```
Variables & Atoms

mother(juliana, beatrix).

Calls:
:- mother(X, Y).
    X = juliana
    Y = beatrix

:- mother(_, _). /* anonymous variable */
    yes

:- mother(juliana, juliana).
    no
Left-right Selection Rule

\[\text{Call:}
\]
\[\text{:- } p.\]
\[\text{:- } q, r, s.\]
\[\text{:- } r, s.\]
\[\text{:- } s.\]
Top-bottom Selection Rule

\[ p(a). \]
\[ p(b). \]
\[ p(c). \]
\[ p(X) : - q(X). \]
\[ q(d). \]
\[ q(e). \]

Call:
\[ : - p(Y). \]
\[ Y = a; \]
\[ Y = b; \]
\[ Y = c; \]
\[ Y = d; \]
\[ Y = e; \]
\[ no \]
Backtracking: systematic search for alternatives

Example: search for paths in tree $T$
branch(a, b).
branch(a, c).
branch(c, d).
branch(c, e).
path(X, X).
path(X, Y) :-
  branch(X, Z), path(Z, Y).
Backtracking

branch(a, b).
branch(a, c).
branch(c, d).
branch(c, e).
path(X, X).
path(X, Y) :-
  branch(X, Z), path(Z, Y).

:- path(a, d). /* query */
path(a, d) :- branch(a, Z), path(Z, d).

branch(a, Z) => 1
Z = b
branch(a, b).

branch(a, b)
Backtracking

branch(a, b).
branch(a, c).
branch(c, d).
branch(c, e).
path(X, X).
path(X, Y) :-
    branch(X, Z), path(Z, Y).

:- path(a, d). /* query */
path(a, d) :- branch(a, Z), path(Z, d).

X = b, Y = d

path(b, d) :- branch(b, Z'), path(Z', d).
Backtracking

branch(a, b).
branch(a, c).
branch(c, d).
branch(c, e).
path(X, X).
path(X, Y) :-
  branch(X, Z), path(Z, Y).

path(b, d) :- branch(b, Z'), path(Z', d).

3

branch(b, Z')

backtrack

3
branch(a, b).
branch(a, c).
branch(c, d).
branch(c, e).
path(X, X).
path(X, Y) :-
    branch(X, Z), path(Z, Y).

:- path(a, d). /* query */

path(a, d) :- branch(a, Z), path(Z, d).

branch(a, c).
branch(a, Z) 1'
Z = c

1'
Backtracking

```
branch(a, b).
branch(a, c).
branch(c, d).
branch(c, e).
path(X, X).
path(X, Y) :-
    branch(X, Z), path(Z, Y).
```

```
:- path(a, d). /* query */
path(a, d) :- branch(a, Z), path(Z, d).
```

```
X = c, Y = d
path(c, d) :- branch(c, Z'), path(Z', d).
```

```
path(c, d) :- branch(c, Z'), path(Z', d).
```

```
Z = c
path(c, d)
```

```
X = c, Y = d
```

```
2' 2'
```
Backtracking

\[
\begin{align*}
\text{branch}(a, b). \\
\text{branch}(a, c). \\
\text{branch}(c, d). \\
\text{branch}(c, e). \\
\text{path}(X, X). \\
\text{path}(X, Y) & : - \\
& \quad \text{branch}(X, Z), \text{path}(Z, Y). \\
\end{align*}
\]

\[
\begin{align*}
\text{path}(c, d) & : - \text{branch}(c, Z'), \text{path}(Z', d). \\
\end{align*}
\]
Backtracking

branch(a, b).
branch(a, c).
branch(c, d).
branch(c, e).
path(X, X).
path(X, Y) :-
    branch(X, Z), path(Z, Y).

path(c, d) :- branch(c, Z'), path(Z', d).

4
branch(c, Z'), path(Z', d).
Z' = d

4
path(d, d)
X = d
path(d, d)
Terminology

- From programming languages (Prolog as procedural language):

  \[ \text{nat}(0). \]
  \[ \text{nat}(\text{s}(X)) :\text{-} \text{nat}(X). \]

  - term: \text{nat}(0), \text{nat}(\text{s}(X)), \text{nat}(X),
  \text{:-}(\text{nat}(\text{s}(X)), \text{nat}(X)), \text{s}(X), 0, X

  - functor: \text{s}, \text{nat}, :\text{-}

  - principal functor: \text{nat} in \text{nat}(\text{s}(X)), :\text{-} in
  \text{:-}(\text{nat}(\text{s}(X)), \text{nat}(X)), \text{s} in \text{s}(X)

  - number: 0

  - variable: \text{X}
Inversion of Computation (1)

- **Example:** concatenation of lists
  \[ U = V ° W \]
  with \( U, V, W \) lists and \( ° \) concatenation operator

- **Usage:**
  - \([a, b] = [a] ° W \Rightarrow W = [b]\)
  - \([a, b] = V ° [b] \Rightarrow V = [a]\)
  - \(U = [a] ° [b] \Rightarrow U = [a, b]\)
  - \([a, b] = V ° W?\)
Inversion of Computation (2)

- Prolog concatenation of lists:
  \[
  \text{concat}([], U, U).
  \]
  \[
  \text{concat}([X|U], V, [X|W]) :- \text{concat}(U, V, W).
  \]

- \text{concat} as \text{constructor}:
  \[
  ?- \text{concat}([a, b], [c, d], X).
  \]
  \[
  X = [a, b, c, d]
  \]

- \text{concat} used for \text{decomposition}:
  \[
  ?- \text{concat}(X, Y, [a, b, c, d]).
  \]
  \[
  X = []
  \]
  \[
  Y = [a, b, c, d]
  \]
concat used for decomposition:
?- concat(X, Y, [a, b, c, d]).

X = []
Y = [a, b, c, d];
X = [a]
Y = [b, c, d];
X = [a, b]
Y = [c, d];

...

Inversion of Computation (3)
Order of Clauses (1)

- **LP**: order is irrelevant
- **Prolog**: order may be relevant
- **Example:**

  ```prolog
  member(X,[_|Y]) :-
      member(X,Y).
  member(X,[X|_]).
  :- member(a,[b,a,c]).
  ```
/*1*/  member(X, [___|Y]) :-
    member(X, Y).
/*2*/  member(X, [X|___]).

?- member(a, [a,b]).
  X = a, Y = [b]              match with 1
  ?- member(a, [b]).          next call
    X' = a, Y' = []           match with 1
    ?- member(a, []).          fail 1 and 2
  fail 1 and 2
  fail 1, backtracking to 2
  X = a                      match 2
  yes! (but not efficient)
Order of Clauses (3)

/*1*/ member(X, [_|Y]) :-
    member(X, Y).
/*2*/ member(X, [X|_]).

?- member(X, [a, b]).
   X' = X, Y = [b]  match with 1
   ?- member(X', [b]).
      X'' = X', Y' = []  match with 1
      ?- member(X'', []).  fail 1 and 2
   X' = b                 fail 1, match 2
   X = b; backtracking
   X = a
   yes! (but not efficient)

?- member(X, [a, b]).
   match with 1
   ?- member(X', [b]).
      next call
      X'' = X', Y' = []  match with 1
      ?- member(X'', []).  fail 1 and 2
   X' = b                 fail 1, match 2
   X = b; backtracking
   X = a
   yes! (but not efficient)
Order of Clauses (4)

/*1*/ member(X, [Y|_]) :-
    member(X, Y).

/*2*/ member(X, [X|_]).

?- member(a, Z).
   X = a, Z = [Y|_]    match 1
?- member(a, Y).
   X’ = a, Y = [Y’|_]    match 1
?- member(a, Y’).

... Stack overflow 📣
Conclusions Order of Clauses

- **LP:** order clauses is irrelevant

- **Prolog:**
  - Order has effect on *efficiency* of program
  - Order may affect termination: terminating program + order change ≠ terminating program
Order of Conditions (1)

- Length of list with successor function
  \( s : \mathbb{N} \rightarrow \mathbb{N}, \text{ with } s(x) = x + 1 \)

- Program:

  /*1*/ length([], 0).
  /*2*/ length([_|X], N) :-
      length(X, M),
      N = s(M).

- Use:

  ?- length([a, b], N).
  N = s(s(0))
Order of Conditions (2)

- Program:

  /*1*/ length([], 0).
  /*2*/ length([__|X], N) :-
      length(X, M),
      N = s(M).

- Use:

  ?- length(L, s(0)).
  L = [_A];

  Stack overflow 🚨
Order of Conditions (3)

/*1*/ length([], 0).
/*2*/ length([_|X], N) :-
    length(X, M),
    N = s(M).

Trace:
?- length(L, s(0)).
L = [_A|X], N = s(0)  match 2
?- length(X, M), s(0) = s(M).
X = [], M = 0   subcall
match 1
?- s(0) = s(0).
match
L = [_A];
backtracks
... (1 fails)
Order of Conditions (4)

/*1*/ length([], 0).
/*2*/ length([_|X], N) :-
    length(X, M),
    N = s(M).

- Trace:
?- length(L, s(0)).
  L = [_A|X], N = s(0)  match 2
  ?- length(X, M), s(0) = s(M).  subcall
    X = [_B|X'], N = M  match 2
    ?- length(X', M'), M = s(M'), s(0) = s(M).  subcall
      ...

Order of Conditions (5)

- **Program:**
  
  ```
  /*1*/ length([], 0).
  /*2*/ length([_|X], N) :-
     N = s(M),
     length(X, M).
  ```

- **Use:**
  
  ```
  ?- length(L, s(0)).
  L = [_A];
  ```

  **no */
Declarative vs Procedural

- Order of clauses and conditions in Prolog programs may be changed, but

- This may be at the expense of:
  - loss of termination
  - compromised efficiency

- Schema for procedural programming:
  - special case first (top, left)
  - general case (e.g. including a recursive call) last (bottom, right)
Fail & Cut

- Notation: fail and !
- Control predicates: affect backtracking
- Used for:
  - efficiency reasons
  - implementing tricks
Enforcing Backtracking: fail

?- fail.
  no  (no match)

Program:
  p(a).
  p(b).

Query:
  ?- p(X).  (match)
  X = a
  yes
Fail - no Recursion

- Program:
  p(a).
p(b).
p(X) :- q(X).
q(c).

- Query:
  ?- p(X), write(X), nl, fail.
  a
  b
  c
  no
  backtracking
Fail - with Recursion

Program:

/*1*/  member(X, [X|_]).
/*2*/  member(X, [_|Y]) :-
    member(X,Y).

Query/call:
?- member(Z,[a,b]), write(Z), nl, fail.
   Z = X, X = a            match 1
?- write(a), nl, fail.
   a backtracking
?- member(Z,[a,b]), write(Z), nl, fail.
   Z = X, Y = [b]          match 2
Controlling Backtracking: !

- Procedural meaning of the cut !:

\[
A :\neg B_1, B_2, !, B_3, B_4.
\]

- Search for alternatives
- Stop searching
Program:
\[ p(a). \]
\[ p(b). \]
\[ q(X) :- p(X), r(X). \]
\[ r(Y) :- !, t(Y). \]
\[ r(a). \]
\[ t(c). \]

Execution:
\[- q(Z). \]
\[ Z = X \]
\[- p(X), r(X). \]
\[ X = a \]
\[- r(a). \]
\[ Y = a \]
\[- t(a). \]
fail, no backtracking to r(a).
Try \( X = b \)
State Space and !

a :- b, c.
a :- f, g.
...

b :- d, !, e.
b :- ....
...

d.

?- a.
?- !, e, c.
fail

?- b, c.
?- d, !, e, c.
fail

?- f, g.

fail
Various Applications of !

- Cut as commitment operator:
  
  ```
  if X < 3 then Y = 0
  if X \geq 3 \text{ and } X < 6 \text{ then } Y = 2
  if X \geq 6 \text{ then } Y = 4
  ```

- Prolog:
  
  ```prolog
  t(X, 0) :- X < 3.
  t(X, 2) :- X \geq 3, X < 6.
  t(X, 4) :- X \geq 6.
  ```
Commitment Operator

- Cut as commitment operator:
  
  /*1*/ \( t(X, 0) : - X < 3. \)
  /*2*/ \( t(X, 2) : - X \geq 3, X < 6. \)
  /*3*/ \( t(X, 4) : - X \geq 6. \)

- Execution trace:

  ?- \( t(1, Y) \), \( Y > 2. \) \hspace{1cm} \textit{match 1}
  ?- \( 1 < 3, 0 > 2. \) \hspace{1cm} \textit{fail 1}
  ?- \( 0 > 2. \) \hspace{1cm} \textit{fail 2}
  ?- \( 1 \geq 3, 1 < 6, 1 > 2. \) \hspace{1cm} \textit{match 2}
  ?- \( \ldots \) \hspace{1cm} \textit{fail 2}
  ?- \( 1 \geq 6, 4 > 2. \) \hspace{1cm} \textit{match 3, fail 3}
Commitment Operator

- Cut as commitment operator:

  /*1*/  \texttt{t(X, 0)} :: \texttt{X < 3}, !.
  /*2*/  \texttt{t(X, 2)} :: \texttt{X >= 3}, \texttt{X < 6}, !.
  /*3*/  \texttt{t(X, 4)} :: \texttt{X >= 6}.

- Execution trace:

  ?- \texttt{t(1, Y)}, Y > 2. \hspace{1cm} \text{match 1}
  ?- 1 < 3, !, 0 > 2. \hspace{1cm} \text{fail 1}
  ?- !, 0 > 2. \hspace{1cm} \text{no}
Various Applications of !

- Cut used for removal of conditions:

  \[
  \text{min}(X, Y) \text{ is } X \text{ if } X \leq Y \\
  \text{min}(X, Y) \text{ is } Y \text{ if } X > Y
  \]

- Prolog:

  \[
  \text{min}(X, Y, X) :- X \leq Y. \\
  \text{min}(X, Y, Y) :- X > Y.
  \]

- Execution:

  \[
  ?- \text{min}(3, 5, Z). \\
  ?- 3 \leq 5. \quad \text{match 1}
  \]

  \[
  Z = 3 \quad \text{yes}
  \]
Removal of Conditions

- Cut used for removal of conditions:

  \[
  \text{min}(X, Y, Z) : \begin{align*}
  & X =< Y, !, \\
  & Z = X. \\
  \text{min}(X, Y, Y).
  \end{align*}
  \]

- Execution:

  ?- \text{min}(3, 5, W).
  \text{match 1}
  \text{W = 3 yes}
Removal of Conditions

- **Cut used for removal of conditions:**
  \[
  \text{min}(X, Y, Z \Rightarrow X) :\ \\
  X =\leq Y, \ \\
  Z = X.
  \]

- **Execution:**
  \[
  \text{?- min}(3, 5, 5).
  \]
  
  
  fail 1, match 2

  yes

  *why included?*
Change in Meaning?

- Cut used for removal of conditions:
  
  \[
  \text{min}(X, Y, Z) :-
  \]
  
  \[
  X \leq Y, \quad (! \text{ omitted})
  \]
  
  \[
  Z = X.
  \]
  
  \[
  \text{min}(X, Y, Y).
  \]

- Execution:
  
  \[
  ?- \text{min}(3, 5, W), W = 5.
  \]
  
  \[
  ?- 3 \leq 5, 5 = 3, W = 5. \quad \text{match 1}
  \]
  
  \[
  ?- 5 = 3, W = 5. \quad \text{fail}
  \]
  
  \[
  ?- W = 5. \quad \text{match 2 (with Y = 5 = W)}
  \]
  
  \[
  W = 5
  \]
  
  yes
Negation by Failure

- Simulation of negation: \( \text{not}(p) \) is true if \( p \) is false (fails):

  \[
  \text{not}(X) :- \text{call}(X), !, \text{fail}.
  \]

- Example:

  \[
  p(a).
  q(X) :- p(X), \text{not}(r(X)).
  r(c).
  ?- q(Y).
  \]

  yes
Circumvention of double search:

/*1*/  member(X, [X|_]) :- !.
/*2*/  member(X,[_|Y]) :-
        member(X,Y).

Example:

?- member(a, [a, b, a]).
yes

?- member(X, [a, b]).
X = a;
no
Green and Red Cuts

**Green cut:**
- when omitted, does not change declarative (logical) meaning of program
- used to increase efficiency

**Red cut:**
- when omitted, declarative meaning of program is changed
- used for efficiency
- used to enforce termination
Green and Red Cuts

- **Green cut:**
  - commitment operator

- **Red cut:**
  - removal of conditions
  - cut-fail combination (see notes)
  - single solution
The working environment of Prolog, containing all loaded Prolog programs is called: the ‘database’

The database can be manipulated by the programs themselves

Compare: Pascal program that modifies itself during execution
Prolog ‘Database’

add new clauses

remove clauses

parent(jim, bob).
pred(X,Y) :- parent(X,Y). 
pred(X,Y) :- parent(X,Z), pred(Z,Y).
assertz: add to the end of a definition

assertz(parent(bob, ann)).
### Asserting Clauses

Database

<table>
<thead>
<tr>
<th>collect_data(stop).</th>
</tr>
</thead>
<tbody>
<tr>
<td>collect_data(_) :-</td>
</tr>
<tr>
<td>write('Next item: '),</td>
</tr>
<tr>
<td>read(X),</td>
</tr>
<tr>
<td>assertz(X),</td>
</tr>
<tr>
<td>collect_data(X).</td>
</tr>
</tbody>
</table>

| input_data :- |
| collect_data(start). |

```prolog
?- input_data.
Next item: name(peter).
Next item: age(35).
Next item: stop.
```

- `name(peter)`.  
- `age(35)`.  
- `stop`.  

---
Database Manipulation

- **Asserting (new) clauses:**
  - `assert(C)`: position C unspecified
  - `asserta(C)`: at the beginning of the definition of the predicate
  - `assertz(C)`: at the end of the definition of the predicate

- **Deleting clauses:**
  - `retract(C)`: remove clause matching with C (top to bottom order)
Retracting Clauses

retract: remove from the beginning of the definition

?- retract(parent(X,Y)).
  X = jim
  Y = bob
  yes

Prolog Database

?- dynamic parent/2.
  parent(jim, bob).
  parent(bob, ann).
  parent(john, pete).
  parent(pete, linda).
Retracting Clauses

?- dynamic parent/2.

parent(jim, bob).
parent(bob, ann).
parent(john, pete).
parent(pete, linda).

?- retract_all_facts(parent(X,Y)).
yes