Aim and Learning Outcomes

The aim of this lecture is to:

Introduce you to the main concepts around *quering a knowledge base* and how these are concretely realised in the *Datalog* language.

Learning Outcomes

By the end of this lecture you will be able to:

1. Describe what it means to *query* a knowledge base
2. Define *matching* and compute matching substitutions
3. Apply the *forward chaining* algorithm to find consequences of a knowledge base
4. Explain certain optimisations of the algorithm
Recap

In General

Abstraction

Reality
In General

Abstraction

Domain
Recap

In General

Abstraction

Domain

Database Semantics

Closed World Domain Closure

Unique Names

Datalog

Has Database Semantics

Fact: concrete relationship between objects
e.g. loves(giles, cheese)

Rule: loves(X, Y), has(X, Y) ⇒ happy(X)
Recap

In General

Abstraction

Interpretation

Domain

Database Semantics

Closed World Domain Closure

Unique Names

Datalog

Knowledge Base KB: set of facts and rules
Fact \( f \) is a consequence of KB if all interpretations satisfying KB satisfy \( f \) written \( KB|_f \)

Example:

- Fact: concrete relationship between objects e.g. loves(giles, cheese)
- Rule: loves(X, Y), has(X, Y) \( \Rightarrow \) happy(X)

Giles Reger
Lecture 3
February 2019
In General

Abstraction

Interpretation

Domain

Database Semantics

- Closed World
- Domain Closure
- Unique Names

Datalog has database semantics. Fact: concrete relationship between objects, e.g. loves(giles, cheese). Rule: loves(X, Y), has(X, Y) ⇒ happy(X).
Recap

In General

Abstraction

Interpretation

Domain

Database Semantics

- Closed World
- Domain Closure
- Unique Names

Datalog

Has Database Semantics

Fact: concrete relationship between objects
e.g. loves(giles, cheese)

Rule: loves(X, Y), has(X, Y) ⇒ happy(X)

Knowledge Base $\mathcal{KB}$: set of facts and rules

Fact $f$ is a consequence of $\mathcal{KB}$
If all interpretations satisfying $\mathcal{KB}$ satisfy $f$
written $\mathcal{KB} \models f$
Properties of Datalog

Given a knowledge base there are a finite number of consequences

Why?
Properties of Datalog

Given a knowledge base there are a finite number of consequences

Why? Each rule has a finite number of instances (finite new facts)
Properties of Datalog

Given a knowledge base there are a finite number of consequences.

**Why?** Each rule has a finite number of instances (finite new facts).

Checking if \( f \) is a consequence of \( KB \) is **decidable**.

- An interpretation can be defined by the facts true in it.
- Due to database semantics, \( KB \) has a single minimal interpretation \( M \) satisfying it. If a fact is satisfied by this it is a consequence of \( KB \).
- The set of all facts built from \( O \) and \( R \) is finite, call this \( A \).
- Clearly \( M \subseteq A \); we can search all subsets of \( A \).
Properties of Datalog

Given a knowledge base there are a **finite** number of consequences.

**Why?** Each rule has a finite number of instances (finite new facts).

Checking if \( f \) is a consequence of \( KB \) is **decidable**.

- An interpretation can be defined by the facts true in it.
- Due to database semantics, \( KB \) has a single minimal interpretation \( M \) satisfying it. If a fact is satisfied by this it is a consequence of \( KB \).
- The set of all facts built from \( O \) and \( R \) is finite, call this \( A \).
- Clearly \( M \subseteq A \); we can search all subsets of \( A \).

Finally, can a Datalog knowledge base be **inconsistent**?
Properties of Datalog

Given a knowledge base there are a finite number of consequences

Why? Each rule has a finite number of instances (finite new facts)

Checking if \( f \) is a consequence of \( \mathcal{KB} \) is decidable.

- An interpretation can be defined by the facts true in it
- Due to database semantics, \( \mathcal{KB} \) has a single minimal interpretation \( \mathcal{M} \) satisfying it. If a fact is satisfied by this it is a consequence of \( \mathcal{KB} \)
- The set of all facts built from \( \mathcal{O} \) and \( \mathcal{R} \) is finite, call this \( \mathcal{A} \)
- Clearly \( \mathcal{M} \subseteq \mathcal{A} \); we can search all subsets of \( \mathcal{A} \)

Finally, can a Datalog knowledge base be inconsistent? No, the set of all consequences always exists and defines a satisfying interpretation.
Queries

Given a knowledge base we want to ask queries

These can be ground e.g. is ancestor(giles, adam) true?

Or, more interestingly, they can contain variables e.g. give me all ancestors of giles or more formally all $X$ such that ancestor(giles, $X$) is true.

A query is a fact, possibly containing variables.
The answer to a query $q$ of a knowledge base $KB$ is the set

$$ans(q) = \{ \sigma \mid KB \models \sigma(q) \}$$

e.g. the set of all substitutions, which when applied to $q$ produce a ground fact that is a consequence of $KB$. 
The answer to a query $q$ of a knowledge base $KB$ is the set

$$\text{ans}(q) = \{\sigma \mid KB \models \sigma(q)\}$$

e.g. the set of all substitutions, which when applied to $q$ produce a ground fact that is a consequence of $KB$.

If the query has no answers then $\text{ans}$ is empty. Can this happen?
The answer to a query $q$ of a knowledge base $\mathcal{KB}$ is the set

$$\text{ans}(q) = \{\sigma \mid \mathcal{KB} \models \sigma(q)\}$$

e.g. the set of all substitutions, which when applied to $q$ produce a ground fact that is a consequence of $\mathcal{KB}$.

If the query has no answers then $\text{ans}$ is empty. Can this happen?

What will happen if $q$ is ground?
The answer to a query $q$ of a knowledge base $KB$ is the set

$$\text{ans}(q) = \{ \sigma \mid KB \models \sigma(q) \}$$

e.g. the set of all substitutions, which when applied to $q$ produce a ground fact that is a consequence of $KB$.

If the query has no answers then $\text{ans}$ is empty. Can this happen?

What will happen if $q$ is ground? The substitution will be empty
The **answer** to a query $q$ of a knowledge base $\mathcal{KB}$ is the set

$$\text{ans}(q) = \{ \sigma \mid \mathcal{KB} \models \sigma(q) \}$$

e.g. the set of all substitutions, which when applied to $q$ produce a ground fact that is a consequence of $\mathcal{KB}$.

If the query has no answers then $\text{ans}$ is empty. Can this happen?

What will happen if $q$ is ground? The substitution will be empty.

Will $\text{ans}(q)$ always be finite?
The answer to a query \( q \) of a knowledge base \( \mathcal{KB} \) is the set

\[
\text{ans}(q) = \{ \sigma \mid \mathcal{KB} \models \sigma(q) \}
\]

e.g. the set of all substitutions, which when applied to \( q \) produce a ground fact that is a consequence of \( \mathcal{KB} \).

If the query has no answers then \( \text{ans} \) is empty. Can this happen?

What will happen if \( q \) is ground? The substitution will be empty

Will \( \text{ans}(q) \) always be finite? Yes - there are finite consequences
Computing the Set of Consequences

Given our initial set of facts $F_0$ we want to add \textit{new} consequences until we reach a \textit{fixed-point}

Let our knowledge base $KB$ consist of facts $F_0$ and rules $RU$

Define the \textit{next} set of facts as follows

$$F_i = F_{i-1} \cup \left\{ \sigma(\text{head}) \mid \begin{array}{l}
\text{body} \Rightarrow \text{head} \in RU \\
\sigma(\text{body}) \in F_{i-1}
\end{array} \right\}$$

This reaches a fixed point when $F_j = F_{j+1}$

As there are finite consequences this will terminate
Computing the Set of Consequences

Given our initial set of facts $F_0$ we want to add new consequences until we reach a fixed-point.

Let our knowledge base $KB$ consist of facts $F_0$ and rules $RU$.

Define the next set of facts as follows:

$$F_i = F_{i-1} \cup \left\{ \sigma(head) \mid \begin{array}{l}
\text{body} \Rightarrow \text{head} \in RU \\
\sigma(body) \in F_{i-1}
\end{array} \right\}$$

This reaches a fixed point when $F_j = F_{j+1}$.

As there are finite consequences this will terminate.

How do we find $\sigma$? How do we compute $F_{i+1}$ efficiently?
Matching

A fact is ground if it does not contain variables.

Given a fact $f_1$ and a ground fact $f_2$ we say $f_2$ matches $f_1$ if there exists a substitution $\sigma$ such that $f_2 = \sigma(f_1)$.

Examples:

<table>
<thead>
<tr>
<th>Ground fact $f_2$ matches</th>
<th>fact $f_1$ using</th>
<th>substitution $\sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>happy(giles)</td>
<td>happy($X$)</td>
<td>${X \mapsto \text{giles}}$</td>
</tr>
<tr>
<td>loves(giles, cheese)</td>
<td>loves($X$, cheese)</td>
<td>${X \mapsto \text{giles}}$</td>
</tr>
<tr>
<td>loves(giles, cheese)</td>
<td>loves($X$, $Y$)</td>
<td>${X \mapsto \text{giles}, Y \mapsto \text{cheese}}$</td>
</tr>
<tr>
<td>happy(giles)</td>
<td>happy(giles)</td>
<td>${}$</td>
</tr>
</tbody>
</table>

Note that loves(giles, cheese) does not match with loves($X$, $X$).
Computing Matching Substitutions

Match two facts given an existing substitution

```python
def match(f1 = name1(args1), f2 = name2(args2), σ):
    if name1 and name2 are different then return ⊥;
    for i ← 0 to length(args1) do
        if args1[i] is a variable and args1[i] ∉ σ then
            σ = σ ∪ {args1[i] ↦ args2[i]}
        else if σ(args1[i]) ≠ args2[i] then
            return ⊥
    return σ
```

If names are different, no match. For each parameter of \( f_1 \), if it is an unseen variable then extend \( σ \), otherwise check that things are consistent.

Matching is an instance of unification, which we will meet later. In unification both sides can contain variables.
def match(f_1 = name_1(args_1), f_2 = name_2(args_2), σ):
    if name_1 and name_2 are different then return ⊥;
    for i ← 0 to length(args_1) do
        if args_1[i] is a variable and args_1[i] ∉ σ then
            σ = σ ∪ {args_1[i] ↦ args_2[i]}
        else if σ(args_1[i]) ≠ args_2[i] then
            return ⊥
    end
return σ

match(parent(X, Y), parent(giles, mark), {X ↦ giles})
Computing Matching Substitutions

def match(f_1 = name_1(args_1), f_2 = name_2(args_2), σ):
    if name_1 and name_2 are different then return ⊥;
    for i ← 0 to length(args_1) do
        if args_1[i] is a variable and args_1[i] ∉ σ then
            σ = σ ∪ \{args_1[i] ↦ args_2[i]\}
        else if σ(args_1[i]) ≠ args_2[i]) then
            return ⊥
    end
    return σ

match(parent(X, Y), parent(giles, mark), \{X ↦ giles\})

- f_1 = parent(X, Y)
- f_2 = parent(giles, mark)
- σ = \{X ↦ giles\}
Computing Matching Substitutions

```python
def match(f_1 = name_1(args_1), f_2 = name_2(args_2), σ):
    if name_1 and name_2 are different then return ⊥;
    for i ← 0 to length(args_1) do
        if args_1[i] is a variable and args_1[i] ∉ σ then
            σ = σ ∪ {args_1[i] ↦ args_2[i]}
        else if σ(args_1[i]) ≠ args_2[i]) then
            return ⊥
        end
    end
    return σ
```

match(parent(X, Y), parent(giles, mark), {X ↦ giles})

- f_1 = parent(X, Y)
- f_2 = parent(giles, mark)
- σ = {X ↦ giles}
def match($f_1 = name_1(\text{args}_1), f_2 = name_2(\text{args}_2), \sigma$):
    if $name_1$ and $name_2$ are different then return $\bot$;
    for $i \leftarrow 0$ to $\text{length}(\text{args}_1)$ do
        if $\text{args}_1[i]$ is a variable and $\text{args}_1[i] \notin \sigma$ then
            $\sigma = \sigma \cup \{(\text{args}_1[i] \mapsto \text{args}_2[i])\}$
        else if $\sigma(\text{args}_1[i]) \neq \text{args}_2[i]$ then
            return $\bot$
    end
    return $\sigma$

match(parent($X, Y$), parent(giles, mark), $\{X \mapsto \text{giles}\}$)

- $f_1 = \text{parent}(X, Y)$
- $f_2 = \text{parent}(\text{giles}, \text{mark})$
- $\sigma = \{X \mapsto \text{giles}\}$
- $\text{args}_1[0] = X$
- $\text{args}_2[0] = \text{giles}$
Computing Matching Substitutions

```python
def match(f_1 = name_1(args_1), f_2 = name_2(args_2), σ):
    if name_1 and name_2 are different then return ⊥;
    for i ← 0 to length(args_1) do
        if args_1[i] is a variable and args_1[i] ∉ σ then
            σ = σ ∪ {args_1[i] ↦ args_2[i]}
        else if σ(args_1[i]) ≠ args_2[i] then
            return ⊥
    end
    return σ
```

match(parent(X, Y), parent(giles, mark), {X ↦ giles})

- f_1 = parent(X, Y)
- f_2 = parent(giles, mark)
- σ = {X ↦ giles}
- args_1[0] = X
- args_2[0] = giles
- σ(args_1[0]) = {X ↦ giles}(X) = giles
Computing Matching Substitutions

def match(f_1 = name_1(args_1), f_2 = name_2(args_2), σ):
    if name_1 and name_2 are different then return ⊥;
    for i ← 0 to length(args_1) do
        if args_1[i] is a variable and args_1[i] ∉ σ then
            σ = σ ∪ {args_1[i] ↦ args_2[i]}
        else if σ(args_1[i]) ≠ args_2[i] then
            return ⊥
    end
    return σ

match(parent(X, Y), parent(giles, mark), {X ↦ giles})

- f_1 = parent(X, Y)
- f_2 = parent(giles, mark)
- σ = {X ↦ giles}
- args_1[1] = Y
- args_2[1] = mark
Computing Matching Substitutions

def match(f1 = name1(args1), f2 = name2(args2), σ):
    if name1 and name2 are different then return ⊥;
    for i ← 0 to length(args1) do
        if args1[i] is a variable and args1[i] ∉ σ then
            σ = σ ∪ {args1[i] ↦ args2[i]}
        else if σ(args1[i]) ≠ args2[i] then
            return ⊥
    end
    return σ

match(parent(X, Y), parent(giles, mark), {X ↦ giles})

- f1 = parent(X, Y)
- f2 = parent(giles, mark)
- σ = {X ↦ giles, Y ↦ mark}
- args1[1] = Y
- args2[1] = mark
Computing Matching Substitutions

```python
def match(f1 = name1(args1), f2 = name2(args2), σ):
    if name1 and name2 are different then return ⊥;
    for i ← 0 to length(args1) do
        if args1[i] is a variable and args1[i] ∉ σ then
            σ = σ ∪ {args1[i] ↦→ args2[i]}
        else if σ(args1[i]) ≠ args2[i] then
            return ⊥
    end
    return σ

match(parent(X, Y), parent(giles, mark), {X ↦→ giles})
```

- $f_1 = \text{parent}(X, Y)$
- $f_2 = \text{parent}(\text{giles}, \text{mark})$
- $σ = \{X ↦→ \text{giles}, Y ↦→ \text{mark}\}$
Computing Matching Substitutions

```python
def match(f1 = name1(args1), f2 = name2(args2), σ):
    if name1 and name2 are different then return ⊥;
    for i ← 0 to length(args1) do
        if args1[i] is a variable and args1[i] ∉ σ then
            σ = σ ∪ {args1[i] ↦ args2[i]}
        else if σ(args1[i]) ≠ args2[i]) then
            return ⊥
    end
    return σ
```

match(parent(X, Y), parent(giles, mark), {X ↦ bob})

- f1 = parent(X, Y)
- f2 = parent(giles, mark)
- σ = {X ↦ bob}
def match(f₁ = name₁(args₁), f₂ = name₂(args₂), σ):
    if name₁ and name₂ are different then return ⊥;
    for i ← 0 to length(args₁) do
        if args₁[i] is a variable and args₁[i] ∉ σ then
            σ = σ ∪ {args₁[i] ↦ args₂[i]}
        else if σ(args₁[i]) ≠ args₂[i]) then
            return ⊥
    end
    return σ

match(parent(X, Y), parent(giles, mark), {X ↦ bob})

- f₁ = parent(X, Y)
- f₂ = parent(giles, mark)
- σ = {X ↦ bob}
def match(f₁ = name₁(args₁), f₂ = name₂(args₂), σ):
    if name₁ and name₂ are different then return ⊥;
    for i ← 0 to length(args₁) do
        if args₁[i] is a variable and args₁[i] ∉ σ then
            σ = σ ∪ {args₁[i] ↦ args₂[i]}
        else if σ(args₁[i]) ≠ args₂[i] then
            return ⊥
    end
    return σ

match(parent(X, Y), parent(giles, mark), {X ↦ bob})
def match($f_1 = name_1(args_1), f_2 = name_2(args_2), \sigma)$:
    if $name_1$ and $name_2$ are different then return $\perp$;
    for $i \leftarrow 0$ to length($args_1$) do
        if $args_1[i]$ is a variable and $args_1[i] \notin \sigma$ then
            $\sigma = \sigma \cup \{args_1[i] \mapsto args_2[i]\}$
        else if $\sigma(args_1[i]) \neq args_2[i]$ then
            return $\perp$
        end
    end
    return $\sigma$

match(parent($X, Y$), parent(giles, mark), \{$X \mapsto bob\$})

- $f_1 = parent(X, Y)$
- $f_2 = parent(giles, mark)$
- $\sigma = \{X \mapsto bob\}$
- $args_1[0] = X$
- $args_2[0] = giles$
- $\sigma(args_1[0]) = \{X \mapsto bob\}(X) = bob$
def match(f_1 = name_1(args_1), f_2 = name_2(args_2), σ):
    if name_1 and name_2 are different then return ⊥;
    for i ← 0 to length(args_1) do
        if args_1[i] is a variable and args_1[i] ∉ σ then
            σ = σ ∪ {args_1[i] ↦ args_2[i]}
        else if σ(args_1[i]) ≠ args_2[i] then
            return ⊥
    end
    return σ

match(parent(X, Y), parent(giles, mark), {X ↦ bob}) = ⊥
Matching A Rule Body

We lift the matching algorithm to match a list of facts (the rule body) against a set of ground facts (the known consequences).

```python
def match(body, F):
    matches = {∅}
    for f₁ ∈ body do
        new = ∅
        for σ₁ ∈ matches do
            for f₂ ∈ F do
                σ₂ = match(f₁, f₂, σ₁)
                if σ₂ ≠ ⊥ then new.add(σ₂);
            end
        end
        matches = new
    end
    return matches
```
Matching A Rule Body

\[
\text{match}(\text{parent}(X, Y), \text{man}(X), \{ \begin{array}{l}
\text{parent}(\text{giles, mark}), \text{man}(\text{giles}) \\
\text{parent}(\text{bob, sara}), \text{man}(\text{bob})
\end{array} )}
\]

def match(body, F):
    matches = {\emptyset}
    for \( f_1 \in \text{body} \) do
        new = \emptyset
        for \( \sigma_1 \in \text{matches} \) do
            for \( f_2 \in F \) do
                \( \sigma_2 = \text{match}(f_1, f_2, \sigma_1) \)
                if \( \sigma_2 \neq \bot \) then new.add(\( \sigma_2 \));
            end
        end
        matches = new
    end
    return matches
Matching A Rule Body

match(parent(X, Y), man(X), \{ parent(giles, mark), man(giles) \\
parent(bob, sara), man(bob) \})

def match(body, \mathcal{F}):
    matches = \{\emptyset\}
    for f_1 \in body do
        new = \emptyset
        for \sigma_1 \in matches do
            for f_2 \in \mathcal{F} do
                \sigma_2 = match(f_1, f_2, \sigma_1)
                if \sigma_2 \neq \bot then new.add(\sigma_2);
            end
        end
        matches = new
    end
    return matches
Matching A Rule Body

\[
\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l}
\text{parent(giles, mark), man(giles)} \\
\text{parent(bob, sara), man(bob)}
\end{array}\right\})
\]

```python
def match(body, F):
    matches = {\{\}\}
    for \( f_1 \in \text{body} \) do
        new = \{\\}
        for \( \sigma_1 \in \text{matches} \) do
            for \( f_2 \in F \) do
                \( \sigma_2 = \text{match}(f_1, f_2, \sigma_1) \)
                if \( \sigma_2 \neq \bot \) then new.add(\( \sigma_2 \));
            end
        end
        matches = new
    end
    return matches
```
Matching A Rule Body

match(parent(X, Y), man(X), \{ parent(giles, mark), man(giles) \\ parent(bob, sara), man(bob) \})

def match(body, \mathcal{F}):  
matches = \{\emptyset\}  
for f_1 \in body do  
  new = \emptyset  
  for \sigma_1 \in matches do  
    for f_2 \in \mathcal{F} do  
      \sigma_2 = match(f_1, f_2, \sigma_1)  
      if \sigma_2 \neq \bot then new.add(\sigma_2);  
    end  
  end  
  matches = new  
end  
return matches
Matching A Rule Body

\[
\text{match(parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l}
\text{parent}(\text{giles, mark}), \text{man}(\text{giles}) \\
\text{parent}(\text{bob, sara}), \text{man}(\text{bob})
\end{array} \right\})
\]

def match(body, F):
    matches = \{\emptyset\}
    for \ f_1 \in body \ do
        new = \emptyset
        for \ \sigma_1 \in matches \ do
            for \ f_2 \in F \ do
                \sigma_2 = \text{match}(f_1, f_2, \sigma_1)
            \end
        \end
        if \ \sigma_2 \neq \bot \ then \ new.add(\sigma_2);
    \end
    matches = new
\end

return matches
Matching A Rule Body

match(parent(X, Y), man(X), \{ parent(giles, mark), man(giles),
    parent(bob, sara), man(bob) \})

def match(body, \mathcal{F}):
    matches = \{\emptyset\}
    for f_1 \in body do
        new = \emptyset
        for \sigma_1 \in matches do
            for f_2 \in \mathcal{F} do
                \sigma_2 = match(f_1, f_2, \sigma_1)
                if \sigma_2 \neq \bot then new.add(\sigma_2);
            end
        end
        \sigma_1 = \emptyset
        matches = \sigma_1
    end
    return matches

Matching A Rule Body

match(parent(X, Y), man(X), \{ parent(giles, mark), man(giles) \\ parent(bob, sara), man(bob) \})

```python
def match(body, \mathcal{F}):
    matches = \{\emptyset\}
    for f_1 \in body do
        new = \emptyset
        for \sigma_1 \in matches do
            for f_2 \in \mathcal{F} do
                \sigma_2 = match(f_1, f_2, \sigma_1)
                if \sigma_2 \neq \bot then new.add(\sigma_2); f_2 = parent(giles, mark)
            end
            \sigma_2 = \{X \mapsto giles, Y \mapsto mark\}
        end
        matches = new
    end
    return matches
```

Matching A Rule Body

match(parent(X, Y), man(X), {parent(giles, mark), man(giles)
parent(bob, sara), man(bob)})

```python
def match(body, F):
    matches = {∅}
    for f1 ∈ body do
        new = ∅
        for σ1 ∈ matches do
            for f2 ∈ F do
                σ2 = match(f1, f2, σ1)
                if σ2 ≠ ⊥ then new.add(σ2); σ1 = ∅
            end
        end
        matches = new
    end
    return matches
```

matches = {∅}
f1 = parent(X, Y)
new = 

{ {X ↦ giles, Y ↦ mark} }
f2 = parent(giles, mark)
σ2 = {X ↦ giles, Y ↦ mark}
Matching A Rule Body

\[
\text{match}(\text{parent}(X, Y), \text{man}(X), \{ \begin{array}{l}
\text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \\
\text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob})
\end{array} \})
\]

def match(body, \mathcal{F}):
    matches = \{\emptyset\}
    for f_1 \in \text{body} do
        new = \emptyset
        for \sigma_1 \in \text{matches} do
            for f_2 \in \mathcal{F} do
                \sigma_2 = \text{match}(f_1, f_2, \sigma_1)
                if \sigma_2 \neq \bot then new.add(\sigma_2)
                end
            end
            \sigma_1 = \emptyset
        end
        new = \emptyset
    end
    return matches
Matching A Rule Body

match(parent(X, Y), man(X), \{
    parent(giles, mark), man(giles)
    parent(bob, sara), man(bob)
\})

def match(body, \mathcal{F}):
    matches = \{\emptyset\}
    for f_1 \in body do
        new = \emptyset
        for \sigma_1 \in matches do
            for f_2 \in \mathcal{F} do
                \sigma_2 = match(f_1, f_2, \sigma_1)
                if \sigma_2 \neq \bot then new.add(\sigma_2); \sigma_1 = \emptyset
            end
        end
        matches = new
    end
    return matches

matches = \{\emptyset\}
\begin{align*}
f_1 &= parent(X, Y) \\
new &= \{ \{ X \mapsto \text{giles}, Y \mapsto \text{mark} \} \} \\
\sigma_1 &= \emptyset \\
f_2 &= \text{man(giles)} \\
\sigma_2 &= \bot
\end{align*}
Matching A Rule Body

\[
\text{match}(\text{parent}(X, Y), \text{man}(X), \left\{ \begin{array}{l} 
\text{parent}(\text{giles, mark}), \text{man}(\text{giles}) \\
\text{parent}(\text{bob, sara}), \text{man}(\text{bob}) 
\end{array} \right\})
\]

```python
def match(body, F):
    matches = \{\emptyset\}
    for f_1 \in body do
        new = \emptyset
        for \sigma_1 \in matches do
            for f_2 \in F do
                \sigma_2 = \text{match}(f_1, f_2, \sigma_1)
                if \sigma_2 \neq \bot then new.add(\sigma_2); \sigma_1 = \emptyset
            end
        end
        matches = new
    end
    return matches
```
Matching A Rule Body

match(parent(X, Y), man(X), \{parent(giles, mark), man(giles)\})

```python
def match(body, F):
    matches = {\{\}\}
    for f_1 \in body do
        new = \{\}
        for \sigma_1 \in matches do
            new = \{\}
            for f_2 \in F do
                \sigma_2 = match(f_1, f_2, \sigma_1)
                if \sigma_2 \neq \bot then new.add(\sigma_2);
                \sigma_1 = \{\}
            end
        end
        matches = new
    end
    return matches
```

Giles Reger
Lecture 3
February 2019
Matching A Rule Body

match(parent(X, Y), man(X), \{ parent(giles, mark), man(giles) \} )

**def** match(body, \( F \)):

matches = \{\emptyset\}

for \( f_1 \in body \) do

new = \emptyset

for \( \sigma_1 \in matches \) do

for \( f_2 \in F \) do

\( \sigma_2 = \text{match}(f_1, f_2, \sigma_1) \)

if \( \sigma_2 \neq \bot \) then new.add(\( \sigma_2 \)); \( \sigma_1 = \emptyset \)

end

end

matches = new

end

return matches

matches = \{\emptyset\}

\( f_1 = \text{parent}(X, Y) \)

new = \{ \{ X \mapsto \text{giles}, Y \mapsto \text{mark} \} \}

\( f_2 = \text{parent}(\text{bob}, \text{sara}) \)

\( \sigma_2 = \{ X \mapsto \text{bob}, Y \mapsto \text{sara} \} \)
Matching A Rule Body

match(parent(X, Y), man(X), \{ parent(giles, mark), man(giles) \\
parent(bob, sara), man(bob) \})

def match(body, F):
    matches = \{\emptyset\}
    for f_1 \in body do
        new = \emptyset
        for \sigma_1 \in matches do
            for f_2 \in F do
                \sigma_2 = match(f_1, f_2, \sigma_1)
                if \sigma_2 \neq \bot then new.add(\sigma_2);
            end
        end
        matches = new
    end
    return matches

matches = \{\emptyset\}
f_1 = parent(X, Y)
new = \{ \{ X \mapsto giles, Y \mapsto mark \} \\{ X \mapsto bob, Y \mapsto sara \} \}
\sigma_1 = \emptyset
f_2 = parent(bob, sara)
\sigma_2 = \{ X \mapsto bob, Y \mapsto sara \}
Matching A Rule Body

match(parent(X, Y), man(X), \{ parent(giles, mark), man(giles) \\
parent(bob, sara), man(bob) \})

```python
def match(body, F):
    matches = {\{\}}
    for f_1 in body do
        new = \{
            \{
                X \mapsto giles, Y \mapsto mark
            },
            \{
                X \mapsto bob, Y \mapsto sara
            }
        }
        for \sigma_1 in matches do
            for f_2 in F do
                \sigma_2 = match(f_1, f_2, \sigma_1)
                if \sigma_2 \neq \bot then new.add(\sigma_2); \sigma_1 = \{
            end
        end
        matches = new
    end
    return matches
```
Matching A Rule Body

match(parent(X, Y), man(X), {parent(giles, mark), man(giles), parent(bob, sara), man(bob)})

```python
def match(body, F):
    matches = {∅}
    for f1 ∈ body do
        new = Ø
        for σ1 ∈ matches do
            for f2 ∈ F do
                σ2 = match(f1, f2, σ1)
                if σ2 ≠ ⊥ then new.add(σ2); σ1 = Ø
            end
        end
        matches = new
    end
    return matches
```

matches = {Ø}
f1 = parent(X, Y)
new =
{ {X ↦ giles, Y ↦ mark} {X ↦ bob, Y ↦ sara} }
σ1 = Ø
f2 = man(bob)
σ2 = ⊥
Matching A Rule Body

\[
\text{match}(\text{parent}(X, Y), \text{man}(X), \{ \text{parent}(\text{giles}, \text{mark}), \text{man}(\text{giles}) \}, \text{parent}(\text{bob}, \text{sara}), \text{man}(\text{bob}))
\]

```python
def match(body, F):
    matches = \{\emptyset\}
    for f1 ∈ body do
        new = \emptyset
        for σ1 ∈ matches do
            for f2 ∈ F do
                σ2 = match(f1, f2, σ1)
                if σ2 ≠ ⊥ then new.add(σ2);
            end
        end
        matches = new
    end
    return matches
```

matches = \{\emptyset\}
\[f_1 = \text{parent}(X, Y)\]
new = \{ \{X ↦ giles, Y ↦ mark\}, \{X ↦ bob, Y ↦ sara\} \}
\[σ_1 = \emptyset\]
\[f_2 = \text{man}(\text{bob})\]
\[σ_2 = ⊥\]
match(parent(X, Y), man(X), \{ parent(giles, mark), man(giles) \\
parent(bob, sara), man(bob) \})

def match(body, \mathcal{F}): 
matches = \{\emptyset\}
for f_1 \in body do
    new = \emptyset
    for \sigma_1 \in matches do
        for f_2 \in \mathcal{F} do
            \sigma_2 = \text{match}(f_1, f_2, \sigma_1)
            if \sigma_2 \neq \bot \text{ then } new.add(\sigma_2);
        end
    end
    matches = new
end
return matches
def match(body, \( \mathcal{F} \)):
    matches = \{\emptyset\}
    for \( f_1 \in \text{body} \) do
        new = \emptyset
        for \( \sigma_1 \in \text{matches} \) do
            for \( f_2 \in \mathcal{F} \) do
                \( \sigma_2 = \text{match}(f_1, f_2, \sigma_1) \)
                if \( \sigma_2 \neq \bot \) then new.add(\( \sigma_2 \));
            end
        end
        matches = new
    end
    return matches

Clearly inefficient

The order in which we check elements in the body can effect the complexity as we can get a large set of initial fact on the first item and find that most are inconsistent with the next one

In reality we do something cleverer
Forward Chaining Algorithm

Compute the next set of consequences whilst there are new consequences. Search all consequences for facts matching the query.

```python
def forward(facts F₀, rules RU, query q):
    F = ∅; new = F₀
    do
        F = F ∪ new; new = ∅
        for body ⇒ head ∈ RU do
            for σ ∈ match(body, F) do
                if σ(head) ∉ F then new.add(σ(head))
            end
        end
    while new ≠ ∅
    ans = ∅
    for f ∈ F do σ = match(q, f, ∅); if σ ≠ ⊥ then ans.add(σ)
    return ans
```
Forward Chaining Algorithm

Compute the next set of consequences whilst there are new consequences. Search all consequences for facts matching the query.

```python
def forward(facts \( F_0 \), rules \( R \), query \( q \)):
    \( F = \emptyset \); \( new = F_0 \)
    do
        \( F = F \cup new \); \( new = \emptyset \)
        for body \( \Rightarrow head \in R \) do
            for \( \sigma \in \text{match}(body, F) \) do
                if \( \sigma(head) \not\in F \) then \( new.add(\sigma(head)) \)
            end
        end
    while new \( \neq \emptyset \)
    ans = \emptyset
    for \( f \in F \) do \( \sigma = \text{match}(q, f, \emptyset) \); if \( \sigma \neq \bot \) then ans.add(\sigma) end
    return ans
```
Forward Chaining Algorithm

Compute the next set of consequences whilst there are new consequences. Search all consequences for facts matching the query.

```python
def forward(facts F₀, rules RU, query q):
    F = ∅; new = F₀
    do
        F = F ∪ new; new = ∅
        for body ⇒ head ∈ RU do
            for σ ∈ match(body, F) do
                if σ(head) /∈ F then new.add(σ(head))
            end
        end
    while new /≠ ∅
    ans = ∅
    for f ∈ F do σ = match(q, f, ∅); if σ /≠ ⊥ then ans.add(σ)
    return ans
```

Forward Chaining Algorithm

Compute the next set of consequences whilst there are new consequences. Search all consequences for facts matching the query.

```python
def forward(facts F₀, rules RU, query q):
    F = ∅; new = F₀
    do
        F = F ∪ new; new = ∅
        for body ⇒ head ∈ RU do
            for σ ∈ match(body, F) do
                if σ(head) ∉ F then new.add(σ(head))
            end
        end
    while new ≠ ∅
    ans = ∅
    for f ∈ F do σ = match(q, f, ∅); if σ ≠ ⊥ then ans.add(σ)
    return ans
```

Compute the next set of consequences whilst there are new consequences. Search all consequences for facts matching the query.

```python
def forward(facts F₀, rules RU, query q):
    F = ∅; new = F₀
    do
        F = F ∪ new; new = ∅
        for body ⇒ head ∈ RU do
            for σ ∈ match(body, F) do
                if σ(head) ∉ F then new.add(σ(head))
            end
        end
    while new ≠ ∅
    ans = ∅
    for f ∈ F do σ = match(q, f, ∅); if σ ≠ ⊥ then ans.add(σ)
    return ans
```
Forward Chaining Algorithm

Compute the next set of consequences whilst there are new consequences. Search all consequences for facts matching the query.

```python
def forward(facts F₀, rules RU, query q):
    F = ∅; new = F₀
    do
        F = F ∪ new; new = ∅
        for body ⇒ head ∈ RU do
            for σ ∈ match(body, F) do
                if σ(head) ∉ F then new.add(σ(head))
            end
        end
    while new ≠ ∅
    ans = ∅
    for f ∈ F do σ = match(q, f, ∅); if σ ≠ ⊥ then ans.add(σ)
    return ans
```
Forward Chaining Algorithm

Compute the next set of consequences whilst there are new consequences. Search all consequences for facts matching the query.

```python
def forward(facts F_0, rules RU, query q):
    F = ∅; new = F_0
    do
        F = F ∪ new; new = ∅
        for body ⇒ head ∈ RU do
            for σ ∈ match(body, F) do
                if σ(head) /∈ F then new.add(σ(head))
            end
        end
    end
    while new ≠ ∅
    ans = ∅
    for f ∈ F do σ = match(q, f, ∅); if σ ≠ ⊥ then ans.add(σ)
    return ans
```
Forward Chaining Algorithm

Compute the next set of consequences whilst there are new consequences. Search all consequences for facts matching the query.

```
def forward(facts F₀, rules RU, query q):
  F = ∅; new = F₀
  do
    F = F ∪ new; new = ∅
    for body ⇒ head ∈ RU do
      for σ ∈ match(body, F) do
        if σ(head) /∈ F then new.add(σ(head))
      end
    end
  while new ≠ ∅
  ans = ∅
  for f ∈ F do σ = match(q, f, ∅); if σ ≠ ⊥ then ans.add(σ)
  return ans
```
Observation: The current algorithm for matching against known consequences is inefficient; it involves multiple iterations over all known consequences.

Solution 1: Use heuristics to select the order in which facts in the body are matched e.g. pick least frequently occurring name first.

Solution 2: Store known facts in a data structure that facilitates quick lookup of matching facts. We will see such a data structure for unification towards the end of the course.
Observation: On each step the only new additions come from rules that are triggered by new facts.

Solution: Use the previous set of new facts as an initial filter to identify which rules are relevant and which further facts need to match against existing facts.
**Observation:** We can derive a lot of facts that are irrelevant to the query

**Solution 1:** Rewrite the knowledge base to remove/reduce rules that produce irrelevant facts. Computationally expensive but may be worth it if similar queries executed often. Similar to query optimisation in database.

**Solution 2:** Backward Chaining. Start from the query and work backwards to see which facts support it. This is what Prolog does.
Summary

Queries are facts possibly containing variables

To answer queries we can compute all consequences and check these

We can use forward chaining to compute consequences

This relies on matching, which can be tricky to implement efficiently

Next time: Prolog!