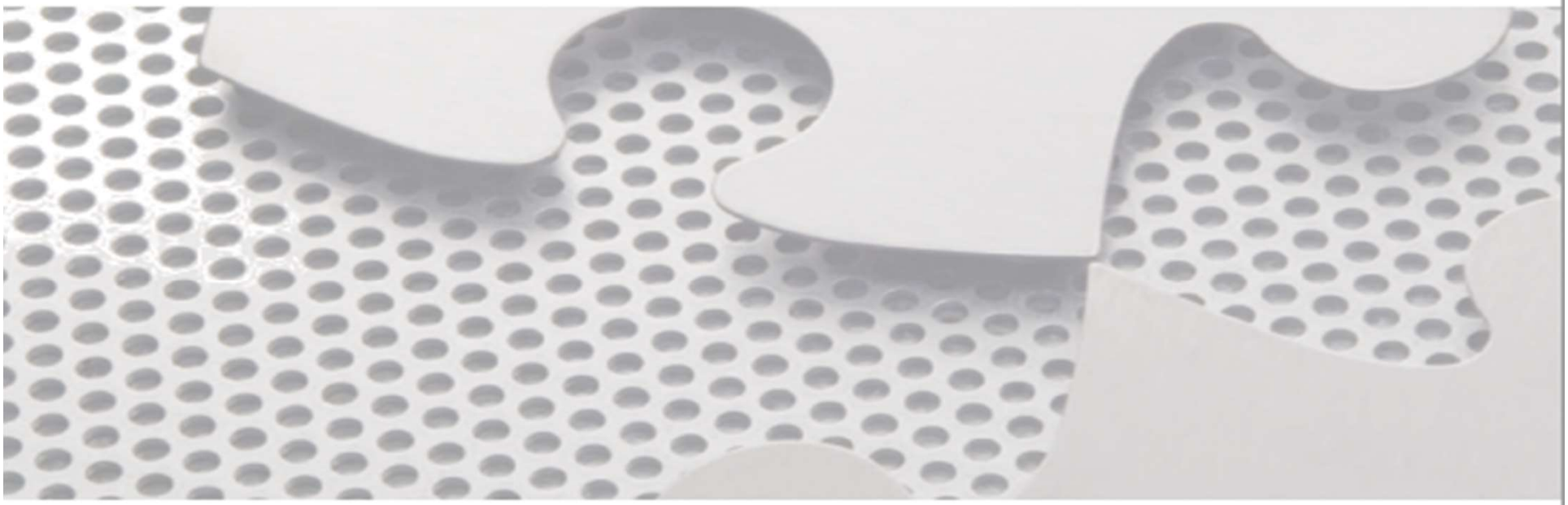
Logic Programming



Chapter 4

Companion slides from the book

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

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

Introduction (cont'd.)

- Logical statements can be used as formal specifications
- 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

- 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

0 is a natu

2 is a natu

For all x, i al number, then so is the successor of x.

−1 is a n ber

predicate

constant

Bound Variable

  - Translation into pr ulus

```
natural(0).
natural(2).
For all x, natural(x)    natural(successor (x)).
```

  - natural(-1).

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)))
```

  - *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, 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.)

- Example 3:

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 a

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

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**
    - `there exists y, legs(Human, 2) ?`
    - `yes: …`

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.)

- 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)).
```

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.)

- 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).
```

- 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.)

- 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 $c \leftarrow a$

- 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),
```

  - To get this subgoal:

```
gcd(10, 5, x).
← gcd(5, 0, x).
```

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

Another Example

```
ancestor(x, y)    parent(x, z), ancestor(z, y).
ancestor(x, x).
parent(amy, bob).
```

if we provide the query:

```
ancestor(x, bob).
```

- X=bob or amy?

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