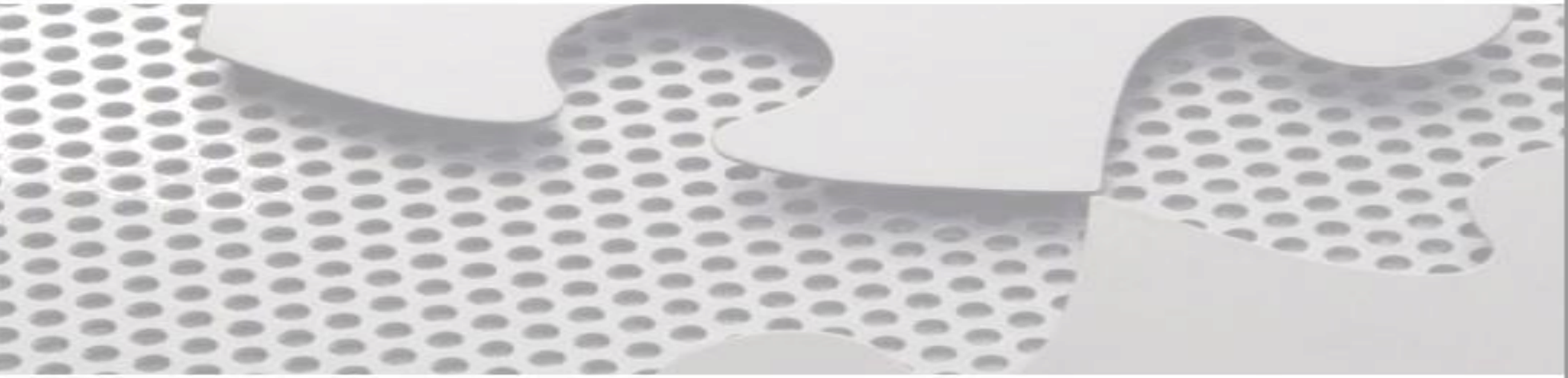
Logic Programming



Chapter 4

Companion slides from the book

Introduction

- **Logic**: the science of reasoning and proof
  - Existed since the time of ancient Greek philosophers
- Logic is closely associated with computers and programming languages
  - Circuits are designed using Boolean algebra
  - Logical statements are used to describe **axiomatic semantics**, the semantics of programming languages
- Logic programming typically restricts itself to well-behaved fragments of logic

https://groups.seas.harvard.edu/courses/cs152/2015sp/lectures/lec23-prolog.pdf

Logic Programming

- We can think of logic programs as having two interpretations.
- In the declarative interpretation, a logic program declares what is being computed.
- In the procedural interpretation, a logic program describes how a computation takes place.
- In the declarative interpretation, one can reason about the correctness of a program without needing to think about underlying computation mechanisms
- This makes declarative programs easier to understand, and to develop.
- A lot of the time, once we have developed a declarative program using a logic programming language, we also have an executable specification
  - A procedural interpretation that tells us how to compute what we described.
  - This is one of the appealing features of logic programming.
  - An understanding of the underlying computational mechanism is needed to make the execution of the declarative program efficient

Logic and Logic Programs

- **First-order predicate calculus**: a way of formally expressing logical statements
- **Logical statements**: statements that are either true or false
- **Axioms**: logical statements that are assumed to be true and from which other true statements can be **proved**

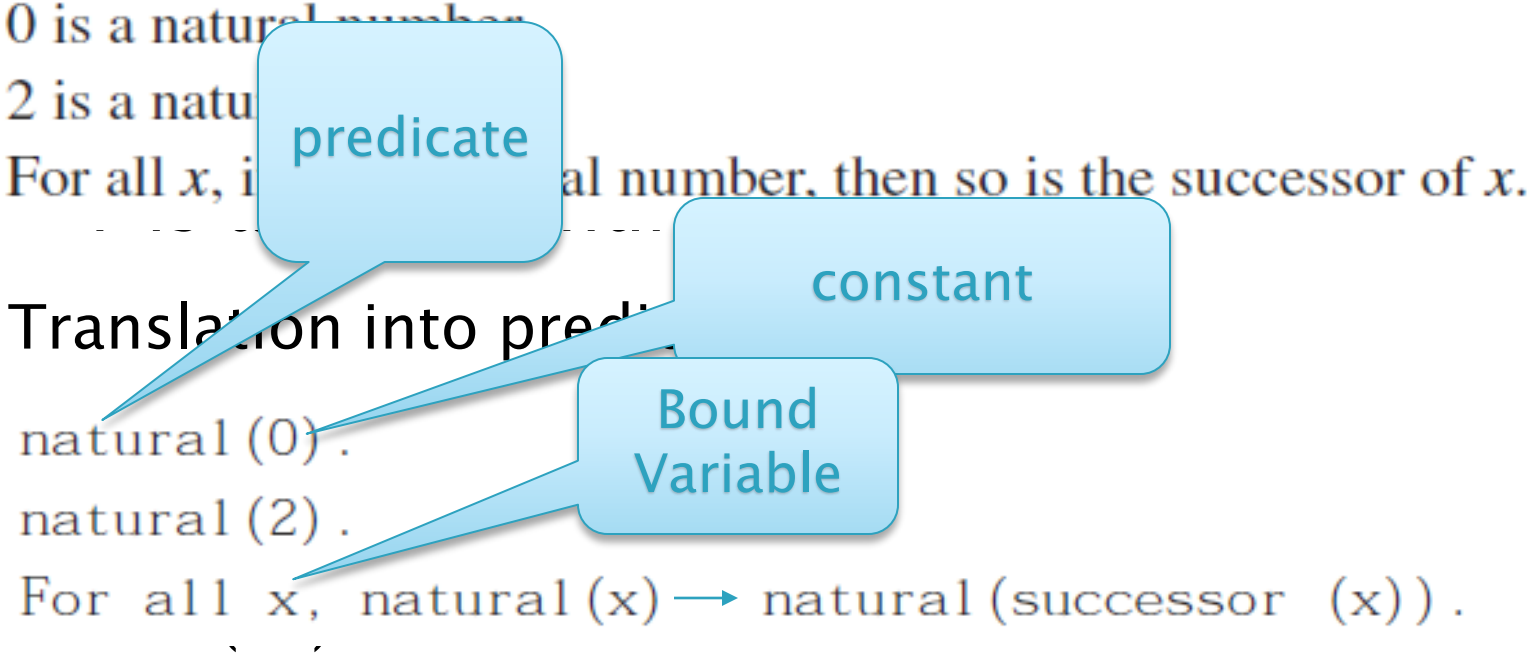
Logic and Logic Programs (cont'd.)

Symbols in statements:

- *Constants (a.k.a. atoms)*
  numbers (e.g., `0`) or names (e.g., `bill`).
- *Predicates*
  Boolean functions (true/false) . Can have arguments. (e.g. `parent(X,Y)`).
- *Functions*
  non-Boolean functions (`successor(X)`).
- *Variables*
  Stands for yet unspecified quantities
  e.g., `X`.
- *Connectives (operations)*
  `and, or, not`
  `implication (→):a→b (b or not a)`
  `equivalence (↔): a↔b (a→b and b→a)`

Logic and Logic Programs (cont'd.)

- Example 1:
  - These English statements are logical statements



  - *x* in the third statement is a variable that stands for an as yet unspecified quantity

Logic and Logic Programs (cont'd.)

- Example 1 (cont'd.)
  - First and third statements are axioms
  - Second statement can be proved since

```
2 = successor(successor(0))
        and natural(0) → natural (successor(0))

      → natural(successor(successor (0)))
```

Logic and Logic Programs (cont'd.)

- **Universal quantifier**: a relationship among predicates is true for all things in the universe named by the variable
  - Ex: *for all x, natural(x) natural(successor(x))*

- **Existential quantifier**: a predicate is true for at least one thing in the universe indicated by the variable
  - Ex: *there exists x, natural(x)*

- A variable introduced by a quantifier is said to be **bound** by the quantifier

Logic and Logic Programs (cont'd.)

- A variable not bound by a quantifier is said to be **free**
- Arguments to predicates and functions can only be **terms**: combinations of variables, constants, and functions
  - Terms cannot contain predicates, quantifiers, or connectives

Quantifier examples

- Examples:
  - ∀x(speaks(x,Russian))
  - ∃x(speaks(x,Russian))
  - ∀x ∃y(speaks(x,y))
  - ∃ x ∀ y(speaks(x,y))

Example 2

- *"Every man is mortal.*
- *John is a man. Therefore, John is mortal"*

Every man is mortal : $(\forall x)\ (MAN(x) \rightarrow MORTAL(x))$
John is a man : $MAN(john)$
John is mortal : $MORTAL(john)$

Logic and Logic Programs (cont'd.)

- First-order predicate calculus also has inference rules
- **Inference rules**: ways of deriving or proving new statements from a given set of statements
- Example: from the statements $a \rightarrow b$ and $b \rightarrow c$, one can derive the statement $a \rightarrow c$, written formally as:

$$\frac{(a \rightarrow b) \text{ and } (b \rightarrow c)}{a \rightarrow c}$$

Logic and Logic Programs (cont'd.)

- **Logic programming language**: a notational system for writing logical statements together with specified algorithms for implementing inference rules
- **Logic program**: the set of logical statements that are taken to be axioms
- The statement(s) to be derived can be viewed as the input that initiates the computation
  - Also called **queries** or **goals**

Logic and Logic Programs (cont'd.)

- Logic programming systems are sometimes referred to as **deductive databases**
  - Consist of a set of statements and a deduction system that can respond to queries
  - System can answer queries about facts and queries involving implications
- **Control problem**: specific path or sequence of steps used to derive a statement

Logic and Logic Programs (cont'd.)

- The statements represent the logic of the computation while the deductive systems provide the control by which a new statement is derived
- Logical programming paradigm as a pseudo equation (Kowalski):

  algorithm = logic + control

- Compare this with imperative programming (Wirth):

  algorithms = data structures + programs

- Since logic programs do not express the control, in theory, operations can be carried out in any order or simultaneously
  - Logic programming languages are natural candidates for parallelism

Logic and Logic Programs (cont'd.)

- Automated deduction systems have difficulty handling all of first-order predicate calculus
  - Too many ways of expressing the same statements
  - Too many inference rules
- Most logic programming systems restrict themselves to a particular subset of predicate calculus called Horn clauses

Horn Clauses

- **Horn clause**: a statement of the form

$$a_1 \text{ and } a_2 \text{ and } a_3 \ldots \text{and } a_n \rightarrow b$$

- The a_i are only allowed to be simple statements
  - No *or* connectives and no quantifiers allowed
- This statement says that a_1 through a_n imply b, or that b is true if all the a_i are true
  - b is the **head** of the clause
  - The $a_1,\ldots,a_n$ is the **body** of the clause
- If there are no a_i*'s,* the clause becomes → b
  - b is always true and is called a **fact**

Horn Clauses (cont'd.)

- Example 4: first-order predicate calculus:

```
natural(0).
for all x, natural(x) → natural (successor (x)).
```

- Translate these into Horn clauses by dropping the quantifier:

```
natural(0).
natural(x) → natural (successor(x)).
```

1. Or connectives should be broken down to multiple clauses
2. Variables of the universal quantifiers will appear at the head of the clause
3. Variables with the existential quantifier will appear at the body of the clause

Horn Clauses (cont'd.)

- Example 5: logical description for the Euclidian algorithm for greatest common divisor of two positive integers *u* and *v*:

The gcd of *u* and 0 is *u*.
The gcd of *u* and *v*, if *v* is not 0,

is the same as the gcd of *v* and the remainder of dividing *v* into *u*.

- First-order predicate calculus:

```
for all u, gcd(u, 0, u).
for all u, for all v, for all w,
not zero(v) and gcd(v, u mod v, w) → gcd(u, v, w).
```

Horn Clauses (cont'd.)

- Note that `gcd(u,v,w)` is a predicate expressing that `w` is the `gcd` of `u` and `v`
- Translate into Horn clauses by dropping the quantifiers:

```
gcd(u, 0, u).
not zero(v) and gcd(v, u mod v, w) → gcd(u, v, w).
```

Horn Clauses (cont'd.)

x is a grandparent of *y* if *x* is the parent of someone who is the parent of *y*.

Horn Clauses (cont'd.)

- **Procedural interpretation**: Horn clauses can be reversed to view them as a procedure

$$b \leftarrow a_1 \text{ and } a_2 \text{ and } a_3 \ldots \text{and } a_n$$

- This becomes procedure *b*, wherein the body is the operations indicated by the a_i's

- Most logic programming systems write Horn clauses backward and drop the *and* connectives:

```
gcd(u, 0, u).
gcd(u, v, w) ← not zero(v), gcd(v, u mod v, w).
```

- Similar to standard programming language expression for the gcd:

```
gcd(u, v) = if v = 0 then u else gcd(v, u mod v).
```

Horn Clauses (cont'd.)

- Horn clauses do not supply the algorithms, only the properties that the result must have
- Horn clauses can also be viewed as **specifications** of procedures rather than strictly as implementations

- Example: specification of a sort procedure:

```
sort(x, y) ← permutation(x, y) and sorted(y).
```

Horn Clauses (cont'd.)

- Parsing of natural language was a motivation for the original development of Prolog
- **Definite clause grammars**: the particular kind of grammar rules used in Prolog programs
- Logic programs can be used to directly construct parsers

Definite Clause Grammars

- A grammar is a precise definition of which sequences of words or symbols belong to some language
- Prolog provides a notational extension called DCG (Definite Clause Grammar) that allows the direct implementation of formal grammars

sentence → noun_phrase, verb_phrase

e.g “ The man ran.”

noun_phrase → noun

noun_phrase → determiner, noun

verb_phrase → intransitive_verb

verb_phrase → transitive_verb, noun_phrase

https://info-ruc.github.io/nlp20/dcg.pdf

Definite Clause Grammars

- We can add our own arguments to the non-terminals in DCG rules

```
sentence --> noun(Num), verb_phrase(Num).
verb_phrase(Num) --> verb(Num), noun(_).
noun(singular) --> [bob].
noun(plural) --> [students].
verb(singular) --> [likes].
verb(plural) --> [like].


| ?- sentence([bob, likes, students], []).

| ?- sentence([students, likes, bob], []).
```

Horn Clauses (cont'd.)

- Variable scope
  - Variables used in the head can be viewed as parameters
  - Variables used only in the body can be viewed as local, temporary variables
- Queries or goal statements: the exact opposite of a fact
  - Are Horn clauses with no head

```
natural(0)← .
natural(2).
```

Resolution and Unification

- **Resolution**: an inference rule for Horn clauses
  - If the head of the first Horn clause matches with one of the statements in the body of the second Horn clause, can replace the head with the body of the first in the body of the second
- Example: given two Horn clauses

$$a \leftarrow a_1, \ldots, a_n.$$
$$b \leftarrow b_1, \ldots, b_m.$$

  - If b_i matches a, then we can infer this clause:

$$b \leftarrow b_1, \ldots, b_i\text{-}1, a_1, \ldots, a_n, b_{i+1}, \ldots, b_m.$$

Resolution and Unification (cont'd.)

- Example: given *b*←*a* and *c*←*b*
  - Resolution says *c*←*a*

```
natural(0)← .
natural(0).
```

- Example: given *b*←*a* and *c*←*b*
  - Combine: *b,c*←*a,b*
  - Cancel the *b* on both sides: *c*←*a*

```
natural(0)← .
natural(0)← natural(0)
```

If the system eventually succeeds in eliminating all goals– thus deriving the empty Horn clause– then the original statement has been proved.

Resolution and Unification (cont'd.)

- A logic processing system uses this process to match a goal and replace it with the body, creating a new list of goals, called **subgoals**
- If all goals are eventually eliminated, deriving the empty Horn clause, then the original statement has been proved
- To match statements with variables, set the variables equal to terms to make the statements identical and then cancel from both sides
  - This process is called **unification**
  - Variables used this way are said to be **instantiated**

Unification

- Unification: process by which variables are instantiated to match during resolution
  - Basic expression whose semantics is determined by unification is equality
- Prolog's unification algorithm:
  - Constant unifies only with itself
  - Uninstantiated variable unifies with anything and becomes instantiated to that thing
  - Structured term (function applied to arguments) unifies with another term only if the same function name and same number of arguments

Unification (cont'd.)

- Examples:

```
?- me = me.

?- me = you.

?- me = X.

?- f(a, X) = f(Y, b).
```

```
?- f(X) = g(X).

?- f(X) = f(a, b).

?- f(a, g(X)) = f(Y, b).

?- f(a, g(X)) = f(Y, g(b)).
```

Resolution and Unification (cont'd.)

Given $b \leftarrow a$ and $c \leftarrow b$
Combine: $b,c \leftarrow a,b$

- Example 10: gcd with resolution and unification
- Goal:

```
gcd(u, 0, u).
gcd(u, v, w) ← not zero(v), gcd(v, u mod v, w).

← gcd(15, 10, x).
```

- Resolution fails with first clause (10 does not match 0), so use the second clause and unify

```
gcd(15, 10, x) ← not zero(10), gcd(10, 15 mod 10, x),
                 gcd(15, 10, x).
```

Resolution and Unification (cont'd.)

- Example 10 (cont'd.):
  - If *zero(10)* is false, then *not zero(10)* is true
  - Simplify 15 mod 10 to 5, and cancel *gcd (15, 10, x)* from both sides, giving

```
not zero(10), gcd(10, 15 mod 10, x)
```

```
← gcd(10, 5, x).
```

  - Use unification as before

```
gcd(10, 5, x) ← not zero(5), gcd(5, 10 mod 5, x),
                                gcd(10, 5, x).
```

  - To get this subgoal

```
← gcd(5, 0, x).
```

  - This now matches the first rule, so setting x to 5 gives the empty statement

Resolution and Unification (cont'd.)

- A logic programming system must have a fixed algorithm that specifies:
  - Order in which to attempt to resolve a list of goals
  - Order in which clauses are used to resolve goals
- In some cases, order can have a significant effect on the answers found
- Logic programming systems using Horn clauses and resolution with prespecified orders require that the programmer is aware of the way the system produces answers

The Language Prolog

- **Prolog**: the most widely used logic programming language
  - Uses Horn clauses
  - Implements resolution via a strictly depth-first strategy
- There is now an ISO standard for Prolog
  - Based on the Edinburgh Prolog version developed in the late 1970s and early 1980s

- We can of course **implement** theorem provers in Prolog!
- This is because Prolog is a Turing complete **programming language**, and every theorem prover that can be implemented on a computer can *also* be implemented in Prolog.

Notation and Data Structures

- Prolog notation is almost identical to Horn clauses
  - Implication arrow ← becomes :-
  - Variables are uppercase, while constants and names are lowercase
  - In most implementations, can also denote a variable with a leading underscore
  - Use comma for *and*, semicolon for *or*
  - List is written with square brackets, with items separated by commas
  - Lists may contain terms or variables

Execution in Prolog

- Most Prolog systems are interpreters
- Prolog program consists of:
  - Set of Horn clauses in Prolog syntax, usually entered from a file and stored in a dynamically maintained database of clauses
  - Set of goals, entered from a file or keyboard
- At runtime, the Prolog system will prompt for a query
- Unification, recursion and search are the three basic mechanisms of Prolog

Arithmetic

- Prolog has built-in arithmetic operations
  - Terms can be written in infix or prefix notation
- Prolog cannot tell when a term is arithmetic or strictly data
  - Must use built-in predicate *is* to force evaluation

```
?- write(3 + 5).
3 + 5

?- X is 3 + 5, write(X).
X = 8
```

Notation and Data Structures (cont'd.)

- Can specify head and tail of list using a vertical bar
- Example: `[H|T] = [1, 2, 3]` means
  - `H = 1, T = [2, 3]`
- Example: `[X, Y|Z] = [1, 2, 3]` means
  - `X=1, Y=2, and Z=[3]`
- Built-in predicates include `not`, `=`, and I/O operations `read`, `write`, and `nl` (newline)
- Less than or equal is usually written `=<` to avoid confusion with implication

Unification

- Unification: process by which variables are instantiated to match during resolution
  - Basic expression whose semantics is determined by unification is equality
- Prolog's unification algorithm:
  - Constant unifies only with itself
  - Uninstantiated variable unifies with anything and becomes instantiated to that thing
  - Structured term (function applied to arguments) unifies with another term only if the same function name and same number of arguments

Unification (cont'd.)

- Examples:

```
?- me = me.
```

```
?- me = you.
```

```
?- me = X.
```

```
?- f(a, X) = f(Y, b).
```

```
?- f(X) = g(X).
```

```
?- f(X) = f(a, b).
```

```
?- f(a, g(X)) = f(Y, b).
```

```
?- f(a, g(X)) = f(Y, g(b)).
```

Unification (cont'd.)

- Unification causes uninstantiated variables to share memory (to become aliases of each other)
  - Example: two uninstantiated variables are unified

```
?- X = Y.
X = _23
Y = _23
```

Unification for List

- Can specify head and tail of list using a vertical bar
- Example: `[H|T] = [1, 2, 3]` means
  - `H = 1, T = [2, 3]`
- Example: `[X, Y|Z] = [1, 2, 3]` means
  - `X=1, Y=2, and Z=[3]`
- Built-in predicates include `not`, `=`, and I/O operations `read`, `write`, and `nl` (newline)
- Less than or equal is usually written `=<` to avoid confusion with implication

Unifying with a list

- **Pattern-directed invocation**: using a pattern in place of a variable unifies it with a variable used in that place in a goal

- `cons(X,Y,L):- L=[X|Y].`

```
cons(X, Y, [X|Y]).
```

```
?- cons(0, [1, 2, 3], A).
A = [0, 1, 2,3]
```

- There is no need for a prepend function
- `X = [1|[2,3,4,5,8]].`
- No need for a function to extract the head/tail of a list
- `[X|Y] = [1,2,3,4,5].`
- `X = 1,`
- `Y = [2, 3, 4, 5].`

Unification (cont'd.)

- **Append** procedure:

```
append(X, Y, Z) :- X = [], Y = Z.
append(X,Y,Z)   :-
```

- First clause: appending a list to the empty list gives just that list
- Second clause: appending a list whose head is *A* and tail is *B* to a list *Y* gives a list whose head is also *A* and whose tail is *B* with *Y* appended

Unification with pattern

- Append procedure rewritten more concisely:

```
append([], Y, Y).
```

```
append(X, Y, Z) :- X = [], Y = Z.
append(X,Y,Z)   :- X = [A|B], Z = [A|W], append(B, Y, W).
```

Unification with pattern

- Append can also be run backward and find all the ways to append two lists to get a specified list:

```
?- append(X, Y, [1, 2]).
X = []
Y = [1, 2] ->;
```

```
X = [1]
Y = [2] ->;
```

```
X = [1,2].
Y = []
```

Unification (cont'd.)

- **Reverse** procedure:

```
reverse([],[]).
```

- Reverse of an empty list is empty
- Continues to search for solutions setting the subgoals
  - Reversing the non-empty tail part of the list
  - Then inserting the head of the list as the last element of the reversed list

Recursion

- `factorial(0,1).`
  `∀ N and ∀ M, factorial(N-1,M) → factorial(N,N*M).`

- factorial(0,1).
- factorial(N,F) :- N>0, N1 is N-1, factorial(N1,F1), F is N * F1.

Tail Recursion

- If the continuation is empty and there are no backtrack points, nothing need be placed on the stack; execution can simply jump to the called procedure, without storing any record of how to come back. This is called LAST-CALL OPTIMIZATION

- A procedure that calls itself with an empty continuation and no backtrack points is described as TAIL RECURSIVE, and last-call optimization is sometimes called TAIL-RECURSION OPTIMIZATION

- Fact(acc,n) {return n==1?acc:Fact(acc*n,n-1)}

- factorial(0,F,F).
- factorial(N,A,F) :-
- N > 0,
- A1 is N*A,
- N1 is N -1,
- factorial(N1,A1,F).
- ?-factorial(5,1,F).

Unification (cont'd.)

```
reverse([],[]).
reverse([H|T], L)  :- reverse(T, L1),
                      append(L1, [H], L).
```

- List: [1,2,3,4] Accumulator: []
- List: [2,3,4] Accumulator: [1]
- List: [3,4] Accumulator: [2,1]
- List: [4] Accumulator: [3,2,1]
- List: [] Accumulator: [4,3,2,1]

A predicate which carries out the required initialization of the accumulator :

```
reverse([H|T],L) :-  accReverse([H|T],[],L).

accReverse([H|T],A,L):-  accReverse(T,[H|A],L).
accReverse([],A,A).
```

Unification (cont'd.)

```
gcd(U, 0, U).
gcd(U, V, W) :- not(V = 0) , R is U mod V, gcd(V, R, W).

append([], Y, Y).
append([A|B], Y, [A|W]) :- append(B, Y, W).

reverse([], []).
reverse([H|T], L) :- reverse(T, L1),
                     append(L1, [H], L).
```

Figure 4.1 Prolog clauses for `gcd`, `append`, and `reverse`

Form a list by repeating a number say H, N times.

Execution in Prolog (cont'd.)

- Example 11: clauses entered into database

```
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
ancestor(X, X).
parent(amy, bob).
```

- Queries:

```
?- ancestor(amy,bob).
yes.

?- ancestor(bob,amy).
no.

?- ancestor(X,bob).
X = amy ->_
```

Execution in Prolog (cont'd.)

- Example 11 (cont'd.): use semicolon at prompt (meaning *or*)

```
?- ancestor(X,bob).
X = amy ->;
X = bob

?-_
```

- Use carriage return to cancel the continued search

Prolog's Search Strategy

- Prolog applies resolution in a strictly linear fashion
  - Replaces goals from left to right
  - Considers clauses in the database from top down
  - Subgoals are considered immediately
  - This search strategy results in a depth-first search on a tree of possible choices
- 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).
```

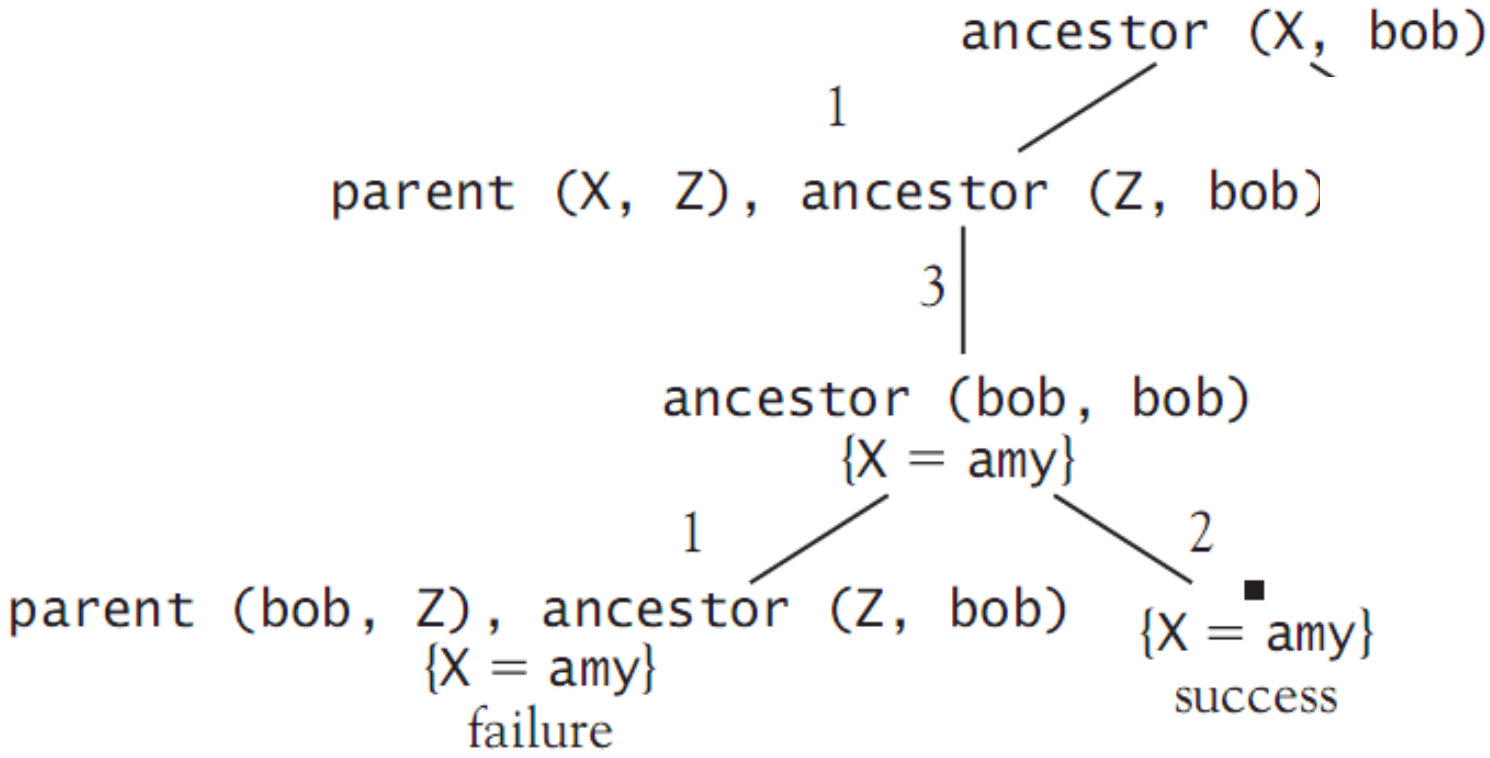


Figure 4.2 A Prolog search tree showing subgoals, clauses used for resolution, and variable instantiations

Prolog's Search Strategy (cont'd.)

- Leaf nodes in the tree occur either when no match is found for the leftmost clause or when all clauses have been eliminated (success)
- If failure, or the user indicates a continued search with a semicolon, Prolog **backtracks** up the tree to find further paths
- Releases instantiation of variables as it does so
- Depth-first strategy is efficient: can be implemented in a stack-based or recursive fashion

Infinite Loop

- Causes Prolog to go into an infinite loop attempting to satisfy *ancestor (Z, Y)*, continually reusing the first clause
- This happens because the first clause are written in a left recursive way that no other causes precede
- Prolog evaluates the clauses in left recursive manner
- Breadth-first search would always find solutions if they exist
  - Far more expensive than depth-first, so not used

Loops and Control Structures

- Can use DFS strategy with 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.
```

- Example: trying to generate integers from 1 to 10

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

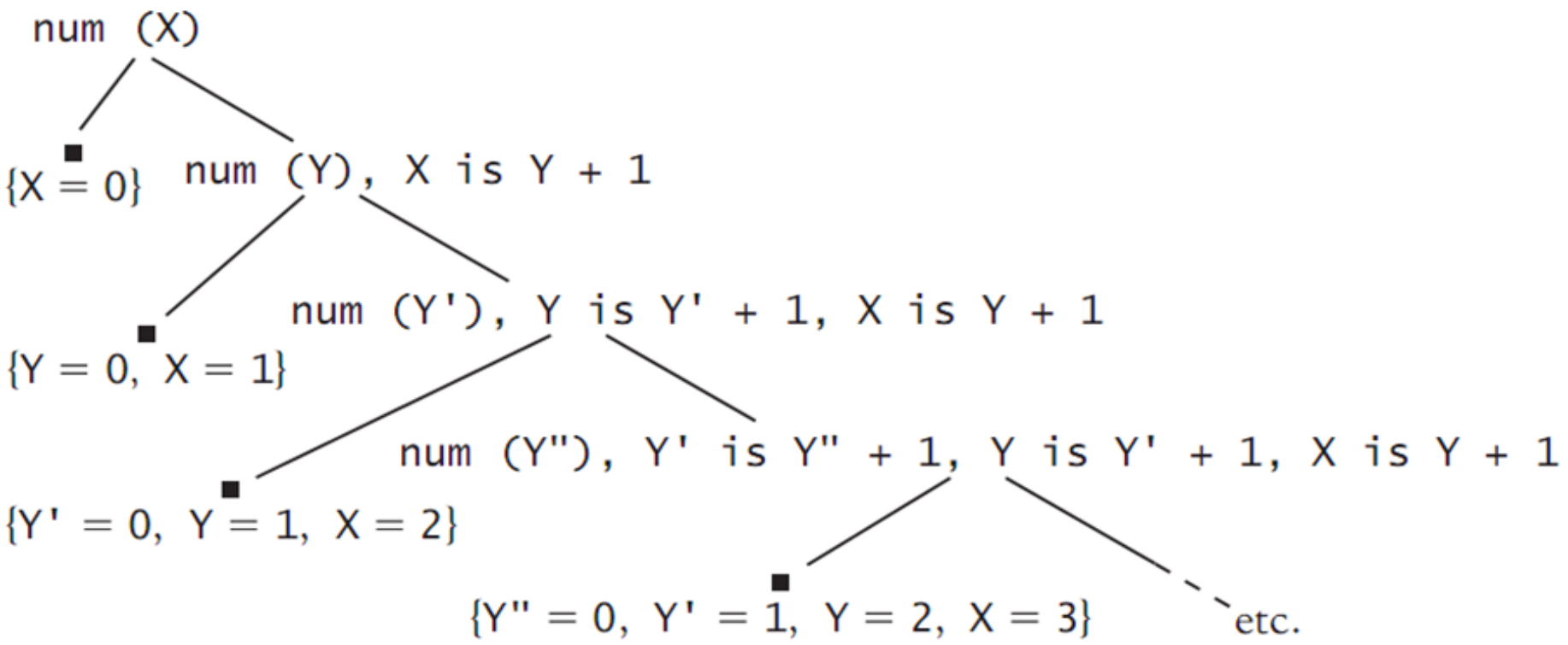
```
writeList(I,J):-num(X),
                I=<X, X=<J,
                write(X),
                nl,
                X=J.
```

- 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
- ?- max(2,3,3)
- ?- max(3,2,3)
- ?- max(3,3,3)
- ?- max(2,3,5)
- ?- max(2,3,X)

output

```
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

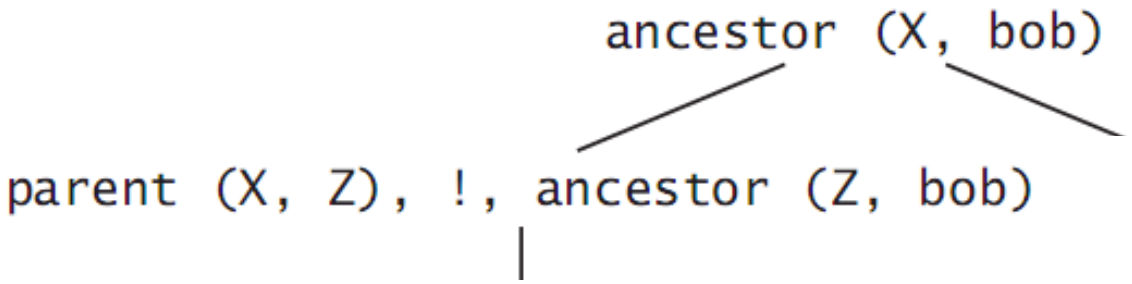
```
D :- A, B.
D :- not(A), C.
```

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.)

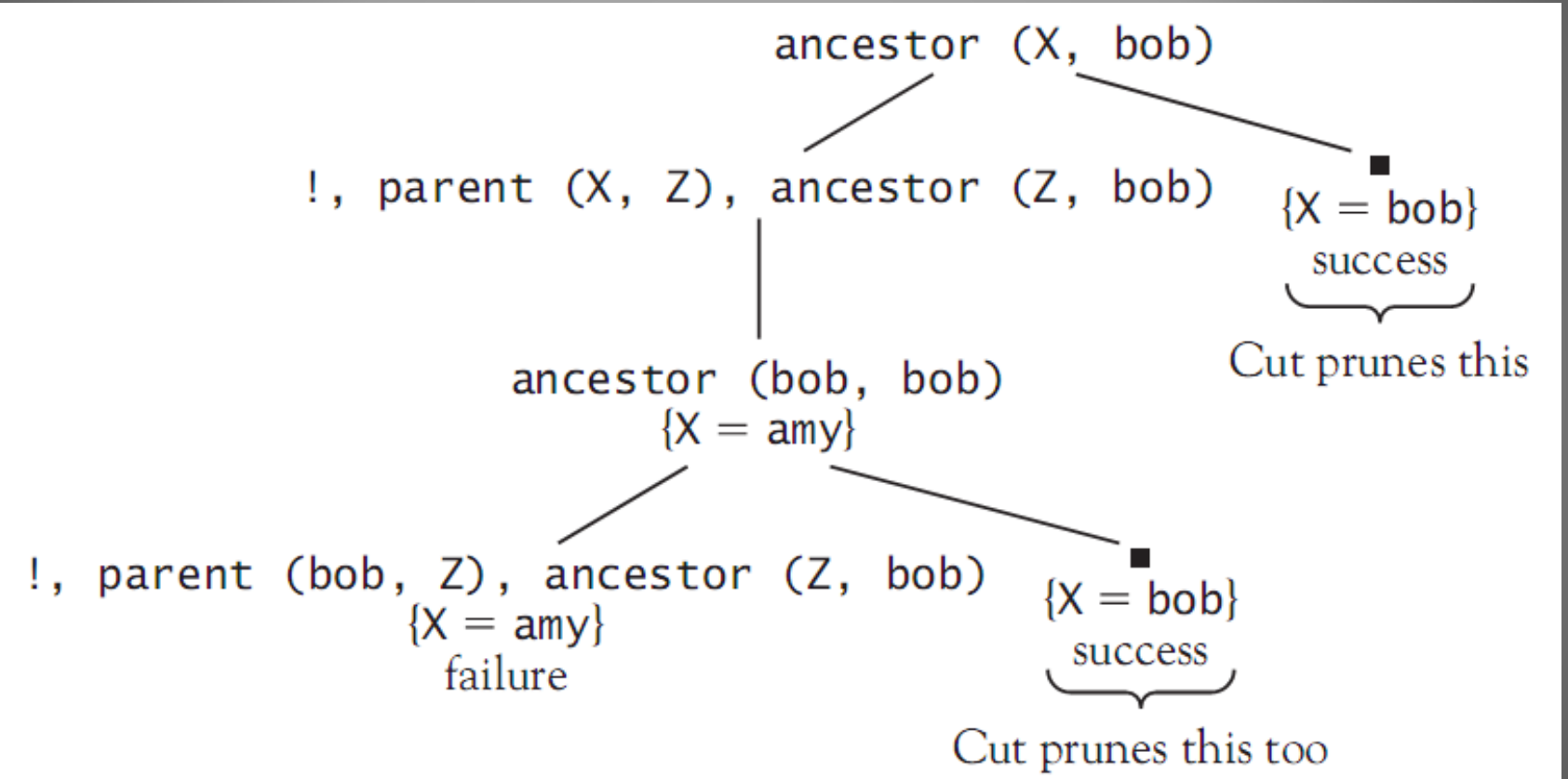


Figure 4.5 A further use of the cut prunes all solutions from Figure 4.2

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),
        R is R1+N.
```

Fibonacci Series

```
fib(0, 1) :- !.
fib(1, 1) :- !.
 fib(N, F) :- N > 1, N1 is N-1, N2 is N-2,
fib(N1, F1), fib(N2, F2), F is F1+F2.
```

- Memoising can be added to convert the complexity of O(2^n) to linear one
- Tabling effectively inverts the execution order for this case: it suspends the final addition (F is F1+F2) until the two preceding Fibonacci numbers have been added to the answer tables

https://www.swi-prolog.org/pldoc/man?section=tabling-memoize

Insertion sort

```
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;

        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

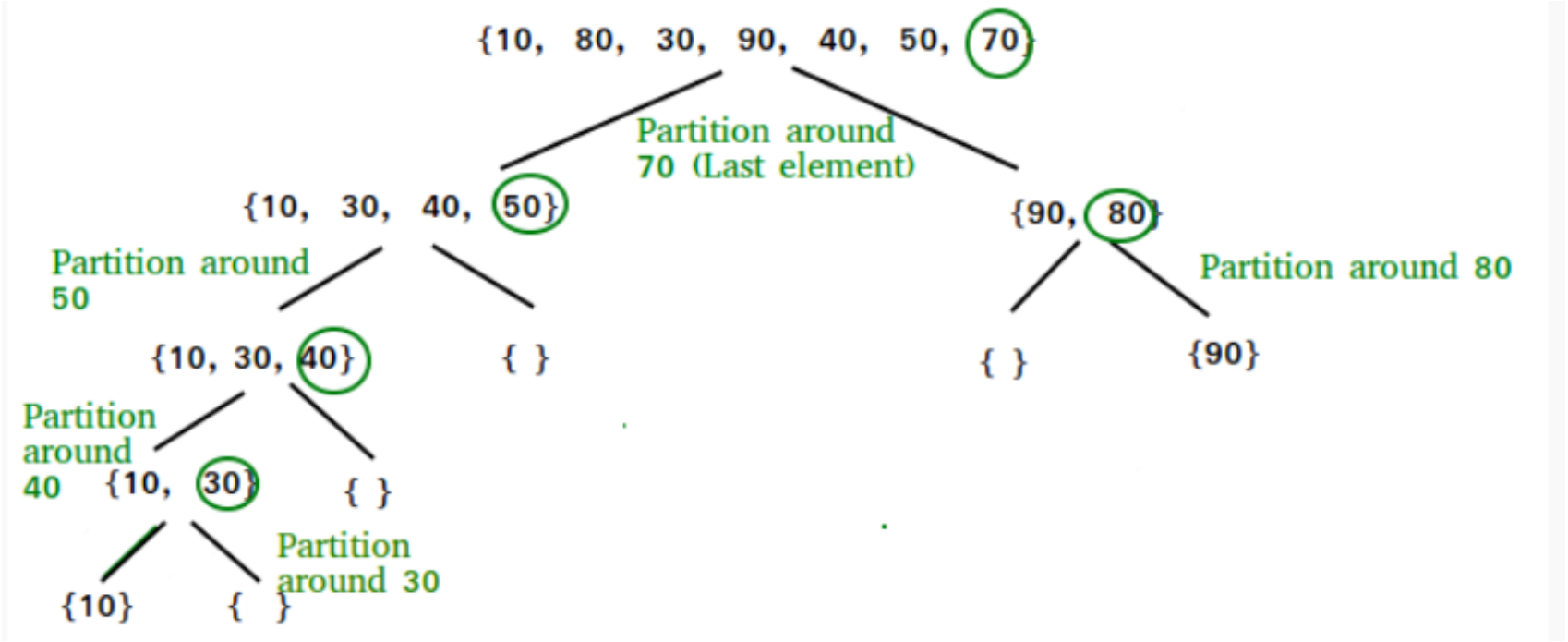
Insertion sort

1. insert(X,[],[X]).
2. insert(X,[H|T],[X,H|T]):- X =< H.
3. insert(X,[H|T],[H|NewT]):- X > H,
   insert(X,T,NewT).

1. isort([],[]).
2. isort([X|T],SList):- isort(T,ST), insert(X,ST,SList).

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

Quick Sort


{10, 80, 30, 90, 40, 50, 70}
Partition around 70 (Last element)
{10, 30, 40, 50}
{90, 80}
Partition around 50
Partition around 80
{10, 30, 40}
{ }
{ }
{90}
Partition around 40
{10, 30}
{ }
Partition around 30
{10}
{ }

Quick sort

```
qsort([], []).
qsort([H|T], S) :- split(H, T, L, R), qsort(L, L1), qsort(R, R1),
                   append(L1, [H|R1], S).

split(P, [A|X], [A|Y], Z) :- A < P,
split(P, [A|X], Y, [A|Z]) :- A >= P, split(P, X, Y, Z).
split(P, [], [], []).
```

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

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

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

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
- Thus a logic programming system has partially met its goal