CS-602 - Design of Problem Solvers

Chapter 3 - Queries and Updates for Logic Problem Solvers

Dr. Mahdi Khemakhem

Department of Computer Science
College of Computer Engineering and Science
Prince Sattam bin Abdulaziz University

AY - 2025/2026

Outline

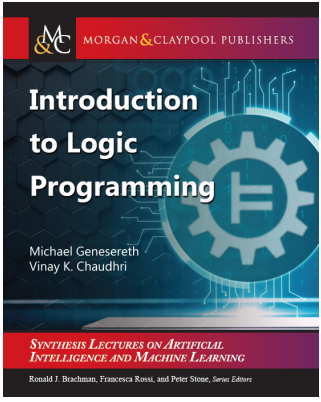
1. Preview

2. Queries
2.1 Query Syntax
2.2 Query Semantics
2.3 Safety
2.4 Predefined Concepts
2.5 Examples
2.6 Key Takeaways
2.7 Exercises
2.8 Homework

3. Updates
3.1 Update Syntax
3.2 Update Semantics
3.3 Simultaneous Updates
3.4 Examples
3.5 Key Takeaways
3.6 Exercises
3.7 Homework

Materials

- **Textbook:** "Introduction to logic programming", 2022, by *Genesereth, Michael, and Vinay K. Chaudhri*
- $\Rightarrow$ **Read Chapters 3 and 4 for this chapter's material**



Outline

1. Preview

2. Queries

3. Updates

Preview

Chapter 2 Recap

In Chapter 2, we learned to represent application states as **datasets**. However, large datasets pose challenges for information retrieval and modification.

This Chapter Covers:

1. **Queries** (Section 2):
   - Various ways of *querying* datasets
   - Building queries to find **specific information**
   - Three types: true/false, fill-in-blanks, compound questions
2. **Updates** (Section 3):
   - Techniques for *updating* dataset information
   - Transforming datasets efficiently
   - Focusing on **changed factoids only**

Key Goal: Efficient information retrieval and modification in large datasets.

Outline

1. Preview

2. Queries
2.1 Query Syntax
2.2 Query Semantics
2.3 Safety
2.4 Predefined Concepts
2.5 Examples
2.6 Key Takeaways
2.7 Exercises
2.8 Homework

3. Updates

Introduction (1/2)

Three Types of Queries

Queries allow us to extract information from datasets with increasing sophistication.

1. **True-or-False Queries:**
   - *Simplest form* of query
   - Check if a **factoid is true** in the dataset
   - *Example: Is Art the parent of Bob?*
2. **Fill-in-the-Blanks Queries:**
   - Find **values** that make a pattern true
   - Use *variables* as placeholders
   - *Examples: Who are Art's children? Who are Bill's parents? Find all parent-child pairs*
3. **Compound Queries:**
   - Use *Boolean combinations* (AND, OR, NOT)
   - *Examples: Is Art parent of Bob* **OR** *Bud? Find people with sons* **AND** *no daughters*

Introduction (2/2)

Section Roadmap

We extend the **dataset language** with features to express powerful queries.

Topics Covered:

1. **Query Syntax:**
   - Define the *syntax* of query language
   - Variables, atoms, literals, and query rules
2. **Query Semantics:**
   - Define the *meaning* of queries
   - How queries are evaluated
3. **Safety:**
   - Important syntactic **restriction**
   - Ensures queries are well-defined
4. **Predefined Concepts:**
   - Arithmetic operators and functions
   - Increase query **expressiveness**

Query Syntax (1/6)

Query Language Extensions

The query language extends datasets with two powerful features: *Variables* and *Query Rules*.

Key Components:

1. **Variables:**
   - Enable **fill-in-the-blanks** queries
   - Act as placeholders for unknown values
   - Example: "Who are Art's children?"
2. **Query Rules:**
   - Express **compound conditions**
   - Support logical operations:
     - **Negations**: condition is false
     - **Conjunctions**: all conditions true (AND)
     - **Disjunctions**: at least one condition true (OR)

Query Syntax (2/6)

Variable, Atom, and Literal

- A *variable* is either *a lone underscore* or a *string of letters*, *digits*, and *underscores beginning with an uppercase letter*.
  For example: `_`, `X23`, `X_23`, and `Somebody` are all **variables**.
- An *atomic sentence*, or *atom*, is **analogous** to a **factoid in a dataset except that the arguments may include variables as well as symbols**.
  For example: if `p` is a **binary predicate** and `a` is a **symbol** and `Y` is a **variable**, then `p(a,Y)` is an **atomic sentence**.
- A *literal* is either an **atom** (called a **positive literal**) or a **negation of an atom** (called a **negative literal**). *In what follows, we write negative literals using the negation sign* $\sim$.
  For example: if `p(a,b)` is an **atom**, then $\sim$`p(a,b)` denotes the **negation of this atom**. Both are literals.

Query Syntax (3/6)

Query Rule

- A *query rule* is an **expression** consisting of:
  - a **distinguished atom**, called the **head** (i.e., the overall **goal**).
    →The predicate in the **head** of a query rule **must be a new predicate** (i.e., not one in the vocabulary of our dataset).
  - a **collection of zero or more literals**, called the **body**. The **literals** in the **body** are called **subgoals**.
    →All of the predicates in the **body** of a query rule **must be dataset predicates**.

→*In what follows, we write rules as in the example shown below.*

```
goal(a,b) :- p(a,b) & ~q(b)
```

Here, `goal(a,b)` is the *head*; `p(a,b) & ~q(b)` is the *body*; and `p(a,b)` and `~q(b)` are *subgoals*.

Query Syntax (4/6)

Power of Query Rule!

- As we shall see in the next section, a **query rule** is something like a **reverse implication** -it is a statement that the **head of the rule** is **true** whenever the **subgoals** are **true**.
  For example: the rule in previous slide states that `goal(a,b)` is true if `p(a,b)` is true and `q(b)` is not true.
- The **expressive power** of **query rules** is **greatly enhanced through the use of variables**. Consider, for example, the rule shown below.

  ```
  goal(X,Y) :- p(X,Y) & ~q(Y)
  ```

  This is a **more general version** of the rule shown in previous slide. Instead of applying to just the specific objects `a` and `b` **it applies to all objects**.
  →**In this case, the rule states that goal is true of any object `X` and any object `Y` if `p` is true of `X` and `Y` and `q` is not true of `Y`.**

Query Syntax (5/6)

Query

- A **query** is a **non-empty**, **finite** set of query rules.
  - Typically, a **query consists of just one rule**.
    In fact, most **Logic Programming Solvers** do not support queries with multiple rules.

→However, queries with multiple rules are sometimes useful and do not add any major complexity, so in what follows we allow for the possibility of queries with multiple rules.

Query Syntax (6/6)

Query Set

- The result of applying a **set of queries** to a dataset is the **union of the results of applying the queries to the dataset.**

Example:

Dataset:

`{p(a,b) p(b,c)}`

Queries:

`goal(X) :- p(X,Y)` → `{goal(a), goal(b)}`

`goal(Y) :- p(X,Y)` → `{goal(b), goal(c)}`

Results:

`{goal(a), goal(b), goal(c)}`

→**Note that a query set is effectively a disjunction.**

Query Semantics (1/6)

Instance

An **instance** of an **expression** (**atom**, **literal**, or **rule**) is one in which **all variables have been consistently replaced by ground terms** (i.e., terms without variables).

For example: if we have a language with symbols `a` and `b`, then the **instances** of `goal(X,Y) :- p(X,Y) & ~q(Y)` are shown below.

```
goal(a,a) :- p(a,a) & ~q(a)
goal(a,b) :- p(a,b) & ~q(b)
goal(b,a) :- p(b,a) & ~q(a)
goal(b,b) :- p(b,b) & ~q(b)
```

Query Semantics (2/6)

Examples

- **Dataset:** {p(a,b) p(b,c) p(c,d) p(d,c)}
- **Query:** goal(X,Y) :- p(X,Y) & ∼p(Y,X)
- **Result:** goal(a,b) goal(b,c)

Query Instances:

goal(a,a) :- p(a,a) & ∼p(a,a)
goal(a,b) :- p(a,b) & ∼p(b,a)
goal(a,c) :- p(a,c) & ∼p(c,a)
goal(a,d) :- p(a,d) & ∼p(d,a)
goal(b,a) :- p(b,a) & ∼p(a,b)
goal(b,b) :- p(b,b) & ∼p(b,b)
goal(b,c) :- p(b,c) & ∼p(c,b)
goal(b,d) :- p(b,d) & ∼p(d,b)
goal(c,a) :- p(c,a) & ∼p(a,c)
goal(c,b) :- p(c,b) & ∼p(b,c)
goal(c,c) :- p(c,c) & ∼p(c,c)
goal(c,d) :- p(c,d) & ∼p(d,c)
goal(d,a) :- p(d,a) & ∼p(a,d)
goal(d,b) :- p(d,b) & ∼p(b,d)
goal(d,c) :- p(d,c) & ∼p(c,d)
goal(d,d) :- p(d,d) & ∼p(d,d)

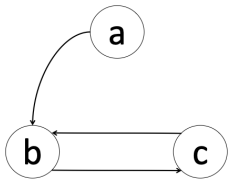
Query Semantics (3/6)

Illustration (1/3)

To illustrate definitions seen in previous slides, consider a **dataset** describing a **small directed graph**.

- In the **sentences** below, we use **symbols** `a`, `b`, and `c` to designate the **nodes of the graph**, and we use the `p` **relation** to designate the **arcs of the graph**.

```
p(a,b)
p(b,c)
p(c,b)
```



Query Semantics (4/6)

Illustration (2/3)

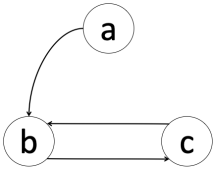
Now suppose we were given the following query:

```
goal(X) :- p(X,Y) & p(Y,X)
```

Here, the **predicate goal** is defined to be **true** of **every node that has an outgoing arc to another node and also an incoming arc from that node**.

Since there are **two variables** here and **three symbols**, there are **nine instances** of this rule:

```
goal(a) :- p(a,a) & p(a,a)    goal(a) :- p(a,b) & p(b,a)
goal(a) :- p(a,c) & p(c,a)    goal(b) :- p(b,a) & p(a,b)
goal(b) :- p(b,b) & p(b,b)    goal(b) :- p(b,c) & p(c,b)
goal(c) :- p(c,a) & p(a,c)    goal(c) :- p(c,b) & p(b,c)
                goal(c) :- p(c,c) & p(c,c)
```



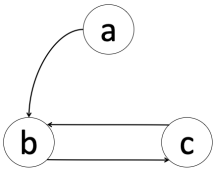
Query Semantics (5/6)

Illustration (3/3)

The body in the first of these instances is not satisfied.

In fact, the **body is true** only in the **sixth** and **eighth** instances. Consequently, the **extension of this query contains just the two atoms shown below**.

```
goal(b)
goal(c)
```



Query Semantics (6/6)

Semantics

The definition of **semantics in terms of rule instances** is simple and clear.

- However, **Logic Programming solvers** typically **do not implement query processing in this way**.
- **There are more efficient ways of computing such extensions.**

→*In subsequent chapters, we look at some algorithms of this sort.*

Try the graph example on Sierra

Safety (1/3)

A **query rule** is **safe** **if and only if** **every variable** **that appears in the head** or **in any negative literal in the body** **also appears** in **at least one positive literal in the body**.

- **The rule shown below is safe**. Every variable in the head and every variable in the negative subgoal appears in a positive subgoal in the body.
  $\rightarrow$*Note that it is okay for the body to contain variables that do not appear in the head.*

```
goal(X) :- p(X,Y,Z) & ~q(X,Z)
```

- **The two rules shown below are not safe.** The first rule is not safe because the variable `Z` appears in the head but does not appear in any positive subgoal. The second rule is not safe because the variable `Z` appears in a negative subgoal but not in any positive subgoal.

```
goal(X,Y,Z) :- p(X,Y)
goal(X,Y,X) :- p(X,Y) & ~q(Y,Z)
```

Safety(2/3)

Why Safety Matters?

Consider the **unsafe** rule: `goal(X,Y,Z) :- p(X,Y)`

- **Given:** Suppose `p(a,b)` is true in our database
- **Rule Application:**
  - The body `p(X,Y)` is satisfied with `X=a` and `Y=b`
  - Therefore, we can conclude `goal(a,b,Z)` is true
- **The Problem:** What value should `Z` have?
  - Variable `Z` appears in the head but not in the body
  - **Intuitively**, `Z` could be *anything*
  - This creates **infinitely many possible conclusions** – practically problematic!

Safety(3/3)

Consider the **unsafe** rule: `goal(X,Y,X) :- p(X,Y) & ∼q(Y,Z)`

- **Given:** Suppose `p(a,b)` is true in our database and `q(b,c)` is false
- **Rule Application:**
  - The body `p(X,Y) & ∼q(Y,Z)` is satisfied with `X=a`, `Y=b`, and `Z=c`
  - Therefore, we can conclude `goal(a,b,a)` is true
- **The Problem:** What value should `Z` have?
  - Variable `Z` appears in a negative subgoal but not in any positive subgoal
  - **Intuitively**, `Z` could be *anything*
  - This creates **infinitely many possible conclusions** – practically problematic!

Predefined Concepts (1/7)

In **practical logic programming languages**, it is common to **predefine useful concepts**.

- These typically include **arithmetic functions** (such as plus, times, max, min), **string functions** (such as concatenation), **equality** and **inequality**, **aggregates** (such as countofall), and so forth.
- In Epilog, equality and inequality are expressed using the relations same and distinct. The sentence `same(`σ`,r)` is true **if and only if (iff)** σ and `r` are identical. The sentence `distinct(`σ`,r)` is true **iff** σ and `r` are different.

Predefined Concepts (2/7)

Examples

- `plus(2,3)` → 5
- `stringappend("abc","def")` → "abcdef"
- `stringify(vinay)` → "vinay"
- `symbolize("vinay")` → vinay
- `min(plus(2,3),times(2,3))` → 5

Many predefined concepts in **EpilogJS** are available via one of the following two links: WEB - PDF.

Predefined Concepts (3/7)

- The `evaluate` **relation** is used to represent equations involving **predefined functions**. *For example:* we would write `evaluate(plus(times(3,3),times(2,3),1),16)` to represent the equation $3^2 + 2 \times 3 + 1 = 16$.
- If `height` and `width` are **binary predicates** relating a **figure** and its **height** and **width**, **we can define the area of the object as shown below:**

```
goal(X,A) :- height(X,H) & width(X,W) & evaluate(times(H,W),A)
```

The area of `X` is `A` if the height of `X` is `H` and the width of `X` is `W` and `A` is the result of multiplying `H` and `W`.

Predefined Concepts (4/7)

Examples

- `same(a,a)` is true
- `same(a,b)` is false
- `distinct(a,a)` is false
- `distinct(a,b)` is true
- `mutex(a,b,c)` is true

→`same` **is not ordinary equality (e.g. 2+2 = 4)**

- `same(plus(2,2),4)` is false
- `distinct(plus(2,2),4)` is true

→Use `evaluate` **to get ordinary equality**

- `evaluate(plus(2,2),V) & same(V,4)` is true
- `evaluate(plus(2,2),4)` is true

Predefined Concepts (5/7)

Syntactic Restrictions

- In **logic programming languages** that **provide such predefined concepts**, **there are usually syntactic restrictions on their use**.
  For example:
  - if a **query** contains a **subgoal** with a **comparison relation** (such as `same` and `distinct`), then **every variable** that **occurs in that subgoal must occur in at least one positive literal in the body and that occurrence must precede the subgoal with the comparison relation**.
  - If a **query** uses `evaluate` in a **subgoal**, then **any variable** that **occurs in the first argument of that subgoal must occur in at least one positive literal in the body and that occurrence must precede the subgoal with the arithmetic relation**.

Predefined Concepts (6/7)

Aggregate Operators

In practical logic programming languages, it is also common to **include predefined aggregate operators**, such as `setofall` and `countofall`.

- **Aggregate operators** are **typically represented as relations with special syntax.**
  - *For example:* the following rule uses the `countofall` operator to request the number of a person's children. `N` is the number of children of `X` **iff** `N` is the count of all `Y` such that `X` is the parent of `Y`.

```
goal(X,N) :- person(X) & evaluate(countofall(Y,parent(X,Y)),N)
```

  → As with **special relations**, there are **syntactic restrictions** on their use. In particular, **aggregate subgoals** *must be safe in that all variables in the second argument must be included in the first argument* or *must be used within positive subgoals of the rule containing the aggregate*.

Predefined Concepts (7/7)

Examples

- Data Set: `{p(a,b), p(a,c), p(b,d)}`
- Query 1: `goal(X,L) :- p(X,Y) & evaluate(countofall(Z,p(X,Z)),L)`
- Result 1: `goal(a,2), goal(b,1)`
- Query 2: `goal(X,L) :- p(X,Y) & evaluate(setofall(Z,p(X,Z)),L)`
- Result 2: `goal(a,[b,c]), goal(b,[d])`

Example 1: Kinship (1/4)

Consider a variation of the **Kinship application** introduced in Chapter 2.

- In this case, our **vocabulary** consists of symbols (representing people) and a **binary predicate** `parent` (which is true of two people if and only if the person specified as the first argument is the parent of the person specified as the second argument).
- Given data about parenthood expressed using this vocabulary, we can write **queries** to **extract information about other relationships as well**.

Example 1: Kinship (2/4)

For example: we can find **grandparents** and **grandchildren** by writing the **query** shown below:

```
goal(X,Z) :- parent(X,Y) & parent(Y,Z)
```

- A person `X` is the **grandparent** of a person `Z` if `X` is the **parent** of a person `Y` and `Y` is the **parent** of `Z`.
- The **variable** `Y` here is a **thread variable** that connects the **first subgoal** to the **second but does not itself appear in the head of the rule**.

Try it on Sierra

Example 1: Kinship (3/4)

In general, we can write queries with multiple rules.
For example: we can collect all of the people mentioned in our dataset by writing the following multi-rule query:

```
goal(X) :- parent(X,Y)
goal(Y) :- parent(X,Y)
```

- In this case the **conditions are disjunctive** (at least one must be true), whereas the **conditions in the grandfather case are conjunctive** (both must be true).

Example 1: Kinship (4/4)

In some cases, it is helpful to use built-in relations in our queries.
For example: For example, we can ask for all pairs of people who are siblings by writing the query rule shown below:

```
goal(Y,Z) :- parent(X,Y) & parent(X,Z) & distinct(Y,Z)
```

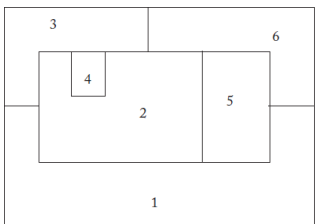
- We use the **distinct condition** here to avoid listing a person as his own sibling.

Try it on Sierra

Example 2: Map Coloring (1/4)

Consider the problem of **coloring planar maps** using **only four colors**, the idea being to assign each region a color so that no two adjacent regions are assigned the same color.

- A typical map is shown at the right. Here we have six regions. Some are adjacent to each other, meaning that they cannot be assigned the same color. Others are not adjacent, meaning that they can be assigned the same color.



Example 2: Map Coloring (2/4)

- We can enumerate the colors to be used as shown below:

```
color(red) color(green) color(blue) color(purple)
```

The **constants** `red`, `green`, `blue`, and `purple` stand for the colors red, green, blue, and purple, respectively.

Example 2: Map Coloring (3/4)

- In the case of the map shown in the previous slide, our **goal** is to **find six colors** (one for each region of the map) such that **no two adjacent regions have the same color**. We can express this goal by writing the query shown below.

```
goal(C1,C2,C3,C4,C5,C6) :- color(C1) & color(C2) & color(C3) &
                           color(C4) &
  color(C5) & color(C6) & distinct(C1,C2) & distinct(C1,C3) &
                          distinct(C1,C5) &
distinct(C1,C6) & distinct(C2,C3) & distinct(C2,C4) & distinct(C2,C5)
      & distinct(C2,C6) & distinct(C3,C4) & distinct(C3,C6) &
                          distinct(C5,C6)
```

Try it on Sierra

Example 2: Map Coloring (4/4)

- **Evaluating the previous query** will **result in 6-tuples of colors** that **ensure that no two adjacent regions have the same color**.
- In problems like this one, **we usually want only one solution rather than all solutions**. However, **finding even one solution is such cases can be costly**.
- In Section 3 "Updates", **we will discuss ways of writing such queries that makes the process of finding such solutions more efficient**.

Key Takeaways

Key Takeaways

- A **query** is a **set of one or more query rules**. Each **query rule** consists of a **head** and a **body**, separated by the symbol `:-`. The head is a single **atom**, and the body is a **conjunction of one or more literals**.
- An **instance** of a query rule is obtained by **substituting symbols for the variables** in the rule.
- The **semantics** of a query is defined in terms of the **set of all facts that can be deduced on the basis of the rules in the program**.
- A **query rule** is **safe** if and only if every variable that appears in the head or in any negative literal in the body also appears in at least one positive literal in the body.
- In practical logic programming languages, it is common to include **predefined concepts**, such as arithmetic functions, string functions, equality and inequality, and aggregate operators.

Exercise 3.1

For each of the following strings, say whether it is a syntactically legal query.

1. `goal(X) :- p(a,f(f(X)))`
2. `goal(X,Y) :- p(X,Y) & ~p(Y,X)`
3. `~goal(X,Y) :- p(X,Y) & p(Y,X)`
4. `goal(P,Y) :- P(a,Y)`
5. `goal(X) :- p(X,b) & p(X,p(b,c))`

Solution 3.1

1. Yes
2. Yes
3. No – negation cannot appear in the head of a rule
4. No – variables cannot appear in predicate position
5. No – `p(b,c)` is not a legal term

Exercise 3.2

Say whether each of the following queries is safe.

1. `goal(X,Y) :- p(X,Y) & p(Y,X)`
2. `goal(X,Y) :- p(X,Y) & p(Y,Z)`
3. `goal(X,Y) :- p(X,X) & p(X,Z)`
4. `goal(X,Y) :- p(X,Y) & ~p(Y,Z)`
5. `goal(X,Y) :- p(X,Y) & ~p(Y,Y)`

Solution 3.2

1. Yes
2. Yes
3. No – variable `Y` appears in the head but not in any positive literal
4. No – variable `Z` appears in a negative literal but not in any positive literal
5. Yes

Exercise 3.3

What is the result of evaluating the query `goal(X,Z) :- p(X,Y) & p(Y,Z)` on the dataset shown below.

```
p(a,b)
p(a,c)
p(b,d)
p(c,d)
```

Solution 3.3

The result is `goal(a,d)`.

Exercise 3.4

Assume we have a dataset with a binary predicate `parent` (which is true of two people if and only if the person specified as the first argument is the parent of the person specified as the second argument).

- Write a query that defines the property of being childless.

→Hint: use the aggregate operator countofall. And be sure your query is safe. (This exercise is not difficult, but it is slightly tricky.)

Solution 3.4

```
goal(X) :- evaluate(countofall(Y,parent(X,Y)),0)
```

Warning

Homework 3.1.1 and 3.1.2 should be submitted on Blackboard before the next course session!

Homework 3.1.1

Given the following Dataset: {p(a,b), p(b,c), p(c,d), p(d,c)}

Determine the query-instances and result for each of the following queries:

1. `goal(X) :- p(X,Y) & p(Y,X)`
2. `goal(X,X) :- p(X,Y) & p(Y,X)`
3. `goal(X,b) :- p(X,Y) & p(Y,X)`
4. `goal(X,f(X)) :- p(X,Y) & p(Y,X)`

Homework 3.1.2

Given the following Dataset: {`parent(art,bob)`, `parent(art,bea)`, `parent(art,ben)`, `parent(bob,eli)`}

Write the query for each of the following requests:

1. Find every person with at least one child.
2. Find every person with at least two children.
3. Find every person with at least three children.
4. Find every person with *exactly* three children.

Outline

1. Preview

2. Queries

3. Updates
3.1 Update Syntax
3.2 Update Semantics
3.3 Simultaneous Updates
3.4 Examples
3.5 Key Takeaways
3.6 Exercises
3.7 Homework

Introduction

- In the preceding section, we saw **how to write queries to extract information from a dataset**.
- In this section, we look at **how to update the information in a dataset**, i.e., how to transform one dataset into another, ideally without rewriting all of the factoids and instead concentrating on only those factoids that have changed their truth values.

Update Syntax (1/3)

- As with the **query language**, the **update language includes the language of datasets** but **provides some additional features**. Again, **we have variables**, but in this case **we have update rules in place of query rules**.
- An *update rule* is an **expression** consisting of a **first non-empty collection of literals** (called *conditions*) and a **second non-empty collection of literals** (called *conclusions*).
  - The **conditions** and **conclusions may be ground or they may contain variables**.
    → **Only one restriction: all variables in conclusions must also appear in the positive conditions.**

Update Syntax (2/3)

For example: the below expression is an **update rules** where `p(a,b)` and `~q(b)` are **conditions**, and `~p(a,b)` and `p(b,a)` are **conclusions**.

```
p(a,b) & ~q(b) ==> ~p(a,b) & p(b,a)
```

- An **update rule** is something like a **condition-action rule**. It is a **statement** that whenever **the conditions are true**, then **the negative conclusions should be deleted from the dataset, and the positive conclusions should be added.**

For example: the rule above states that, if `p(a,b)` is true and `q(b)` is not true, then `p(a,b)` should be removed from the dataset and `p(b,a)` should be added.

Update Syntax (3/3)

- As with **query rules**, the power of **update rules** is **greatly enhanced through the use of variables**.

  For example: the rule shown below is a more general version of the rule shown in the previous slide. **Instead of applying to just the specific objects** `a` **and** `b` **it applies to all objects**.

```
p(X,Y) & ~q(Y) ==> ~p(X,Y) & p(Y,X)
```

- An **update** is a finite collection of **update rules**.
  - Typically, an **update** consists of **just one rule**. However, **updates** with **multiple rules** are sometimes useful and do not add any major complexity, so in what follows **we allow** for the possibility of **updates with multiple rules**.

Update Semantics (1/4)

- An *instance* of an **update rule** is one in which **all variables** have been **consistently replaced by ground terms** (i.e., terms without variables).

For example: if we have a language with symbols `a` and `b`, then the **instances** of `p(X,Y) & ~q(Y) ==> ~p(X,Y) & p(Y,X)` are shown below.

```
p(a,a) & ~q(a) ==> ~p(a,a) & p(a,a)
p(a,b) & ~q(b) ==> ~p(a,b) & p(b,a)
p(b,a) & ~q(a) ==> ~p(b,a) & p(a,b)
p(b,b) & ~q(b) ==> ~p(b,b) & p(b,b)
```

Update Semantics (2/4)

- Suppose we are given a **dataset** Δ and an **update rule** r. We say that an **instance** of r is **active** on Δ if and only if the **conditions** are all true on Δ.
  We define:
  - $A(r, \Delta)$ **(positive update set)** to be the set of all **positive conclusions** in some **active instance** of r.
  - $D(r, \Delta)$ **(negative update set)** to be the set of all **negative conclusions** in some **active instance** of r.
  - $A(\Omega, \Delta)$ **(positive update set)** for a set of rules Ω on a dataset Δ is the union of the **positive updates of the rules** on Δ.
  - $D(\Omega, \Delta)$ **(negative update set)** for a set of rules Ω on a dataset Δ is the union of the **negative updates of the rules** on Δ.
- To obtain the result of applying a **set of update rules** R to a dataset Δ by **removing the negative updates** and **adding in the positive updates**, i.e., the result is $\Delta - D(\Omega, \Delta) \cup A(\Omega, \Delta)$.

Update Semantics (3/4)

Examples (1/2)

Consider the **dataset** shown below. In this case, there are four **symbols** {a, b, c, d} and one **binary predicate** p.

{p(a,a), p(a,b), p(b,c), p(c,c), p(c,d)}

Suppose we want to **drop all** of the p factoids in our dataset in which **the first and second arguments are the same**. To do this, we would specify the next **update**:
p(X,X) ==> ∼p(X,X)
In this case, there would be the next two **active instances** in which the next conditions are true: p(a,a) ==> ∼p(a,a) and p(c,c) ==> ∼p(c,c)
Consequently, after execution of this update, we would have the dataset shown below.

{p(a,b), p(b,c), p(c,d)}

Update Semantics (4/4)

Examples (2/2)

Now suppose that we want to **reverse the arguments of the remaining** `p` factoids in our dataset.

To do this, we would specify an update with `p(X,Y)` as a **condition**; we would include `p(X,Y)` as a **negative conclusion**; and we would specify `p(Y,X)` as a **positive conclusion**.

```
p(X,Y) ==> ~p(X,Y) & p(Y,X)
```

In this case, we would have one variable assignment for every factoid in our dataset; the **negative conclusions** would be {`p(a,b)`, `p(b,c)`, `p(c,d)`}, i.e., every factoid in our dataset; and the **positive conclusions** would be {`p(b,a)`, `p(c,b)`, `p(d,c)`}. **After executing this update on the preceding dataset, we would have the dataset shown below.**

{`p(b,a)`, `p(c,b)`, `p(d,c)`}

Simultaneous Updates

→Note that it sometimes happens that a factoid appears as both a positive and a negative update.

As an example of this, consider an update with `p(X,a)` **as condition**, with `p(X,a)` **as a negative conclusion** and with `p(a,X)` **as a positive conclusion**.

```
p(X,a) ==> ~p(X,a) & p(a,X)
```

In the case of the first dataset shown in the preceding section, `p(a,a)` **would appear as both a positive and a negative update**.

→In such cases, our semantics dictates that the factoid be removed and then added right back in again, with the result that there is no change.

Example 1: Kinship (1/2)

Once again consider the Kinship application; and assume, as before, that we start with a **single binary predicate** `parent` and the following dataset:

```
{parent(art,bob), parent(art,bea), parent(bob,cal), parent(bob,cam),
                  parent(bea,cat), parent(bea,coe)}
```

- *Suppose, for example,* we want to **store information about grandparents and their grandchildren**. We could do this by writing an update like the one shown below.

```
parent(X,Y) & parent(Y,Z) ==> grandparent(X,Z)
```

Applying this update would result in the addition of the following factoids to our dataset.

```
{grandparent(art,cal), grandparent(art,cam), grandparent(art,cat),
                       grandparent(art,coe)}
```

Example 1: Kinship (2/2)

- If we subsequently want to **remove these factoids**, we could execute the update shown below, and we would end up back where we started.

```
grandparent(X,Y) ==> ~grandparent(X,Z)
```

- Now, suppose we want to **reverse the arguments to the parent predicate, relating children and parents rather than relating parents and children**. To do this, we could write the following update.

```
parent(X,Y) ==> ~parent(X,Y) & parent(Y,X)
```

Executing this update would result in the following dataset.

```
{parent(bob,art), parent(bea,art), parent(cal,bob), parent(cam,bob),
parent(cat,bea), parent(coe,bea)}
```

→*In understanding updates like this one, it is important to keep in mind that updates happen atomically—we first compute all factoids to be changed and we then make those changes all at once—before considering any other updates.*

Example 2: Colors (1/9)

Ruby Red, Willa White, and Betty Blue meet for lunch.

- One is wearing a red skirt; one is wearing a white skirt; and one is wearing a blue skirt.
- No one is wearing more than one color, and no two are wearing the same color.
- Betty Blue tells one of her companions, **"Did you notice we are all wearing skirts with different colors from our names?"**; and the other woman, who is wearing a white skirt, says, **"Wow, that's right!"**

Our job is to figure out which woman is wearing which skirt.

Example 2: Colors (2/9)

- One way to solve problems like this is to **enumerate possibilities** and **check**, for each, **whether it satisfies the constraints in the statement of the problem**.
  - This approach works, but it often requires a good deal of search.
- To **make the process of finding solutions more efficient**, we can sometimes **use values we already know to infer additional values** and thereby **cut down on the number of possibilities we need to consider**.

In what follow, we see how we can use updates to implement this technique.
→In this very special case, as we shall see, this technique eliminates search altogether.

Example 2: Colors (3/9)

- To solve the problem, we adopt a vocabulary with six symbols `r`, `w`, `b`, `v`, `x`, and `e`.
  - The first three denote people/colors and the latter three denote "truth values"—true, false, and unknown.
- To express the state of our problem, we use a ternary relation constant `c`. *For example:*, we would write `c(r,w,v)` to mean that **Ruby Red is wearing a white skirt**; we would write `c(r,w,x)` to mean that **Ruby Red is not wearing a white skirt**; and we would write `c(r,w,e)` to mean that **we do not know whether or not Ruby Red wearing a white skirt**.

In solving this problem we start with the dataset shown below. →Initially, we know nothing about who is wearing what.

```
{c(r,r,e), c(r,w,e), c(r,b,e),
 c(w,r,e), c(w,w,e), c(w,b,e),
 c(b,r,e), c(b,w,e), c(b,b,e)}
```

Example 2: Colors (4/9)

We can picture this situation as shown by the below table, with the idea that the value in each cell is an indication of our belief about whether the person listed as the first argument in one of our c **factoids** is wearing a skirt with the color specified as the second argument. **For the sake of clarity, we leave cells empty when they have value** e.

	r	w	b
r			
w			
b			

Example 2: Colors (5/9)

- First we apply the **constraint** that **none of the women is wearing a skirt with the same color as her name**.

```
c(C,C,e) ==> ~c(C,C,e) & c(C,C,x)
```

- After this update, we are left with the state of affairs shown below. We now have `x` values along the diagonal, but we still have empty cells everywhere else.

	r	w	b
r	×		
w		×	
b			×

Example 2: Colors (6/9)

- Next we take into account Betty Blue's comment to someone who is wearing a white skirt, which means that Betty Blue is not wearing a white skirt.

```
c(b,w,e) ==> ~c(b,w,e) & c(b,w,x)
```

- This leaves us with the situation shown below.

	r	w	b
r	×		
w		×	
b		×	×

Example 2: Colors (7/9)

Now we get to the interesting part.

- First, we have updates that tell us that, if there are two occurrences of x in a row or a column and the remaining cell is an e, then the final possibility in that row or column must be a v.

```
c(P,C1,x) & c(P,C2,x) & c(P,C3,e) & mutex(C1,C2,C3) ==> ∼c(P,C3,e) &
  c(P,C3,v) c(P1,C,x) & c(P2,C,x) & c(P3,C,e) & mutex(P1,P2,P3) ==>
                          ∼c(P3,C,e) & c(P3,C,v)
```

- This leaves us with the situation shown below.

	r	w	b
r	×	✓	
w		×	
b	✓	×	×

Example 2: Colors (8/9)

- Similarly, we have updates that tell us that, if there is an occurrence of an `v` in a row or a column and there is a cell containing an `e`, then that `e` should be changed to an `x`.

```
c(P,C1,v) & c(P,C2,e) ==> ∼c(P,C2,e) & c(P,C2,x) c(P1,C,v) &
                c(P2,C,e) ==> ∼c(P2,C,e) & c(P2,C,x)
```

- Applying these updates gives us more information. Since Ruby Red is wearing a white skirt, she must not be wearing a blue skirt. Since the red skirt is wearied by Betty Blue, it can't be wearied by other girls.

	r	w	b
r	×	✓	×
w	×	×	
b	✓	×	×

Example 2: Colors (9/9)

Applying the last updates more times leads to an overall solution to the problem.

	r	w	b
r	×	✓	×
w	×	×	✓
b	✓	×	×

- **This problem is special in that we can solve it only by inferring values from other values.** In **constraint satisfaction problems** like this one, some search is often necessary.
- This case is also special in that it is **easy to express all of the update rules necessary to solve the problem**. For some problems, such as solving **Sudoku puzzles**, it is **impractical to write update rules using our limited update language.**

Key Takeaways

Key Takeaways

- The **update language** extends the **dataset language** by adding **update rules**, which consist of **conditions** and **conclusions**.
- An **update rule** specifies that whenever the **conditions** are true, the **negative conclusions** should be deleted from the dataset and the **positive conclusions** should be added.
- The **semantics of updates** is based on the idea of **active instances** of **update rules**.
- **Simultaneous updates** may result in a factoid being both added and deleted; in such cases, the factoid is simply removed and then added back in again.
- **Updates** can be used to implement **constraint propagation** techniques for solving constraint satisfaction problems.

Exercise 3.5

For each of the following strings, say whether it is a syntactically legal update.

1. `p(a,f(f(X))) ==> p(X,Y)`
2. `P(a,Y) ==> P(Y,a)`
3. `p(X,Y) & p(Y,Z) ==> ∼(X,Y) & ∼p(Y,Z) & p(X,Z)`
4. `p(X,b) ==> f(X,f(b,c))`

Solution 3.5

1. **No.** The conclusion `p(X,Y)` contains the variable `Y`, which does not appear in any positive condition.
2. **Yes.** Both the condition and the conclusion are syntactically legal, and the variable `Y` appears in a positive condition.
3. **No.** The negative conclusion $\sim$`(X,Y)` is not a syntactically legal literal.
4. **No.** The conclusion `f(X,f(b,c))` is not a syntactically legal literal.

Exercise 3.6

What is the result of applying the update rule `p(X,Y) ==> ~p(X,Y) & p(Y,X)` on the dataset shown below.

```
p(a,a)
p(a,b)
p(b,a)
```

Solution 3.6

The result is the dataset shown below.

```
p(a,a) p(b,a) p(a,b)
```

Note that `p(a,a)` appears as both a positive and a negative update, so it is removed and then added back in again.

Exercise 3.7

Suppose we have a kinship dataset with a binary predicate `parent` and a unary predicate `male`.

- Write **update rules** to replace all factoids using the `parent` predicate with equivalent factoids using the binary predicates `father` and `mother`.

Solution 3.7

The following two update rules accomplish this task.

```
parent(X,Y) & male(X) ==> ~parent(X,Y) & father(X,Y)
parent(X,Y) & ~male(X) ==> ~parent(X,Y) & mother(X,Y)
```

Warning

Homework 3.2.1 should be submitted on Blackboard before the next course session!

Homework 3.2.1

Amy, Bob, Coe, and Dan are traveling to different places. **One goes by train, one by car, one by plane, and one by ship**.
Amy hates flying, Bob rented his vehicle, Coe tends to get seasick, and Dan loves trains.

- Write **update rules** to solve this problem by **constraint propagation**.

End of Chapter 3