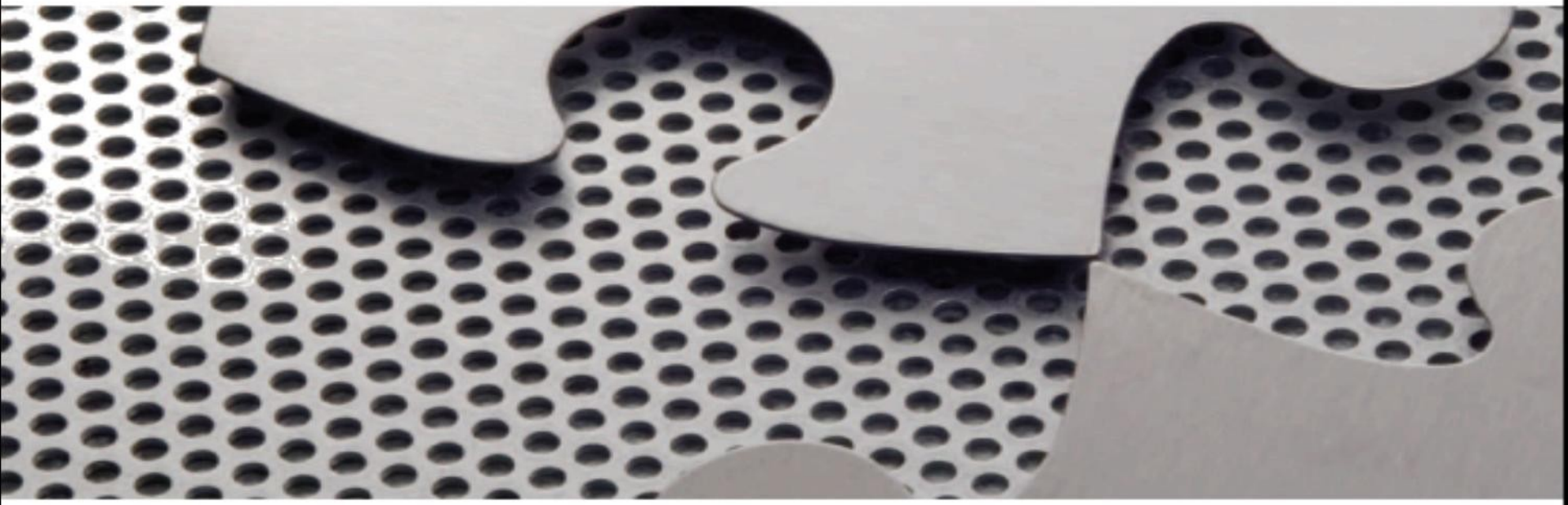
Programming Languages
Third Edition



Chapter 4

Logic Programming

Objectives

- Understand the nature of logic programming
- Understand Horn clauses
- Understand resolution and unification
- Become familiar with the Prolog language
- Explore the problems with logic programming
- Become familiar with the Curry language

Introduction

- **Logic**: the science of reasoning and proof
  - Existed since the time of ancient Greek philosophers
- Mathematical, or symbolic, logic: began with George Boole and Augustus De Morgan in the mid 1800s
- 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

Introduction (cont'd.)

- Logical statements can be used as formal specifications
- Together with axiomatic semantics, they can be used to prove the correctness of a program in a purely mathematical way
- Computers are used to implement the principles of mathematical logic
  - **Automatic deduction systems** or **automatic theorem provers** turn proofs into computation
  - Computation can be viewed as a kind of proof
  - Led to the programming language **Prolog**

Logic and Logic Programs

- Must understand mathematical logic
- **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.)

- First-order predicate calculus statement parts:
  - **Constants**: usually numbers or names
  - **Predicates**: names for functions that are true or false
  - **Functions**: functions that return non-Boolean values
  - **Variables that stand for as yet unspecified quantities**
  - **Connectives**: operations such as *and*, *or*, *not*, *implication* (→) *equivalence* (←→)
  - **Quantifiers**: operations that introduce variables
  - **Punctuation symbols**: parentheses, comma, period

Logic and Logic Programs (cont'd.)

- Example 1:
  - These English statements are logical statements:

    0 is a natural number.

    2 is a natural number.

    For all x, if x is a natural number, then so is the successor of x.

    21 is a natural number.

  - Translation into predicate calculus:

```
natural(0).

natural(2).

For all x, natural(x) → natural(successor (x)).

natural(21).
```

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)))
```

  - Fourth statement cannot be proved from the axioms so can assumed to be false
  - *x* in the third statement is a variable that stands for an as yet unspecified quantity

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*
- **Existential quantifier**: a predicate is true of at least one thing in the universe indicated by the variable
  - Ex: *there exists 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

Logic and Logic Programs (cont'd.)

- Example 2:

A horse is a mammal.
A human is a mammal.
Mammals have four legs and no arms, or two legs and two arms.
A horse has no arms.
A human has arms.
A human has no legs.

```
mammal(horse).
mammal(human).
for all x, mammal(x) →
  legs(x, 4) and arms(x, 0) or legs(x, 2) and arms(x, 2).
arms(horse, 0).
not arms(human, 0).
legs(human, 0).
```

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.)

- From Example 2, we can derive these statements:

```
legs(horse,4).
legs(human,2).
arms (human, 2).
```

- **Theorems**: statements derived from axioms
- **Logic programming**: a collection of statements is assumed to be axioms, and from them a desired fact is derived by the application of inference rules in some automated way

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.)

- Logical programming paradigm (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.)

- There are problems with logic programming systems
- 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.)

- Horn clauses can be used to express most, but not all, logical statements
- 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)).
```

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.)

- Example 6: logical statements

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

- Predicate calculus:

```
for all x, for all y,
(there exists z, parent(x, z) and parent(z, y))
→ grandparent (x, y).
```

- Horn clause:

```
parent(x, z) and parent(z, y) → grandparent(x, y).
```

Horn Clauses (cont'd.)

- Example 7: logical statements:

  For all x, if x is a mammal then x has two or four legs.

- Predicate calculus:

```
for all x, mammal(x) → legs(x, 2) or legs(x, 4).
```

- This may be approximated by these Horn clauses:

```
mammal(x) and not legs(x, 2) → legs(x, 4).
mammal(x) and not legs(x, 4) → legs(x, 2).
```

- In general, the more connectives that appear on the right of a → connective, the harder it is to translate into a set of Horn clauses

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
  - Similar to how context-free grammar rules are interpreted as procedure definitions in recursive descent parsing
  - Logic programs can be used to directly construct parsers

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
- 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.)

- Horn clauses do not supply the algorithms, only the properties that the result must have
- 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).
```

- Note the similarity 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.)

- 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
  - Examples:

```
mammal(human) ←.   — a fact
mammal(human) .    — a query or goal
```

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 $b \leftarrow a$ and $c \leftarrow b$
  - Resolution says $c \leftarrow a$
- Another way: combine left-hand and right-hand sides of both clauses, and cancel statements that match on both sides
- Example: given $b \leftarrow a$ and $c \leftarrow b$
  - Combine: $b,c, \leftarrow a,b$
  - Cancel the b on both sides: $c \leftarrow a$

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**

Resolution and Unification (cont'd.)

- Example 10: gcd with resolution and unification

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

- Goal:

```
← 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:

```
← 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

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

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

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

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

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
```

Arithmetic (cont'd.)

- Greatest common divisor algorithm
  - In generic Horn clauses:

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

  - In Prolog:

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

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.
yes

?- me = you.
no

?- me = X.
X = me

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

```
?- f(X) = g(X).
no

?- f(X) = f(a, b).
no

?- f(a, g(X)) = f(Y, b).
no

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

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
```

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

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

Unification (cont'd.)

- **Append** procedure:

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

- 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 (cont'd.)

- Append procedure rewritten more concisely:

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

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

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`

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).
```

Prolog's Search Strategy (cont'd.)

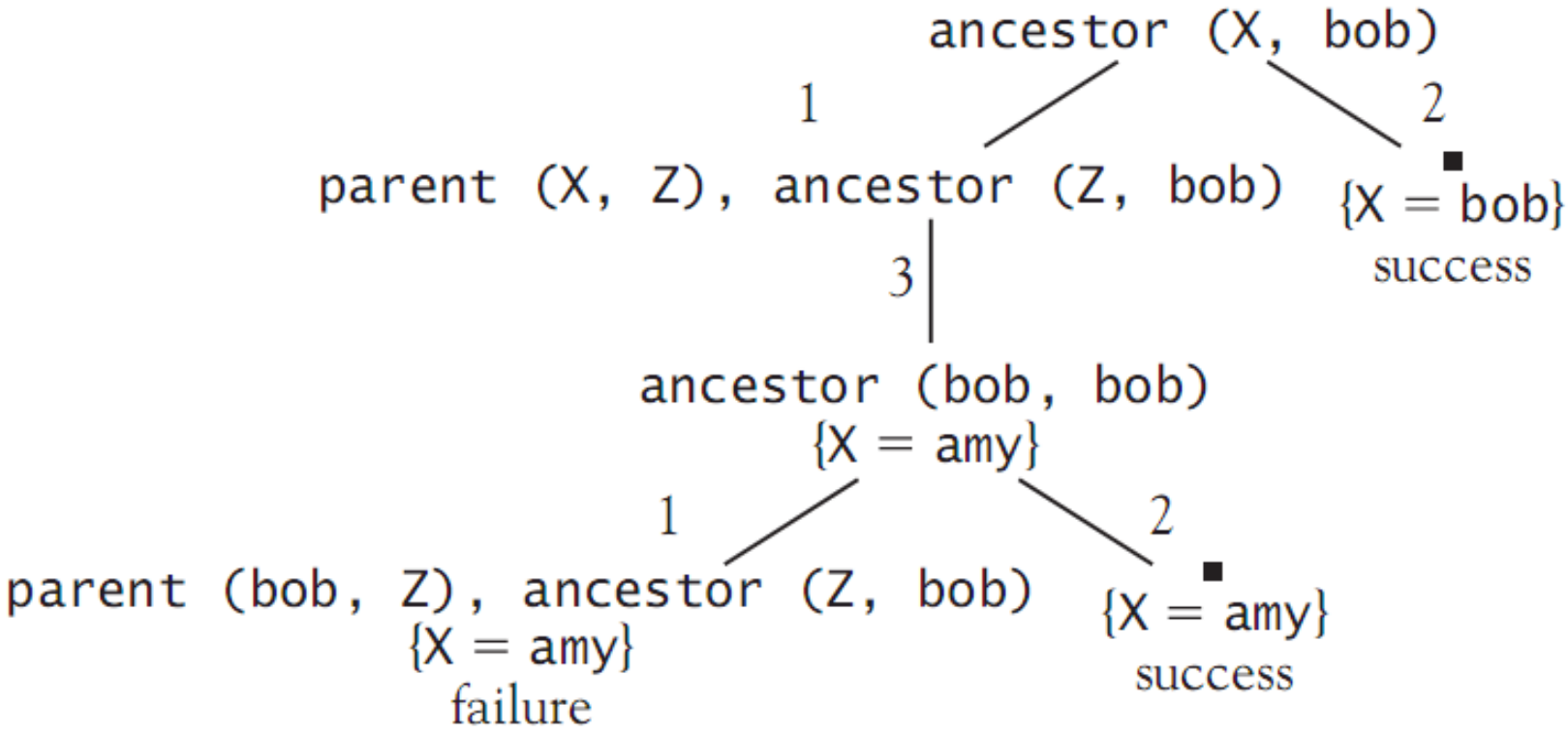


Figure 4.2 A Prolog search tree showing subgoals, clauses used for resolution, and variable instantiations

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
- Depth-first strategy is efficient: can be implemented in a stack-based or recursive fashion
  - Can be problematic if the search tree has branches of infinite depth

Prolog's Search Strategy (cont'd.)

- Example: same clauses in different order

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

(2) ancestor(X, X).

(3) parent(amy, bob).
```

- Causes Prolog to go into an infinite loop attempting to satisfy *ancestor (Z, Y)*, continually reusing the first clause
- 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 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.
```

```
?- printpieces([1, 2]).
[][1,2]
[1][2]
[1,2] []
no
```

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.
```

- The search tree has an infinite branch to the right

Loops and Control Structures (cont'd.)

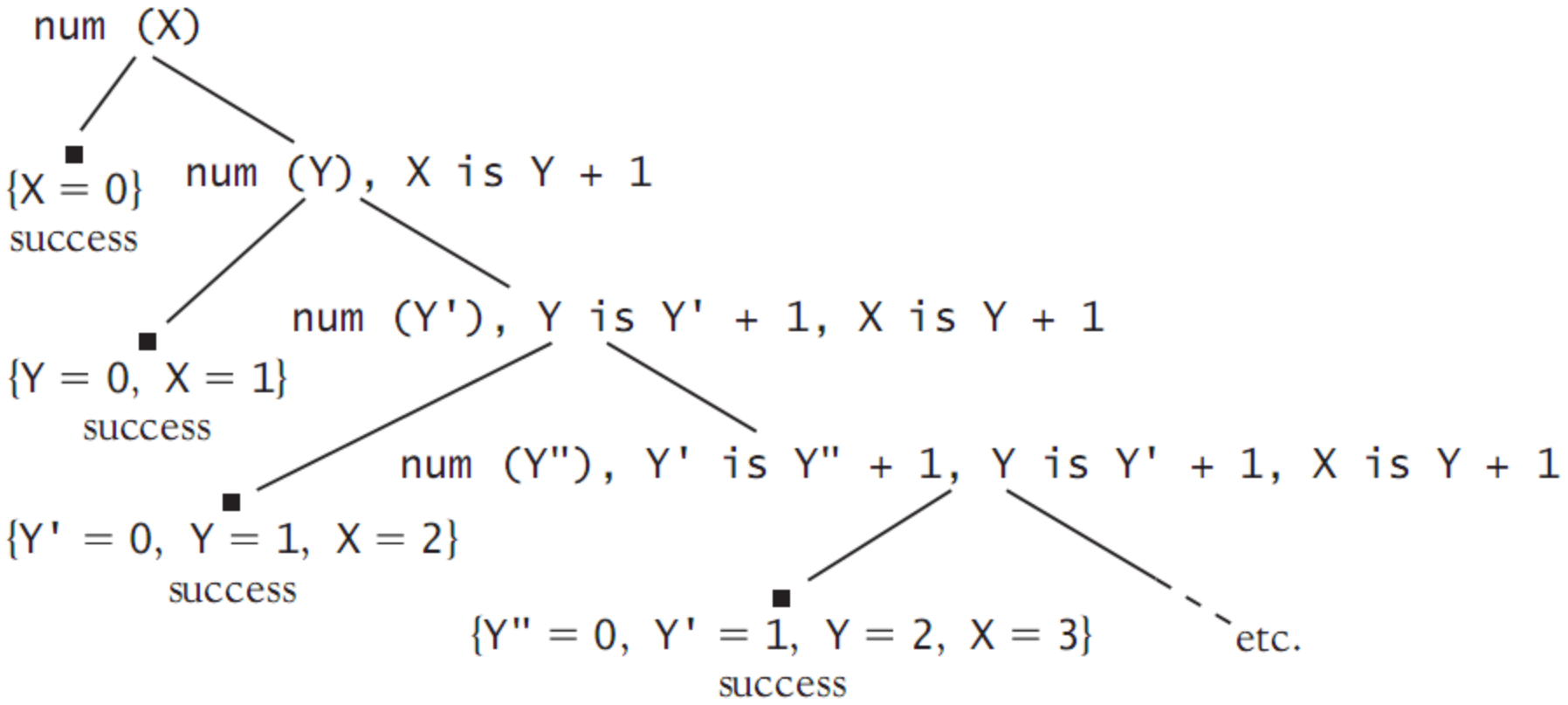


Figure 4.3 An infinite Prolog search tree showing repetitive computations

Loops and Control Structures (cont'd.)

- Example: trying to generate integers from 1 to 10

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

- Causes an infinite loop after `X = 10`, even though `X =< 10` will never succeed
- **cut** operator (written as `!`) freezes a choice when it is encountered

Loops and Control Structures (cont'd.)

- 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).
```

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

Loops and Control Structures (cont'd.)

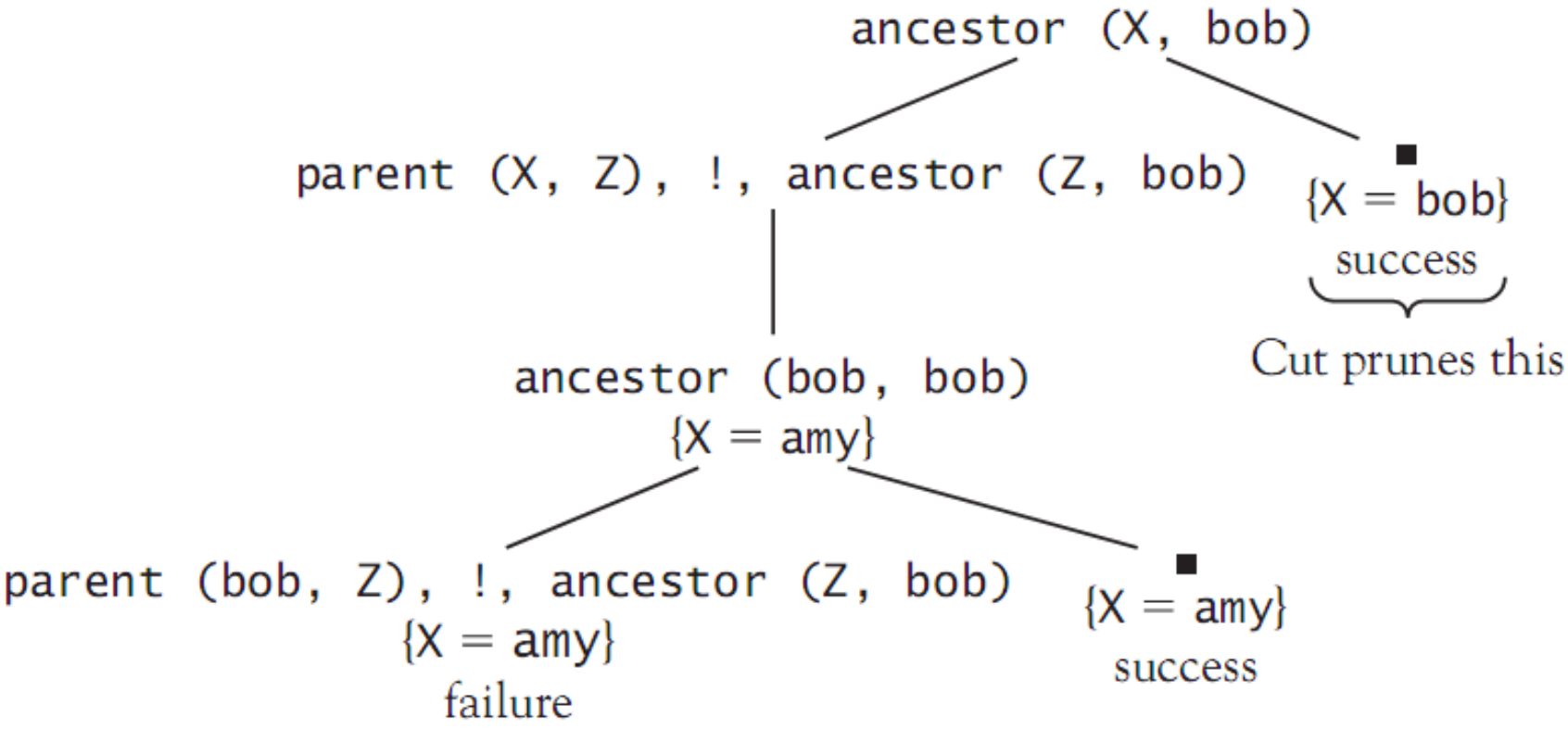


Figure 4.4 Consequences of the cut for the search tree of Figure 4.2

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).
```

  - No solutions will be found

Loops and Control Structures (cont'd.)

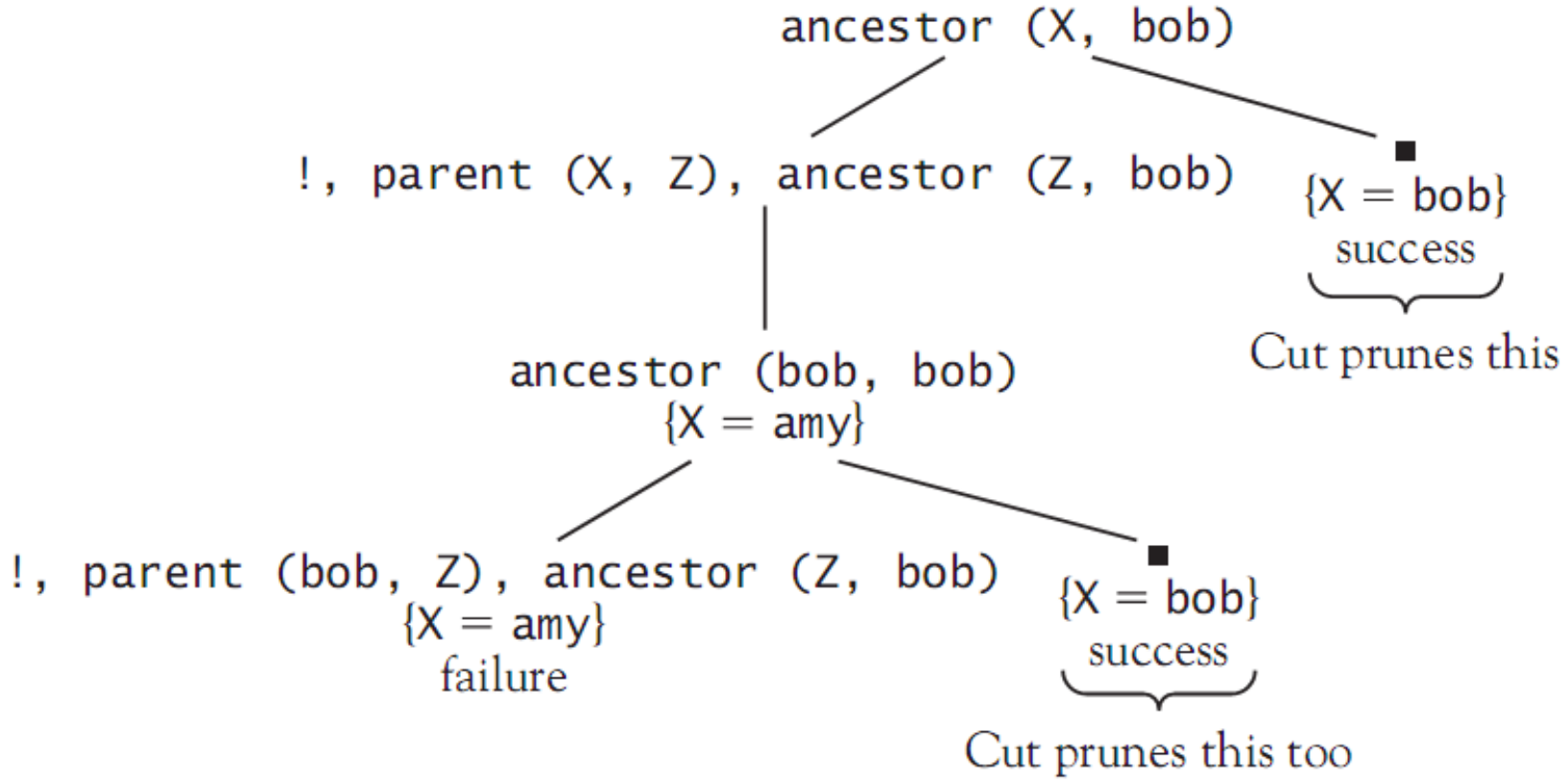


Figure 4.5 A further use of the cut prunes all solutions from Figure 4.2

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).
```

  - Both solutions will still be found since the right subtree of `ancestor(X, bob)` is not pruned
- 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

Loops and Control Structures (cont'd.)

- One solution to infinite loop shown earlier:

```
num(0).
num(X) :- num(Y), X is Y + 1.
writenum(I, J) :- num(X),
                  I =< X,
                  X =< J,
                  write(X), nl,
                  X = J, !,
                  fail.
```

  - `X = J` will succeed when the upper-bound `J` is reached
  - The cut will cause backtracking to fail, halting the search for new values of `X`

Loops and Control Structures (cont'd.)

- 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 (cont'd.)

```
primes(Limit, Ps) :- integers(2, Limit, Is),
                     sieve(Is, Ps).
integers(Low, High, [Low|Rest]) :-
            Low =< High, !, M is Low+1,
            integers(M, High, Rest).
integers(Low, High, []).
sieve([], []).
sieve([I|Is], [I|Ps]) :- remove(I, Is, New),
                         sieve(New, Ps).
remove(P, [], []).
remove(P, [I|Is], [I|Nis]) :-
      not(0 is I mod P), !, remove(P, Is, Nis).
remove(P, [I|Is], Nis) :-
      0 is I mod P, !, remove(P, Is, Nis).
```

Figure 4.6 The Sieve of Eratosthenes in Prolog (adapted from Clocksin and Mellish [1994])

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

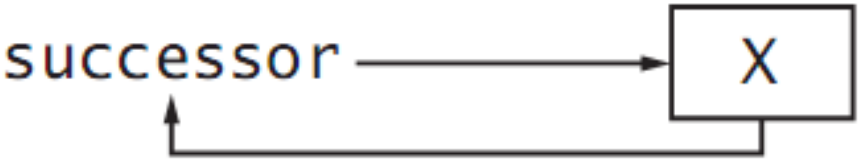


Figure 4-7: Circular structure created by unification

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:

  $p(a)$ and (there exists x, 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 (cont'd.)

```
sorted([]).
sorted([X]).
sorted([X, Y|Z]) :- X =< Y, sorted([Y|Z]).

permutation([], []).
permutation(X, [Y|Z]) :- append(U, [Y|V], X),
                         append(U, V, W),
                         permutation(W, Z).
```

Figure 4.8 The definitions of the `permutation` and `sorted` properties in Prolog

Control Information in Logic Programming (cont'd.)

- This is a mathematical definition of what it means for a list of numbers to be sorted in increasing order
  - As a program, it is one of the slowest possible sorts
  - Permutations of the unsorted list are generated until one of them happens to be sorted
- 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

Control Information in Logic Programming (cont'd.)

```
qsort([], []).
qsort([H|T], S) :- partition(H, T, L, R),
                   qsort(L, L1),
                   qsort(R, R1),
                   append(L1, [H|R1], S).
partition(P, [A|X], [A|Y], Z) :- A < P,
                                 partition(P, X, Y, Z).
partition(P, [A|X], Y, [A|Z]) :- A >= P,
                                 partition(P, X, Y, Z).
partition(P, [], [], []).
```

Figure 4.9 A `quicksort` program in Prolog (adapted from Clocksin and Mellish [1994])

Curry: A Functional Logic Language

- In a functional language, a program is a set of function definitions that specify rules for operating on data to transform it into other data
- In a logic language, a program is a set of rules and facts from which a proof is constructed for a solution to a problem
- Each of these has some specific disadvantages
- The language Curry brings together the advantages of functional and logic programming in a single language

Functional Programming in Curry

- Curry is an extension of Haskell
  - Retains the syntax and semantics of Haskell for functional programming
  - Adds new syntax and semantics for logic programming
- Function definitions are sets of equations as in Haskell
- Curry uses lazy evaluation

Adding Nondeterminism, Conditions, and Backtracking

- A pure functional language supports only deterministic computation
  - Application of a function to a given set of arguments always produces the same value
- Problems such as flipping a coin are underspecified, as their solutions come from a set of values
- Curry supports nondeterminism by allowing a set of equations for a function to be tried in no particular order, using the choice operator ?

Adding Nondeterminism, Conditions, and Backtracking (cont'd.)

- Example:

```
x ? y = x
x ? y = y
```

- Curry does not automatically try the first equation
- If one fails, another equation will be tried
- A nondeterministic function for flipping a coin:

```
flipCoin = 0 ? 1
```

- For the sorting problem, we can use Curry's nondeterminism and backtracking to produce a simple implementation

Adding Nondeterminism, Conditions, and Backtracking (cont'd.)

- Function *sorted* expects a list as an argument and returns a sorted list of the same elements:

```
sorted []   = []
sorted [x]  = [x]
sorted (x:y:ys) | x <= y
                = x : sorted(y:ys)
```

- Third equation includes a **condition** (to the right of the | symbol) that permits evaluation of its right side only if the first element in the list is less than or equal to the second element

Adding Nondeterminism, Conditions, and Backtracking (cont'd.)

- Function `permutation` inserts the first element of a nonempty list into a permutation of the rest of that list:

```
permutation []     = []
permutation (x:xs) = insert x (permutation xs)
```

- Function `insert` places an element at an arbitrary position in a list
  - Defined nondeterministically for nonempty lists

```
insert x ys     = x : ys
insert x (y:ys) = y : insert x ys
```

Adding Logical Variables and Unification

- Logical variables and unification give Curry the ability to solve equations with unknown or partial information
  - Involves viewing some variables as free in the sense that they can be instantiated in a way to satisfy a set of equations that includes them
- Curry uses the symbol =:= to specify an equation to be solved in this manner
- Example:

```
zs ++ [2] =:= [1, 2]
```