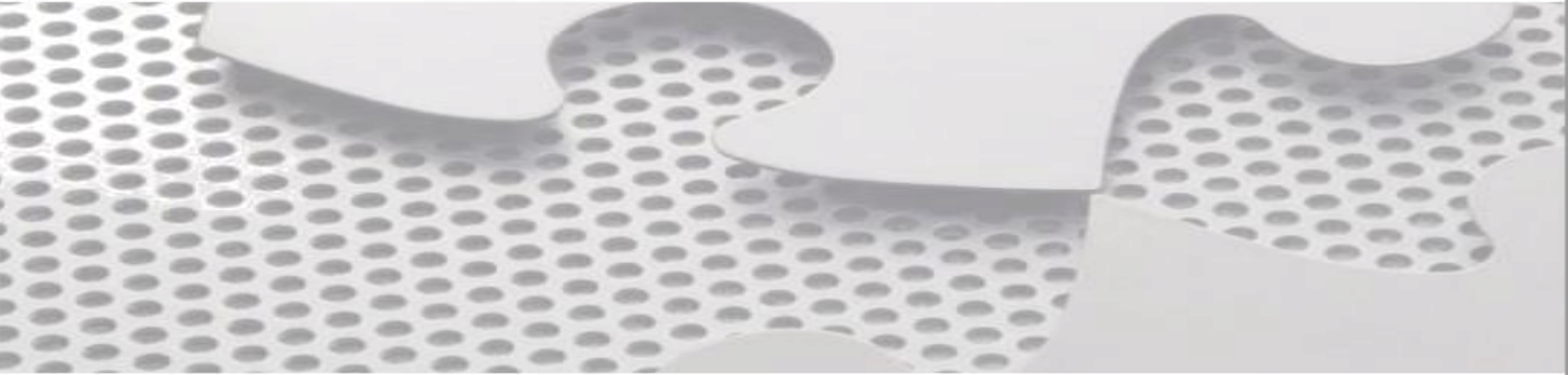
Logic Programming



Chapter 4

Companion slides from the book

Loops and Control Structures

- Can use the backtracking of Prolog to perform loops and repetitive searches
  - Must force backtracking even when a solution is found by using the built-in predicate *fail*
- Example:

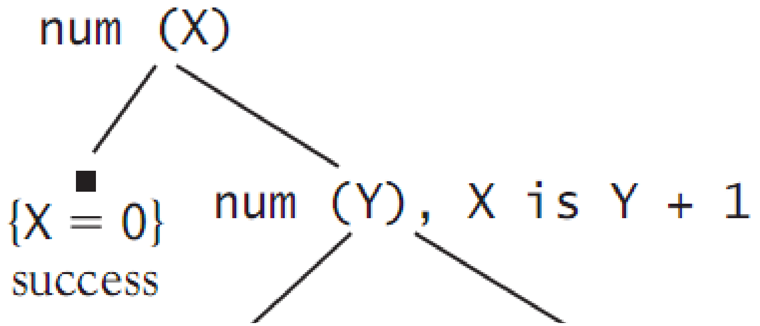
```
printpieces(L) :-append(X, Y, L),
                 write(X),
                 write(Y),
                 nl,
                 fail.
```

Loops and Control Structures (cont'd.)

- Use this technique also to get repetitive computations
- Example: these clauses generate all integers greater than or equal to 0 as solutions to the goal *num(X)*

```
(1) num(0).
(2) num(X) :- num(Y), X is Y + 1.
```

Loops and Control Structures




```
(1) num(0).
(2) num(X) :- num(Y), X is Y + 1.
```

The search tree has an infinite branch to the right

Figure 4.3 An infinite Prolog search tree showing repetitive computations

- Example: trying to generate integers from 1 to 10

```
(1) num(0).
(2) num(X) :- num(Y), X is Y + 1.
```

```
writenum(I, J) :- num(X),
                  I =< X,
                  X =< J,
                  write(X),
                  nl,
                  fail.
```

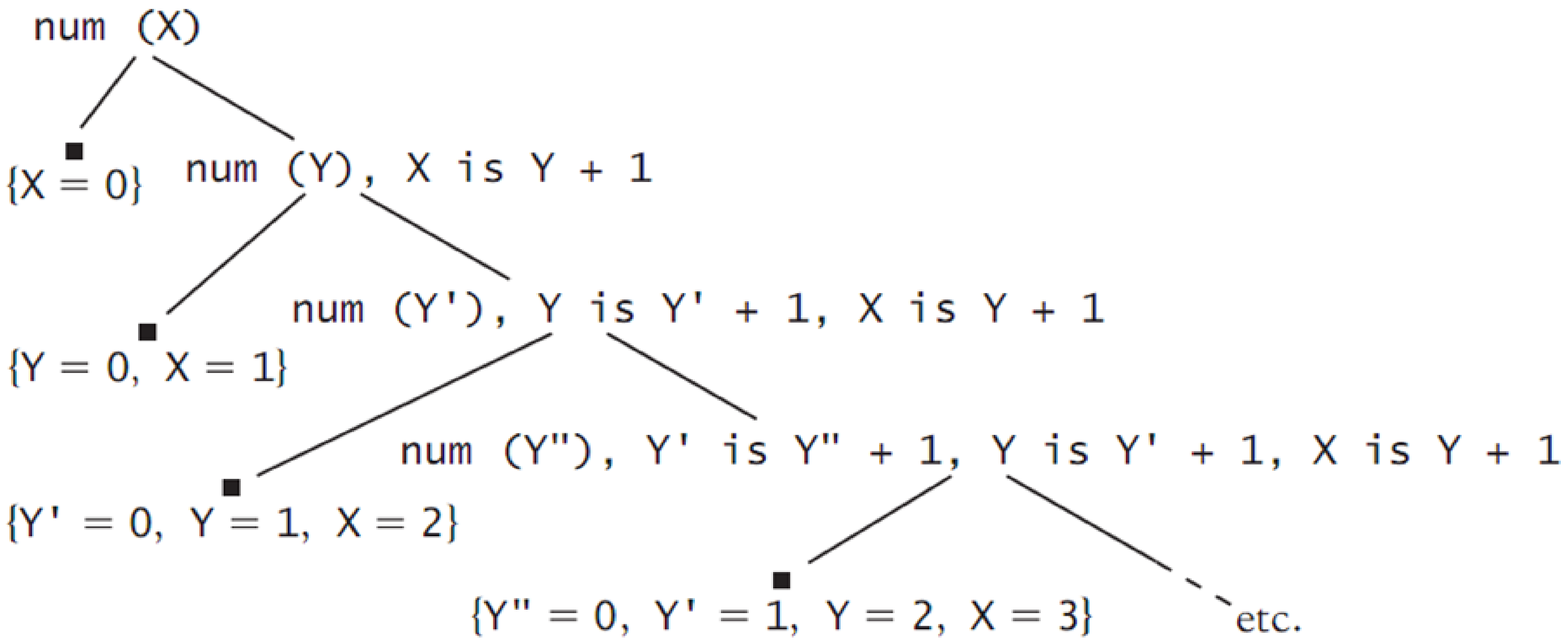
```
writeList(I,J):-num(X),
                I=<X, X=<J,
                write(X),
                nl,
                X=J.
```

- Causes an infinite loop after `X = 10`, even though `X =< 10` will never succeed

List of Numbers

```
num(0).
num(X) :- num(Y),  (X is Y + 1 ).

writeListwithCut(I,J):-num(X), I=<X, X=<J, write(X), nl, X=J, !.
```



Max counting

- Predicate max/3 which takes integers as arguments and succeeds if the third argument is the maximum of the first two.
- Input output
- ?- max(2,3,3)
- ?- max(3,2,3)
- ?- max(3,3,3)
- ?- max(2,3,5)
- ?- max(2,3,X)

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

- There can never be any second solution. So, it should not backtrack.
- The two clauses are mutually exclusive!

```
1.  max(X,Y,Y):-  X  =<  Y,!.
2. max(X,Y,X):-  X>Y.
```

Second clause will be evaluated only if first one does not satisfy. Once got passed the cut, control cannot backtrack!

cut operator (written as `!`) freezes a choice when it is encountered

Green Cuts:- Cuts like this, which doesn't change the meaning of a program

```
1.  max(X,Y,Y) :- X =< Y,!.
2.  max(X,Y,X).
```

```
?- max(100,101,X).        X=101, yes
?- max(3,2,X).            X=3, yes
?- max(2,3,2).
```

```
1.  max(X,Y,Z) :- X =< Y,!, Y = Z.
2.  max(X,Y,X).
```

If–thenelse

- Can also use cut to imitate *if–else* constructs in imperative and functional languages, such as:

  D = if A then B else C

- Prolog code:

```
D :- A, !, B.
D :- C.
```

- Could achieve almost same result without the cut, but *A* would be executed twice

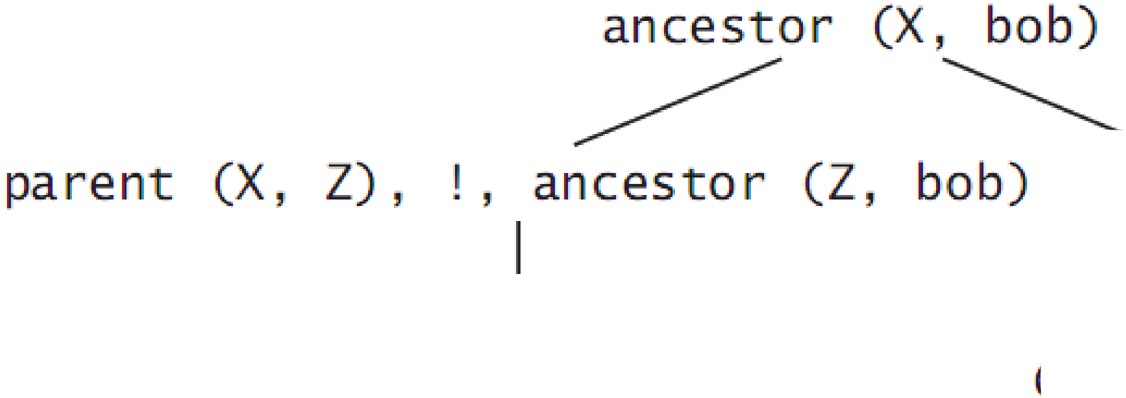
```
D :- A, B.
D :- not(A), C.
```

Loops and Control Structures

- If a cut is reached on backtracking, search of the subtrees of the parent node stops, and the search continues with the grandparent node
  - Cut prunes the search tree of all other siblings to the right of the node containing the cut
- Example:

```
(1) ancestor(X, Y):- parent(X, Z), !, ancestor(Z, Y).
(2) ancestor(X, X).
(3) parent(amy, bob).
```

```
(1) ancestor(X, Y):- parent(X, Z), !, ancestor(Z, Y).
(2) ancestor(X, X).
(3) parent(amy, bob).
```



Only `X = amy` will be found since the branch containing `X = bob` will be pruned

Figure 4.4 Consequences of the cut for the search tree of Figure 4.2

Loops and Control Structures (cont'd.)

- Rewriting this example:

```
(1) ancestor(X, Y) :- !, parent(X, Z), ancestor(Z, Y).
(2) ancestor(X, X).
(3) parent(amy, bob).
```

Loops and Control Structures (cont'd.)

- Rewriting again:

```
(1) ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
(2) ancestor(X, X) :- !.
(3) parent(amy, bob).
```

- Cut can be used to reduce the number of branches in the subtree that need to be followed
- Also solves the problem of the infinite loop in the program to print numbers between `I` and `J` shown earlier

Summation of a list

- Sum(1,1):-!.
- Sum(N,R):-

```
N1 is N-1, Sum(N1,R1),
Res is R1+N.
```

```
1. insert(X,[],[X]).

2. insert(X,[H|Tail],[X,H|Tail]):- X =< H.

3. insert(X,[H|Tail],[H|NewTail]):- X > H,
                                      insert(X,Tail,NewTail).
```

```
1. isort([],[]).

2. isort([X|Tail],SList):- isort(Tail,STail), insert(X,STail,SList).
```

http://kti.ms.mff.cuni.cz/~bartak/prolog/learning.html

Problems with Logic Programming

- Original goal of logic programming was to make programming a specification activity
  - Allow the programmer to specify only the properties of a solution and let the language implementation provide the actual method for computing the solution
- **Declarative programming**: program describes *what* a solution to a given problem is, not *how* the problem is solved
- Logic programming languages, especially Prolog, have only partially met this goal

Problems with Logic Programming (cont'd.)

- The programmer must be aware of the pitfalls in the nature of the algorithms used by logic programming systems
- The programmer must sometimes take an even lower-level perspective of a program, such as exploiting the underlying backtrack mechanism to implement a cut/fail loop

The Occur-Check Problem in Unification

- **Occur-check problem**: when unifying a variable with a term, Prolog does not check whether the variable itself occurs in the term it is being instantiated to
- Example: `is_own_successor :- X = successor(X).`

- This will be true if there exists an `X` for which `X` is its own successor
- But even in the absence of any other clauses for successor, Prolog answers yes

The Occur-Check Problem in Unification (cont'd.)

- This becomes apparent if we make Prolog try to print such an X:

```
is_own_successor(X) :- X = successor(X).
```

  - Prolog responds with an infinite loop because unification has constructed *X* as a circular structure
  - What should be logically false now becomes a programming error

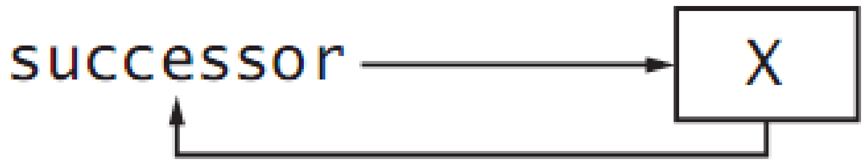


Figure 4-7: Circular structure created by unification

Negation as Failure

- **Closed-world assumption**: something that cannot be proved to be true is assumed to be false
  - Is a basic property of all logic programming systems
- **Negation as failure**: the goal `not(X)` succeeds whenever the goal `X` fails
- Example: program with one clause: `parent(amy,bob).`
- If we ask: `?- not(mother(amy, bob)).`
  - The answer is `yes` since the system has no knowledge of `mother`
  - If we add facts about `mother`, this would no longer be true

Negation as Failure (cont'd.)

- **Nonmonotonic reasoning**: the property that adding information to a system can reduce the number of things that can be proved
  - This is a consequence of the closed-world assumption
- A related problem is that failure causes instantiation of variables to be released by backtracking
  - A variable may no longer have an appropriate value after failure

Negation as Failure (cont'd.)

- Example: assumes the fact *human(bob)*

```
?- human(X).
X = bob

?- not(not(human(X))).
X = _23
```

- The goal `not(not(human(X)))` succeeds because `not(human(X))` fails, but the instantiation of `X` to `bob` is released

Negation as Failure (cont'd.)

- Example:

```
?- X = 0, not(X = 1).
X = 0

?- not (X = 1), X = 0.
no
```

  - The second pair of goals fails because *X* is instantiated to `1` to make `X = 1` succeed, and then `not(X=1)` fails
  - The goal `X = 0` is never reached

Horn Clauses Do Not Express All of Logic

- Not every logical statement can be turned into Horn clauses
  - Statements with quantifiers may be problematic
- Example:

$$p(a) \text{ and } (\text{there exists } x, \text{ not}(p(x))).$$

- Attempting to use Prolog, we might write:

```
p(a).
not(p(b)).
```

  - Causes an error: trying to redefine the *not* operator

Horn Clauses Do Not Express All of Logic (cont'd.)

- A better approximation would be simply *p(a)*
  - Closed-world assumption will force *not(p(X))* to be true for all *X* not equal to *a*
  - But this is really the logical equivalent of:

$$p(a) \text{ and } (\text{for all } x, \text{not}(x = a) \rightarrow \text{not}(p(a))).$$

  - This is not the same as the original statement

Control Information in Logic Programming

- Because of its depth-first search strategy and linear processing of goals and statements, Prolog programs also contain implicit information on control that can cause programs to fail
  - Changing the order of the right-hand side of a clause may cause an infinite loop
  - Changing the order of clauses may find all solutions but still go into an infinite loop searching for further (nonexistent) solutions

Control Information in Logic Programming

- One would want a logic programming system to accept a mathematical definition and find an efficient algorithm to compute it
- Instead, we must specify actual steps in the algorithm to get a reasonable efficient sort
- In logic programming system, we not only provide specifications in our programs, but we must also provide algorithmic control information