CS 2740 Knowledge Representation
Lecture 11

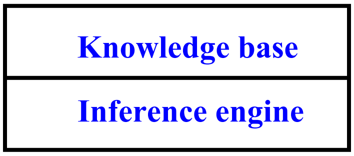
I. Production systems.
II. Frame-based systems.

Milos Hauskrecht

milos@cs.pitt.edu

5329 Sennott Square

Knowledge-based system



- **Knowledge base:**
  - A set of sentences that describe the world in some formal (representational) language (e.g. first-order logic)
  - Domain specific knowledge
- **Inference engine:**
  - A set of procedures that work upon the representational language and can infer new facts or answer KB queries (e.g. resolution algorithm, forward chaining)
  - Domain independent

Automated reasoning systems

- **Theorem provers**
  - Prove sentences in the first-order logic. Use inference rules, resolution rule and resolution refutation.
- **Deductive retrieval systems**
  - Systems based on rules (KBs in Horn form)
  - Prove theorems or infer new assertions
- **Production systems**
  - Systems based on rules with actions in antecedents
  - Forward chaining mode of operation
- **Semantic networks**
  - Graphical representation of the world, objects are nodes in the graphs, relations are various links
- **Frames:**
  - object oriented representation, some procedural control of inference

Production systems

Based on rules, but different from KBs in the Horn form

Knowledge base is divided into:

- **A Rule base (includes rules)**
- **A Working memory (includes facts)**

Rules: a special type of if – then rule

$$p_1 \wedge p_2 \wedge \ldots p_n \Rightarrow a_1, a_2, \ldots, a_k$$

Antecedent:
A conjunction of conditions

Consequent:
a sequence of actions

Basic operation:

- Check if the antecedent of a rule is satisfied
- Decide which rule to execute (if more than one rule is satisfied)
- Execute actions in the consequent of the rule

Working memory

- Consists of a set of facts – statements about the world but also can represent various data structures
- The exact syntax and representation of facts may differ across different systems
- **Examples:**
  - **predicates**
  - such as Red(car12)
  - but only ground statements

  or
  - **(type attr1:value1 attr2:value2 …) objects**

    such as: (person age 27 home Toronto)

    The type, attributes and values are all atoms

Rules

$$p_1 \wedge p_2 \wedge \ldots p_n \Rightarrow a_1, a_2, \ldots, a_k$$

- **Antecedents: conjunctions of conditions**
- **Examples:**
  - a conjunction of literals $A(x) \wedge B(x) \wedge C(y)$
  - simple negated or non-negated statements in predicate logic
- or
  - conjunctions of conditions on objects/object
  - (type attr1 spec1 attr2 spec2 …)
  - Where specs can be an atom, a variable, expression, condition

    (person age [n+4] occupation x)

    (person age $\{< 23 \wedge > 6\}$)

Production systems

$$p_1 \wedge p_2 \wedge \dots p_n \Rightarrow a_1, a_2, \dots, a_k$$

- **Consequent:** a sequence of actions
- An action can be:
  - **ADD** the fact to the working memory (WM)
  - **REMOVE** the fact from the WM
  - **MODIFY** an attribute field
  - **QUERY** the user for input, etc …
- **Examples:**

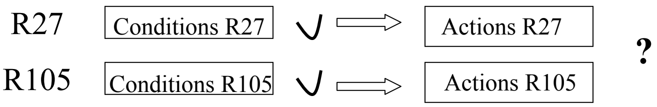
$$A(x) \wedge B(x) \wedge C(y) \Rightarrow add\ \ D(x)$$

- **Or**

(Student name *x*) ⇒ ADD (Person name *x*)

Production systems

- Use **forward chaining to do reasoning**:
  - If the antecedent of the rule is satisfied (rule is said to be “active”) then its consequent can be executed (it is “fired”)
- **Problem:** Two or more rules are active at the same time. Which one to execute next?



R27 Conditions R27 ✓ ⇒ Actions R27

R105 Conditions R105 ✓ ⇒ Actions R105

?

- Strategy for selecting the rule to be fired from among possible candidates is called **conflict resolution**

Production systems

- Why is conflict resolution important? Or, Why do we care about the order?
- Assume that we have two rules and the preconditions of both are satisfied:

R1: $A(x) \wedge B(x) \wedge C(y) \Rightarrow add\ \ D(x)$

R2: $A(x) \wedge B(x) \wedge E(z) \Rightarrow delete\ \ A(x)$

- What can happen if rules are triggered in different order?

Production systems

- Why is conflict resolution important? Or, Why do we care about the order?
- Assume that we have two rules and the preconditions of both are satisfied:

R1: $A(x) \wedge B(x) \wedge C(y) \Rightarrow add\ \ D(x)$

R2: $A(x) \wedge B(x) \wedge E(z) \Rightarrow delete\ \ A(x)$

- What can happen if rules are triggered in different order?
  - If R1 goes first, R2 condition is still satisfied and we infer D(x)
  - If R2 goes first we may never infer D(x)

Production systems

- **Problems with production systems:**
  - Additions and Deletions can change a set of active rules;
  - If a rule contains variables testing all instances in which the rule is active may require a large number of unifications.
  - Conditions of many rules may overlap, thus requiring to repeat the same unifications multiple times.
- **Solution: Rete algorithm**
  - gives more efficient solution for managing a set of active rules and performing unifications
  - Implemented in the system **OPS-5** (used to implement XCON – an expert system for configuration of DEC computers)

Rete algorithm

- Assume a set of rules:

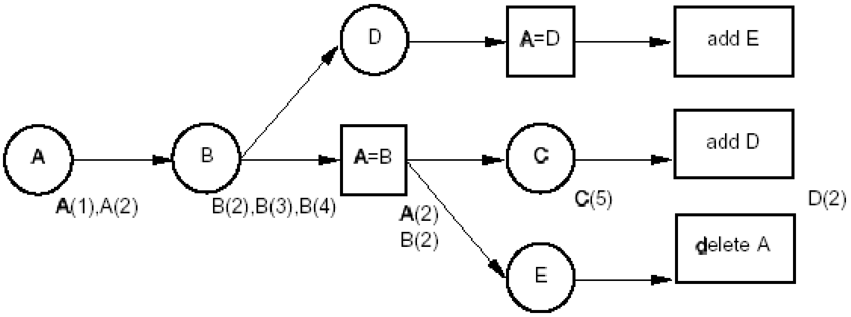
$$A(x) \wedge B(x) \wedge C(y) \Rightarrow add\ \ D(x)$$

$$A(x) \wedge B(y) \wedge D(x) \Rightarrow add\ \ E(x)$$

$$A(x) \wedge B(x) \wedge E(z) \Rightarrow delete\ \ A(x)$$

- And facts: $A(1), A(2), B(2), B(3), B(4), C(5)$
- **Rete:**
  - Compiles the rules to a network that merges conditions of multiple rules together (avoid repeats)
  - Propagates valid unifications
  - Reevaluates only changed conditions

Rete algorithm. Network.



Rules: $A(x) \wedge B(x) \wedge C(y) \Rightarrow \textit{add} \quad D(x)$

$A(x) \wedge B(y) \wedge D(x) \Rightarrow \textit{add} \quad E(x)$

$A(x) \wedge B(x) \wedge E(z) \Rightarrow \textit{delete} \quad A(x)$

Facts: $A(1),\ A(2),\ B(2),\ B(3),\ B(4),\ C(5)$

Conflict resolution strategies

- **Problem:** Two or more rules are active at the same time. Which one to execute next?
- **Solutions:**
  - **No duplication** (do not execute the same rule twice)
  - **Recency.** Rules referring to facts newly added to the working memory take precedence
  - **Specificity.** Rules that are more specific are preferred.
  - **Priority levels.** Define priority of rules, actions based on expert opinion. Have multiple priority levels such that the higher priority rules fire first.

OPS-5

OPS5 (R1):

- A production system – a programming language
- Used to build commercial expert systems like XCON for configuration of the DEC computers

OPS/R2: (Production Systems Technologies inc.)

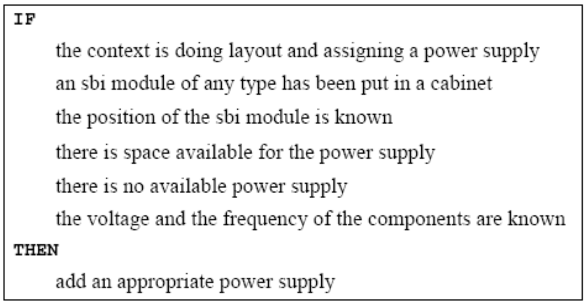
- Support for forward and backward chaining
- Improved Rete algorithm
- Object oriented-rules (with inheritance)
- Multiple WM
- User-defined control

OPS-5

System developed at CMU (as R1) and used extensively at DEC (now owned by Compaq) to configure early Vax computers

Nearly 10,000 rules for several hundred component types

Major stimulus for commercial interest in rule-based expert systems ★



```
IF
      the context is doing layout and assigning a power supply
      an sbi module of any type has been put in a cabinet
      the position of the sbi module is known
      there is space available for the power supply
      there is no available power supply
      the voltage and the frequency of the components are known
THEN
      add an appropriate power supply
```

Frame-based representation

Knowledge representation

Many different ways of representing the same knowledge. Representation may make inferences easier or more difficult.

Example:

- How to represent: “Car #12 is red.”

  Solution 1: ?

Knowledge representation

Many different ways of representing the same knowledge. Representation may make inferences easier or more difficult.

Example:

- How to represent: “Car #12 is red.”

  Solution 1: Red(car12).

  – It’s easy to ask “What’s red?”

  – But we can’t ask “what is the color of car12?”

  Solution 2: ?

Knowledge representation

Many different ways of representing the same knowledge. Representation may make inferences easier or more difficult.

Example:

- How to represent: “Car #12 is red.”

  Solution 1: Red(car12).

  – It’s easy to ask “What’s red?”

  – But we can’t ask “what is the color of car12?”

  Solution 2: Color (car12, red).

  – It’s easy to ask “What’s red?”

  – It’s easy to ask “What is the color of car12?”

  – Can’t ask “What property of car12 has value red?”

  Solution 3: ?

Knowledge representation

Many different ways of representing the same knowledge. Representation may make inferences easier or more difficult.

Example:

- How to represent: “Car #12 is red.”

  Solution 1: Red(car12).
  - It’s easy to ask “What’s red?”
  - But we can’t ask “what is the color of car12?”

  Solution 2: Color (car12, red).
  - It’s easy to ask “What’s red?”
  - It’s easy to ask “What is the color of car12?”
  - Can’t ask “What property of car12 has value red?”

  Solution 3: Prop(car12, color , red).
  - It’s easy to ask all these questions.

Knowledge representation

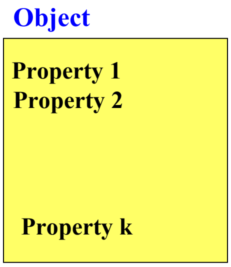
- Prop(Object, Property, Value)
- **Called:** object-property-value representation
- In **FOL statements** about the world, e.g. statements about objects are scattered around
- If we merge many properties of the object of the same type into one structure we get the object-centered representation:

Prop(Object, Property1, Value1)

Prop(Object, Property2, Value2)

…

Prop(Object, Property-n, Value-n)



Object

Property 1
Property 2

Property k

Object-centered representations

Objects: a natural way to organize the knowledge about

- **physical objects:**
  - a desk has a surface-material, # of drawers, width, length, height, color, procedure for unlocking, etc.
  - some variations: no drawers, multi-level surface
- **situations:**
  - a class: room, participants, teacher, day, time, seating arrangement, lighting, procedures for registering, grading, etc.
  - leg of a trip: destination, origin, conveyance, procedures for buying ticket, getting through customs, reserving hotel room, locating a car rental etc.

Important: Objects enable grouping of procedures for determining the properties of objects, their parts, interaction with parts

Frames

Predecessor of object-oriented systems

Two types of frames:

- **individual frames**
  - represent a single object like a person, part of a trip
- **generic frames**
  - represent categories of objects, like students

Example:

- A generic frame: Europian city
- Individual frames: Paris, London, Prague

Frames

- An individual frame is a named list of buckets called **slots**.
- What goes in the bucket is called a **filler of the slot**.

(*frame-name*

 <slot-name1 filler1>

 <slot-name2 filler2 > …)

Frames

Individual frames have a special slot called : INSTANCE-OF whose filler is the name of **a generic frame**:

Example:

```
(toronto            % lower case for individual frames
    <:INSTANCE-OF CanadianCity>
    <:Province ontario>
    <:Population 4.5M>…)
```

Generic frames may have IS-A slot that includes generic frame

- (CanadianCity % upper case for generic frames

```
    <:IS-A City>
    <:Province CanadianProvince>
    <:Country canada>…)
```

Frames – inference control

Slots in **generic frames** can have associated procedures that are executed and ‘control’ inference

Two types of procedures:

- **IF-NEEDED procedure**; executes when no slot filler is given and the value is needed

  (Table

    <:Clearance [**IF-NEEDED** computeClearance]> …)

- **IF-ADDED procedure**. If a slot filler is given its effect may propagate to other frames (say to assure constraints)

  (Lecture

      <:DayOfWeek WeekDay>

      <:Date [**IF-ADDED** computeDayOfWeek]> …)

- the filler for :DayOfWeek will be calculated when :Date is filled

Frames – defaults

(CanadianCity

    <:**IS-A** City>

    <:Province CanadianProvince>

    <:Country canada>…)

(city134

    <:**INSTANCE-OF** CanadianCity>

    ..)

- A country filler is:

Frames – defaults

(CanadianCity
<:**IS-A** City>
<:Province CanadianProvince>
<:Country canada>…)
(city134
<:**INSTANCE-OF** CanadianCity>
..)

- A country filler is: canada

(city135
<:**INSTANCE-OF** CanadianCity>
<:Country holland>)

- A country filler is:

Frames – defaults

(CanadianCity
<:**IS-A** City>
<:Province CanadianProvince>
<:Country canada>…)
(city134
<:**INSTANCE-OF** CanadianCity>
..)

- A country filler is: canada

(city135
<:**INSTANCE-OF** CanadianCity>
<:Country holland>)

- A country filler is: holland

Frames – inheritance

- Procedures and fillers of more general frame are applicable to more specific frame through the inheritance mechanism

```
(CoffeeTable
        <:IS-A Table> ...)
(MahoganyCoffeeTable
        <:IS-A CoffeeTable> ...)

(Elephant
        <:IS-A Mammal>
        <:Colour gray> ...)
(RoyalElephant
        <:IS-A Elephant>
        <:Colour white>)
(clyde
        <:INSTANCE-OF RoyalElephant>)
```

Frames – reasoning

Basic reasoning goes like this:

1. user instantiates a frame, i.e., declares that an object or situation exists
2. slot fillers are inherited where possible
3. inherited **IF-ADDED** procedures are run, causing more frames to be instantiated and slots to be filled.

If the user or any procedure **requires the filler of a slot** then:

1. if there is a filler, it is used
2. otherwise, an inherited **IF-NEEDED** procedure is run, potentially causing additional actions

Frames – reasoning

Global reasoning:

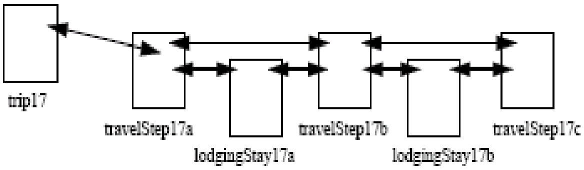
- make frames be major situations or object-types you need to flesh out
- express constraints between slots as **IF-NEEDED** and **IF-ADDED** procedures
- fill in default values when known

Frames – example

A system to **assist in travel planning**

Basic frame types:

- a Trip - be a sequence of TravelSteps, linked through slots
- a TravelStep - terminates in a LodgingStay
- a LodgingStay linked to arriving and departing TravelStep(s)
- TravelSteps includes LodgingStays of their origin and destination



```
(trip17
    <:INSTANCE-OF Trip>
    <:FirstStep travelStep17a>
    <:Traveler  ronB> ...)
```

Frames - examples

TravelSteps and LodgingStays share some properties (e.g., :BeginDate, :EndDate, :Cost, :PaymentMethod), so we might create a more general category as the parent frame for both of them:

```
(Trip
     <:FirstStep TravelStep>
     <:Traveler  Person>
     <:BeginDate  Date>
     <:TotalCost Price> ...)
```

```
(TripPart
     <:BeginDate>
     <:EndDate>
     <:Cost>
     <:PaymentMethod> ...)
```

```
(TravelStep
     <:IS-A  TripPart>
     <:Means>
     <:Origin>   <:Destination>
     <:NextStep>   <:PreviousStep>
     <:DepartureTime>   <:ArrivalTime>
     <:OriginLodgingStay>
     <:DestinationLodgingStay> ...)
```

```
(LodgingStay
     <:IS-A  TripPart>
     <:ArrivingTravelStep>
     <:DepartingTravelStep>
     <:City>
     <:LodgingPlace> ...)
```

Frames - example

Embellish frames with defaults and procedures

```
(TravelStep
     <:Means  airplane> ...)
(TripPart
     <:PaymentMethod  visaCard> ...)
(TravelStep
     <:Origin [IF-NEEDED {if no SELF:PreviousStep then newark}]>)
(Trip
     <:TotalCost
        [IF-NEEDED
          { x←SELF:FirstStep;
           result←0;
           repeat
            {  if  exists x:NextStep
               then
                     { result←result + x:Cost +
                                  x:DestinationLodgingStay:Cost;
                        x←x:NextStep }
               else  return  result+x:Cost }}]>)
```

assume this is 0 if there is no LodgingStay

Program notation (for an imaginary language):

- *SELF* is the current frame being processed
- if *x* refers to an individual frame, and *y* to a slot, then *xy* refers to the filler of the slot

Frames - example

```
(TravelStep
    <:NextStep
        [IF-ADDED
         {if  SELF:EndDate ≠ SELF:NextStep:BeginDate
          then
             SELF:DestinationLodgingStay ←
                 SELF:NextStep:OriginLodgingStay ←
                     create new LodgingStay
                         with :BeginDate = SELF:EndDate
                         and with :EndDate = SELF:NextStep:BeginDate
                         and with :ArrivingTravelStep = SELF
                         and with :DepartingTravelStep = SELF:NextStep
               …}]>
  …)
```

Note: default :City of LodgingStay, etc. can also be calculated:

```
(LodgingStay
    <:City [IF-NEEDED  {SELF:ArrivingTravelStep:Destination}]…>   …)
```

Frames - example

Propose a trip to Toronto on Dec. 21, returning Dec. 22

```
(trip18
    <:INSTANCE-OF  Trip>
    <:FirstStep travelStep18a>)
```

the first thing to do is to create the trip and the first step

```
(travelStep18a
    <:INSTANCE-OF  TravelStep>
    <:BeginDate 12/21/98>
    <:EndDate 12/21/98>
    <:Means>
    <:Origin>
    <:Destination toronto>
    <:NextStep> <:PreviousStep>
    <:DepartureTime>  <:ArrivalTime>)
```

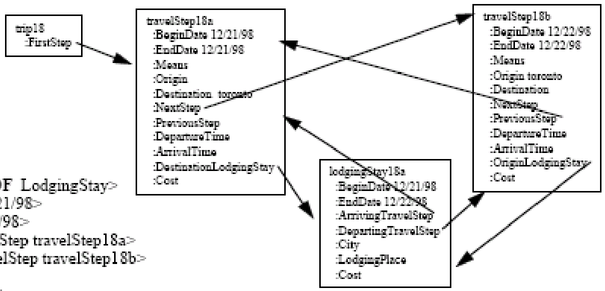
```
(travelStep18b
    <:INSTANCE-OF  TravelStep>
    <:BeginDate 12/22/98>
    <:EndDate 12/22/98>
    <:Means>
    <:Origin toronto>
    <:Destination>
    <:NextStep>
    <:PreviousStep travelStep18a>
    <:DepartureTime>  <:ArrivalTime>)
```

the next thing to do is to create the second step and link it to the first by changing the :NextStep

```
(travelStep18a
    <:NextStep travelStep18b>)
```

Frames - example

IF-ADDED on :NextStep then creates a LodgingStay:



trip18
:FirstStep

travelStep18a
:BeginDate 12/21/98
:EndDate 12/21/98
:Means
:Origin
:Destination toronto
:NextStep
:PreviousStep
:DepartureTime
:ArrivalTime
:DestinationLodgingStay
:Cost

travelStep18b
:BeginDate 12/22/98
:EndDate 12/22/98
:Means
:Origin toronto
:Destination
:NextStep
:PreviousStep
:DepartureTime
:ArrivalTime
:OriginLodgingStay
:Cost

lodgingStay18a
:BeginDate 12/21/98
:EndDate 12/22/98
:ArrivingTravelStep
:DepartingTravelStep
:City
:LodgingPlace
:Cost

(lodgingStay18a
<:**INSTANCE-OF** LodgingStay>
<:BeginDate 12/21/98>
<:EndDate 12/22/98>
<:ArrivingTravelStep travelStep18a>
<:DepartingTravelStep travelStep18b>
<:City>
<:LodgingPlace>)

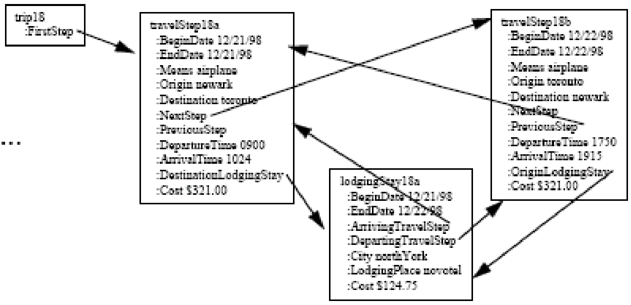
If requested, **IF-NEEDED** can provide :City for lodgingStay18a (toronto)

which could then be overridden by hand, if necessary
(e.g. usually stay in North York, not Toronto)

Similarly, apply default for :Means and default calc for :Origin

Frames - example

So far...



trip18
:FirstStep

travelStep18a
:BeginDate 12/21/98
:EndDate 12/21/98
:Means airplane
:Origin newark
:Destination toronto
:NextStep
:PreviousStep
:DepartureTime 0900
:ArrivalTime 1024
:DestinationLodgingStay
:Cost $321.00

travelStep18b
:BeginDate 12/22/98
:EndDate 12/22/98
:Means airplane
:Origin toronto
:Destination newark
:NextStep
:PreviousStep
:DepartureTime 1750
:ArrivalTime 1915
:OriginLodgingStay
:Cost $321.00

lodgingStay18a
:BeginDate 12/21/98
:EndDate 12/22/98
:ArrivingTravelStep
:DepartingTravelStep
:City northYork
:LodgingPlace novotel
:Cost $124.75

Finally, we can use :TotalCost **IF-NEEDED** procedure (see above) to calculate the total cost of the trip:

- result← 0, x←travelStep18a, x:NextStep=travelStep18b
- result←0+$321.00+$124.75; x← travelStep18b, x:NextStep=NIL
- return: result=$445.75+$321.00 = $766.75

Using a frame-based system

Main purpose of the above:

- embellish a sketchy description with defaults, implied values
- maintain consistency
- use computed values to:
  - allow derived properties to look explicit
  - avoid up front, potentially unneeded computation

Application: Monitoring

- hook to a DB, watch for changes in values
- like an ES somewhat, but monitors are more object-centered, inherited

Frames

- **Declarative vs procedural representation**
  - **Frames allow both declarative and procedural control**
- **Inference is controled via procedures**
  - **Can be very tightly controlled, much like an object oriented programming**
- **Differences from OOP:**
  - Frames control via: instantiate/ inherit/trigger cycles
  - OOP: objects sending messages