8.

Object-Oriented Representation

Organizing procedures

With the move to put control of inference into the user's hands, we're focusing on more <u>procedural</u> <u>representations</u>

knowing facts by executing code

Even production systems are essentially programming languages.

Note also that everything so far is *flat*, i.e., sentence-like representations

- information about an object is scattered in axioms
- procedure fragments and rules have a similar problem

With enough procedures / sentences in a KB, it could be critical to *organize* them

- production systems might have rule sets, organized by context of application
- but this is not a natural, *representational* motivation for grouping

Object-centered representation

Most obvious organizational technique depends on our ability to see the world in terms of objects

- 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, built-in walls (carrel)
- also, *situations* can be object-like:
  - 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.

Suggests clustering procedures for determining properties, identifying parts, interacting with parts, as well as constraints between parts, all of *objects*

- legs of desk connect to and support surface
- beginning of a travel leg and destination of prior one

object-centered constraints

Situation recognition

Focus on objects as an organizational / chunking mechanism to make some things easier to find

Suggests a different kind of reasoning than that covered so far

basic idea originally proposed by Marvin Minsky

- recognize (guess) situation; activate relevant object representations
- use those object representations to set up expectations

  some for verification; some make it easier to interpret new details
- flesh out situation once you've recognized

Wide applicability, but typical applications include

- relationship recognition e.g., story understanding
- data monitoring
- propagation and enforcement of constraints for planning tasks

  this latter is most doable and understandable,
  so we will concentrate on it

Basic frame language

Let's call our object structures <u>frames</u>

note wide variety of interpretations in literature

Two types:

- <u>individual</u> frames

  represent a single object like a person, part of a trip

- <u>generic</u> frames

  represent categories of objects, like students

An individual frame is a named list of buckets called <u>slots</u>. What goes in the bucket is called a <u>filler</u> of the slot. It looks like this:

(*frame-name*
  <slot-name1 filler1>
  <slot-name2 filler2 > …)

where frame names and slot names are atomic, and fillers are either numbers, strings or the names of other individual frames.

Notation:		
	individual frames:	toronto
	slot names:	:Population (note ":" at start)
	generic frames:	CanadianCity

Instances and specializations

Individual frames have a special slot called <u>:INSTANCE-OF</u> whose filler is the name of a generic frame:

(toronto
  <:**INSTANCE-OF** CanadianCity>
  <:Province ontario>
  <:Population 4.5M>…)

(tripLeg123-1
  <:**INSTANCE-OF** TripLeg>
  <:Destination toronto>…)

Generic frames have a syntax that is similar to that of individual frames, except that they have a slot called <u>:IS-A</u> whose filler is the name of another generic frame

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

We say that the frame toronto is an <u>instance</u> of the frame CanadianCity and that the frame CanadianCity is a <u>specialization</u> of the frame City

Procedures and defaults

Slots in generic frames can have associated procedures

1. computing a filler (when no slot filler is given)

```
(Table
   <:Clearance  [IF-NEEDED computeClearanceFromLegs]> …)
```

2. propagating constraints (when a slot filler is given)

```
(Lecture
   <:DayOfWeek  WeekDay>
   <:Date  [IF-ADDED computeDayOfWeek]> …)
```

If we create an instance of Table, the :Clearance will be calculated as needed. Similarly, the filler for :DayOfWeek will be calculated when :Date is filled.

For instances of CanadianCity, the :Country slot will be filled automatically. But we can also have

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

The filler canada in CanadianCity is considered a default value.

IS-A and inheritance

Specialization relationships imply that procedures and fillers of more general frame are applicable to more specific frame: inheritance.

For example, instances of MahoganyCoffeeTable will inherit the procedure from Table (via CoffeeTable)

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

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

Similarly, default values are inheritable, so that Clyde inherits a colour from RoyalElephant, not Elephant

```
(Elephant
      <:IS-A Mammal>
      <:Colour  gray> ...)

(RoyalElephant
      <:IS-A Elephant>
      <:Colour  white>)

(clyde
      <:INSTANCE-OF  RoyalElephant>)
```

Reasoning with frames

Basic (local) 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

Globally:

- 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

⇒ like a fancy, semi-symbolic spreadsheet

Planning a trip

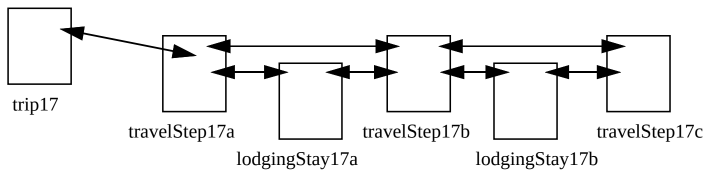
A simple example: a frame system to assist in travel planning (and possibly documentation – automatically generate forms)

Basic structure (main frame types):

- a Trip will be a sequence of TravelSteps

  these will be linked together by slots

- a TravelStep will usually terminate in a LodgingStay (except the last, or one with two travels on one day)
  - a LodgingStay will point to its arriving TravelStep and departing TravelStep
  - TravelSteps will indicate the LodgingStays of their origin and destination



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

Parts of a trip

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                                        (TripPart
     <:FirstStep TravelStep>                      <:BeginDate>
     <:Traveler  Person>                          <:EndDate>
     <:BeginDate  Date>                           <:Cost>
     <:TotalCost Price> ...)                      <:PaymentMethod> …)

(TravelStep                                  (LodgingStay
     <:IS-A  TripPart>                            <:IS-A  TripPart>
     <:Means>                                     <:ArrivingTravelStep>
     <:Origin>   <:Destination>                   <:DepartingTravelStep>
     <:NextStep>   <:PreviousStep>                <:City>
     <:DepartureTime>   <:ArrivalTime>            <:LodgingPlace> …)
     <:OriginLodgingStay>
     <:DestinationLodgingStay> …)
```

Travel defaults and procedures

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

More attached procedures

```
(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 in action

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>)
```

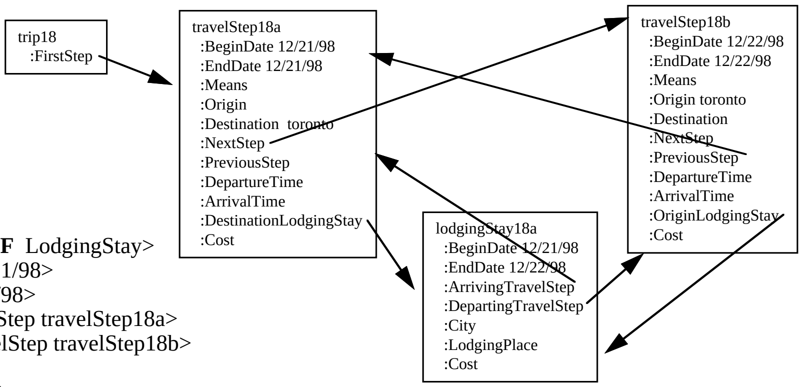
```
(travelStep18b
     <:INSTANCE-OF  TravelStep>
     <:BeginDate 12/22/98>
     <:EndDate 12/22/98>
     <:Means>
     <:Origin toronto>
     <:Destination>
     <:NextStep>
     <:PreviousStep travelStep18a>
     <:DepartureTime>  <:ArrivalTime>)
```

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

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

Triggering procedures

IF-ADDED on :NextStep then creates a LodgingStay:



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

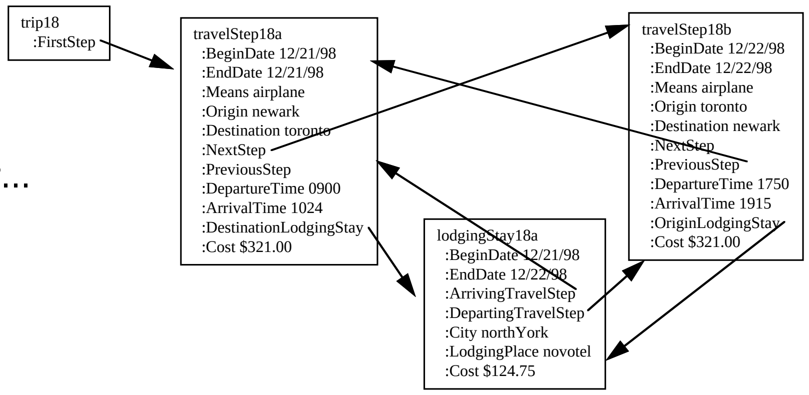
If requested, **IF-NEEDED** can provide :City for lodgingStay18a (toronto)

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

Finding the cost of the trip

So far...



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 the formalism

Main purpose of the above: embellish a sketchy description with defaults, implied values

- maintain consistency
- use computed values to
  1. allow derived properties to look explicit
  2. avoid up front, potentially unneeded computation

Monitoring

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

Scripts for story understanding

generate expectations (e.g., restaurant)

Real, Minsky-like commonsense reasoning

- local cues ⇒ potentially relevant frames ⇒ further expectations
- look to match expectations ; mismatch ⇒ “differential diagnosis”

Extensions

1. Types of procedures
   - **IF-REMOVED**

     e.g., remove TravelStep ⇒ remove LodgingStay
   - “servants” and “demons”

     flexible “pushing” and “pulling” of data
2. Slots
   - multiple fillers
   - “facets” – more than just defaults and fillers
     - [**REQUIRE** <class>] (or procedure)
     - **PREFER** – useful if conflicting fillers
3. Metaframes

   (CanadianCity <:**INSTANCE-OF** GeographicalCityType> …)

   (GeographicalCityType <:**IS-A** CityType>
   <:AveragePopulation NonNegativeNumber> …)
4. Frames as actions (“scripts”)

Object-oriented programming

Somewhat in the manner of production systems, specifying problems with frames can easily slide into a style of *programming*, rather than a declarative object-oriented modeling of the world

- note that direction of procedures (pushing/pulling) is explicitly specified

  not declarative

This drifts close to conventional object-oriented programming (developed concurrently).

- same advantages:
  - definition by specialization
  - localization of control
  - encapsulation
  - etc.
- main difference:
  - frames: centralized, conventional control regime (instantiate/ inherit/trigger)
  - object-oriented programming: objects acting as small, independent agents sending each other messages