EXAMPLE 1: USING FRAMES TO PLAN A TRIP

From <Chapter 8 ■ Object-Oriented Representation> in the book <Knowledge Representation and Reasoning> by Ronald J.Brachman and Hector J.Levesque

In general, in a procedural object-oriented representation system, we consider the kinds of reasoning operations that are relevant for the various types of objects in our application, and we design procedures to deal with them. Marvin Minsky in 1975 suggested the idea of using object oriented groups of procedures to recognize and deal with new situations. Minsky used the term frame for the data structure used to represent these situations. The procedures attached to frames give us a flexible, organized framework for computation.
In this example We will see how the “symbolic spreadsheet” style of reasoning in frame systems is used. This might be particularly useful in supporting the documentation one often uses in a company for reporting expenses.
The basic structure of our representation involves two main types of frames: Trip and TravelStep. A Trip will have a sequence of TravelSteps, A TravelStep will usually terminate in a LodgingStay. In order to make the correspondences work out correctly (and to be able to keep track of what is related to what), a LodgingStay will use slots to point to its arriving TravelStep and its departing TravelStep. Similarly, TravelSteps will indicate the LodgingStays at their origin and destination. Graphically, for a trip with three legs (instances of TravelStep), we might sketch the relationships as in Figure 8.1.

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

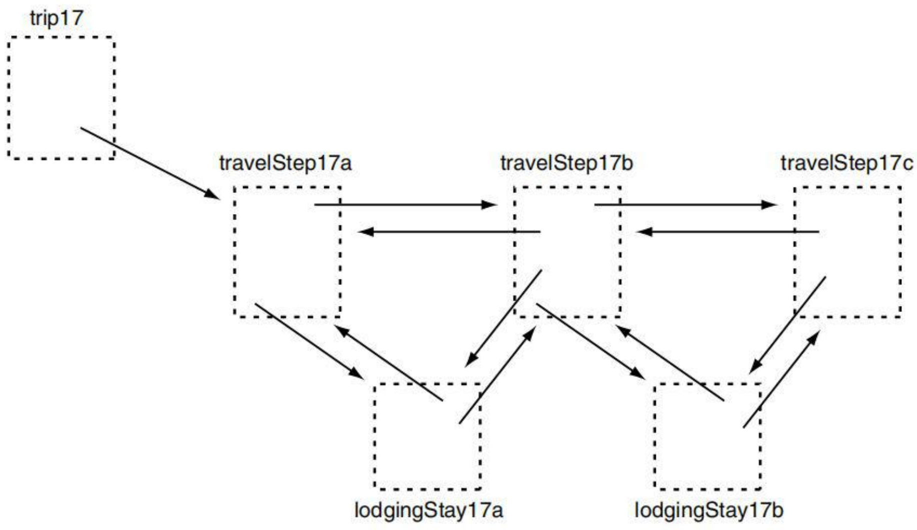

Figure 8.1.

A specific Trip, say trip17, might look like this:

```
(trip17
    <:INSTANCE-OF Trip>
    <:FirstStep travelStep17a>
    <:Traveler ronB>
    <:BeginDate 11/13/98>
    <:EndDate 11/18/98>
    <:TotalCost $1752.45>
    ...)
```

In general, instances of TravelStep and LodgingStay will share some properties (e.g., each has a beginning date, an end date, a cost, and
a payment method), so for representational conciseness, we might posit a more general category, TripPart, of which the two other frames would be specializations:

```
(TripPart
    <:BeginDate Date>
    <:EndDate Date>
    <:Cost Price>
    <:PaymentMethod FormOfPayment>
    ...)
  (LodgingStay
      <:IS-A TripPart>
      <:Place City>
      <:LodgingPlace LodgingPlace>
      <:ArrivingTravelStep TravelStep>
      <:DepartingTravelStep TravelStep>
      ...)
  (TravelStep
        <:IS-A TripPart>
        <:Origin City>
        <:Destination City>
        <:OriginLodgingStay LodgingStay>
        <:DestinationLodgingStay LodgingStay>
        <:Means FormOfTransportation>
        <:DepartureTime Time>
        <:ArrivalTime Time>
        <:NextStep TravelStep>
        <:PreviousStep TravelStep>
        ...)
```

This gives us our basic overall structure for a trip. Next we embellish the frame structure with various defaults as well as procedures that will help us enforce constraints. For example, our trips might most often be made by air, in which case the default filler for the :Means slot of a TravelStep should be airplane:

```
(TravelStep
        <:Means airplane> ...)
```

We might also make a habit of paying for parts of trips with a Visa card:

```
(TripPart
    <:PaymentMethod visaCard> ...)
```

However, perhaps because it provides insurance, we may prefer American Express for travel steps, overriding this default:

```
(TravelStep
    <:PaymentMethod americanExpressCard> ...)
```

As indicated earlier, not all inherited fillers of slots will necessarily be specified as fixed values; it may be more appropriate to compute them from the current circumstances. For example, it would be appropriate to automatically set up the origin of a travel step as our home airport, say Newark, as long as there was no previous travel step—in other words, Newark is the default airport for the beginning of a trip. To do this we introduce two pieces of notation:

- if x refers to an individual frame and y to a slot, then xy refers to the
filler of the slot for the frame;
- SELF will be a way to refer to the frame currently being processed.

Our travel step description would then be augmented to look like this:

```
(TravelStep
    <:Origin
        [IF-NEEDED
            {if no SELF:PreviousStep
                then newark
                else SELF:PreviousStep:Destination}]> ...)
```

This attached procedure says that for any TravelStep, if we want its origin city, use the destination of the previous TravelStep, or newark if there is none.

Another useful thing to do with a travel planning symbolic spreadsheet would be to compute the total cost of a trip from the costs of each of its parts:

```
(Trip
    <:TotalCost
        [IF-NEEDED
            {let result←0;
            let x←SELF:FirstStep;
            repeat
                {if exists x:NextStep
                    then
                        {result←result + x:Cost
                        if exists x:DestinationLodgingStay then
                            result←result + x:DestinationLodgingStay:Cost;
                        x←x:NextStep}
                    else return result + x:Cost}}]> ...)
```

This IF-NEEDED procedure (written in a suggestive pseudocode) iterates through the travel steps, starting at the trip’s :FirstStep. At each step, it adds the cost of the step itself (x:Cost) to the previous result, and if there is a subsequent step, the cost of the lodging stay between those two steps, if any (x:DestinationLodgingStay:Cost).

Another useful thing to expect an automatic travel documentation system to do would be to create a skeletal lodging stay instance each time a new travel leg was added. The following IF-ADDED procedure does a basic form of this:

```
(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 that the first thing done is to confirm that the next travel leg begins on a different day than the one we are starting with ends; presumably no lodging stay is needed if the two travel legs join on the same day.

Note also that the default :Place of a LodgingStay (and other fillers) could also be calculated as another piece of automatic processing:

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

This might be a fairly weak default, however, and its utility would depend on the particular application. It is quite possible that a traveller's preferred default city for lodging is different than the destination city for the arriving leg of the trip (e.g., flights may arrive in San Francisco, but I may prefer as a default to stay in Palo Alto).

We now consider how the various frame fragments we have created might work together in specifying a trip. Imagine that we propose a trip to Toronto on December 21, 2006, returning home the following day. First, we create an individual frame for the overall trip (call it trip18), and one for the first leg of the trip:

```
(trip18
    <:INSTANCE-OF Trip>
     <:FirstStep travelStep18a>)
(travelStep18a
    <:INSTANCE-OF TravelStep>
    <:Destination toronto>
    <:BeginDate 12/21/06>
     <:EndDate 12/21/06>)
```

Because we know we are to return home the next day, we create the second leg of the trip:

```
(travelStep18b
        <:INSTANCE-OF TravelStep>
        <:Destination newark>
        <:BeginDate 12/22/06>
        <:EndDate 12/22/06>
          <:PreviousStep travelStep18a>)
```

The state of affairs after creating travelStep18b is pictured in Figure 8.2.

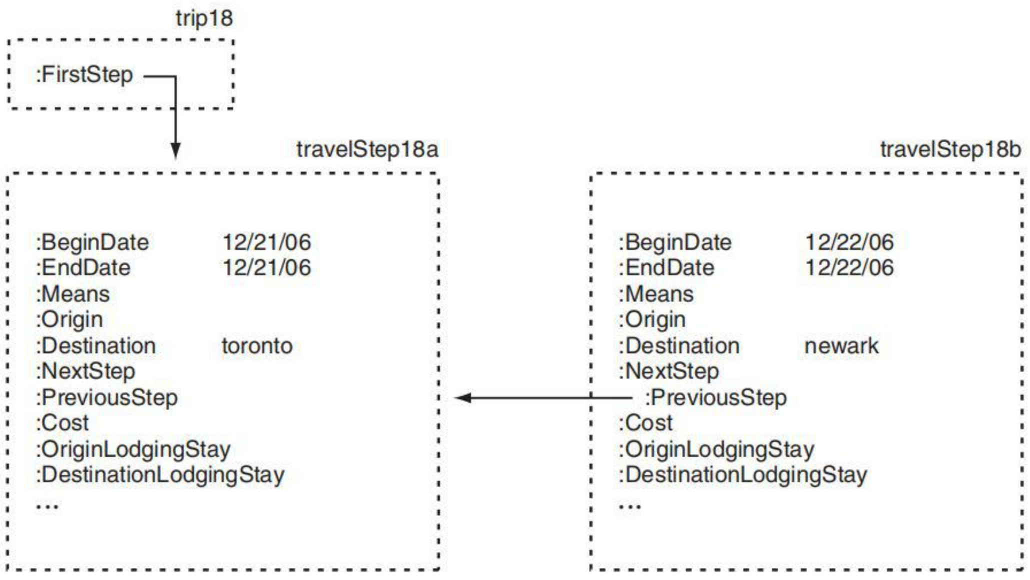

Figure 8.2.

To complete the initial setup, travelStep18a will need its :NextStep slot filled with travelStep18b. (Note that this could be done automatically with an **IF-ADDED** procedure on travelStep: PreviousStep triggered from travelStep18b.) As a consequence of the assignment of travelStep18b as the: NextStep of travelStep18a, a default LodgingStay is automatically created to represent the overnight stay between those two legs of the trip (using the IF-ADDED procedure on the :NextStep slot):

```
(lodgingStay18a
        <:INSTANCE-OF LodgingStay>
        <:BeginDate 12/21/06>
        <:EndDate 12/22/06>
        <:ArrivingTravelStep travelStep18a>
        <:DepartingTravelStep travelStep18b>)
```

Note that the **IF-NEEDED** procedure for the :Place slot of LodgingStay would infer a default filler of toronto for lodgingStay18a, if required. Once we have established the initial structure, we can see how the :Means slot of either step would be filled by default, and a query about the :Origin slot of either step would produce an appropriate default value, as in Figure 8.3 (note that we have included in the figure the values derived by the IF-NEEDED procedures).

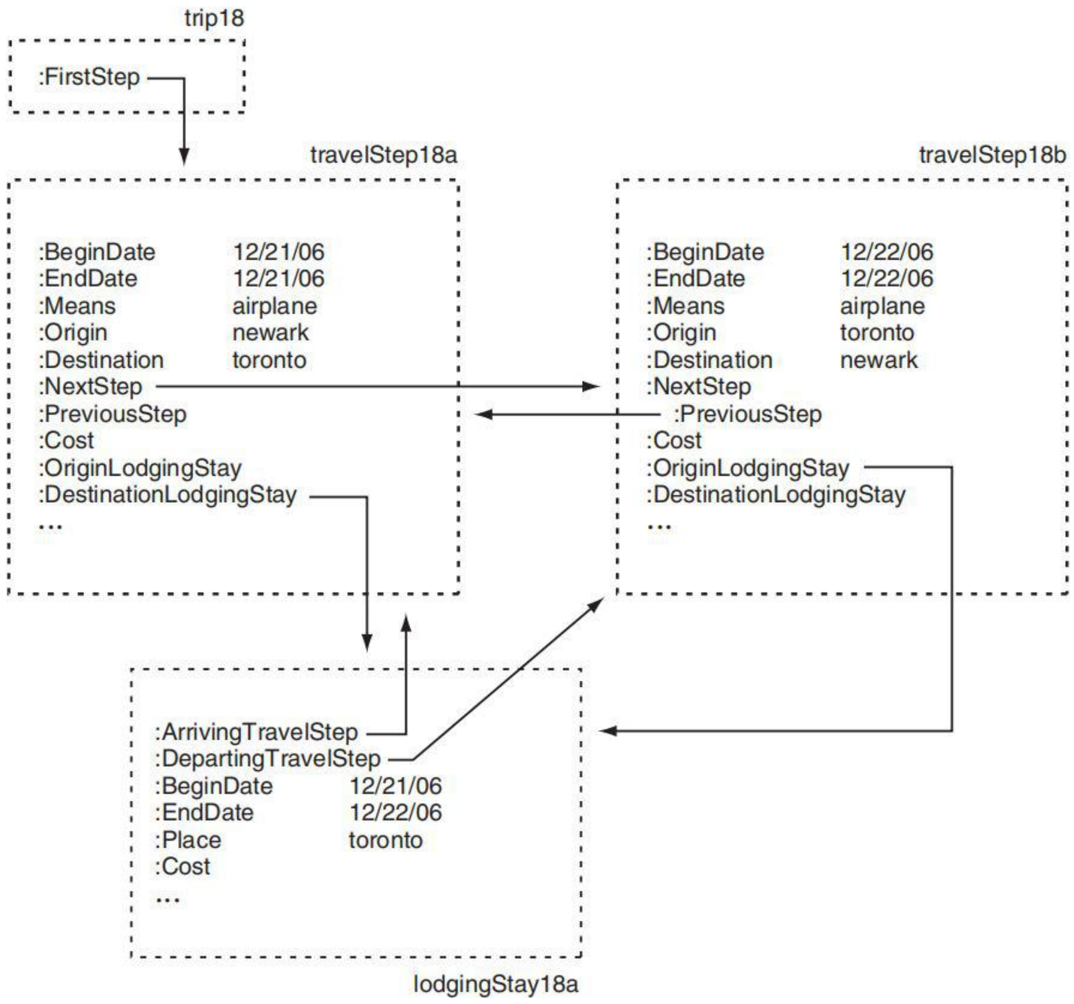

Figure 8.3

For a final illustration, imagine that we have over the course of our trip filled in the :Cost slots for each of the instances of TripPart as follows: travelStep18a:Cost is $321.00; travelStep18b:Cost is $321.00; and lodgingStay18b:Cost is $124.75. If we ask for the :TotalCost of the entire trip, the IF-NEEDED procedure defined earlier will come into play (assuming the :TotalCost slot has not already been filled manually). Given the final state of the trip as completed by the cost assertions, the calculation proceeds as follows:

- result is initialized to 0, and x is initialized to travelStep18a, which makes x:NextStep be travelStep18b;
- the first time through the repeat loop, result is set to the sum of result (0), the cost of x ($321.00), and the cost of the: DestinationLodgingStay of the current step (lodgingStay18a) ($124.75); x is then set to travelStep18b;
- the next time through, because x (travelStep18b) has no following step, the loop is broken and the sum of result ($445.75) and the cost of x ($321.00) is returned.

As a result, a grand total of $766.75 is taken to be the :TotalCost of trip18.

Summary: The trip planning example considered here is typical of how frame systems have been used: Start with a sketchy description of some circumstance and embellish it with defaults and

implied values. The **IF-ADDED** procedures can make updates easier and help to maintain consistency; the **IF-NEEDED** procedures allow values to be computed only when they are needed. There is a tradeoff here, of course, and which type of procedure to use in an application will depend on the potential value to the user of seeing implied values computed up front versus the value of waiting to do computation only as required.
The whole point of object-oriented reasoning is to determine the sort of questions appropriate for a type of object and to design procedures to answer them.

Extension:
Other procedures: An obvious way to increase the expressiveness and utility of the frame mechanism is to include other types of procedures. For trips, for example, we have only considered two forms of questions, exemplified by “What is the total cost of a trip?” (handled by an IF-NEEDED procedure) and “What should I do if I find out about a new leg of a trip?” (handled by an IF-ADDED procedure). Other questions that do not fit these two patterns are certainly possible, such as “What should I do if I cancel a leg of a trip?” (requiring some sort of “if-removed” procedure), “How do I recognize an overly expensive trip?” (**Common sense reasoning**: use local cues from a situation to suggest potentially relevant frames, which in turn would set up further expectations that could drive investigation procedures. For example, a situation where many people in a room were holding what appeared to be wrapped packages, and balloons and cake were in evidence. This would suggest a birthday party, and prompt us to look for the focal person at the party (a key slot of the birthday party frame) and to interpret the meaning of lit candles in a certain way. Expectations set up by the suggested frames could be used to confirm the current hypothesis (that this is a birthday party). If they were subsequently violated, then an appropriately represented “differential diagnosis” attached to the frame could lead the system to suggest other candidate frames, taking the reasoning in a different direction. For example, no candles on the cake and an adult focal person or persons could suggest a retirement or anniversary party.), or “What do I need to look out for in an overseas trip?” and so on.