7 Rules and Rule Chaining

In this chapter, you learn about the use of easily-stated *if–then rules* to solve problems. In particular, you learn about *forward chaining* from assertions and *backward chaining* from hypotheses.

By way of illustration, you learn about two toy systems; one identifies zoo animals, the other bags groceries. These examples are analogous to influential, classic systems that diagnose diseases and configure computers.

You also learn about how to implement rule-based systems. You learn, for example, how search methods can be deployed to determine which of many possible rules are applicable during backward chaining, and you learn how the *rete procedure* does efficient forward chaining.

When you have finished this chapter, you will understand the key ideas that support many of the useful applications of artificial intelligence. Such applications are often mislabeled **expert systems**, even though their problem-solving behavior seems more like that of human novices, rather than of human experts.

RULE-BASED DEDUCTION SYSTEMS

Rule-based problem-solving systems are built using rules like the following, each of which contains several *if* patterns and one or more *then* patterns:

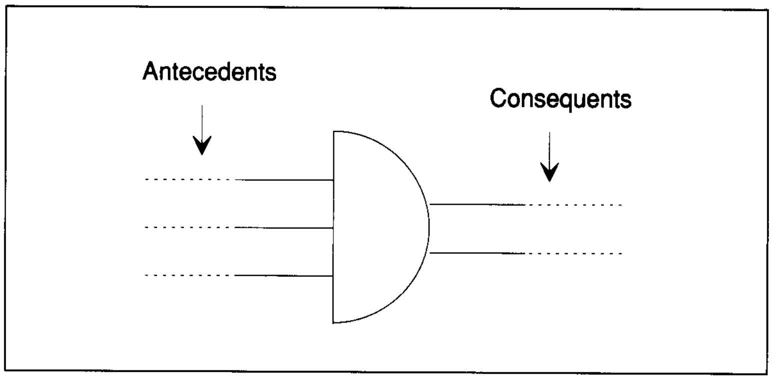


Figure 7.1 A convenient graphical notation for antecedent–consequent rules. The symbol, appropriately, is the same as the one used in digital electronics for AND gates.

```
Rn    If     if_1
             if_2
              ⋮
      then   then_1
             then_2
              ⋮
```

In this section, you learn how rule-based systems work.

Many Rule-Based Systems Are Deduction Systems

A statement that something is true, such as "Stretch has long legs," or "Stretch is a giraffe," is an **assertion.**[†] In all rule-based systems, each *if* pattern is a pattern that may match one or more of the assertions in a collection of assertions. The collection of assertions is sometimes called **working memory**.

In many rule-based systems, the *then* patterns specify new assertions to be placed into working memory, and the rule-based system is said to be a **deduction system**. In deduction systems, the convention is to refer to each *if* pattern as an **antecedent** and to each *then* pattern as a **consequent**. Figure 7.1 shows a graphical notation for deduction-oriented **antecedent–consequent rules**.

Sometimes, however, the *then* patterns specify actions, rather than assertions—for example, "Put the item into the bag"—in which case the rule-based system is a **reaction system**.

In both deduction systems and reaction systems, **forward chaining** is the process of moving from the *if* patterns to the *then* patterns, using the *if* patterns to identify appropriate situations for the deduction of a new assertion or the performance of an action.

[†]Facts and assertions are subtly different: A *fact* is something known to be true; an assertion is a statement that something is a fact. Thus, assertions can be false, but facts cannot be false.

During forward chaining, whenever an *if* pattern is observed to match an assertion, the antecedent is **satisfied**. Whenever all the *if* patterns of a rule are satisfied, the rule is **triggered**. Whenever a triggered rule establishes a new assertion or performs an action, it is **fired**.

In deduction systems, all triggered rules generally fire. In reaction systems, however, when more than one rule is triggered at the same time, usually only one of the possible actions is desired, thus creating a need for some sort of *conflict-resolution procedure* to decide which rule should fire.

A Toy Deduction System Identifies Animals

Suppose that Robbie, a robot, wants to spend a day at the zoo. Robbie can perceive basic features, such as color and size, and whether an animal has hair or gives milk, but his ability to identify objects using those features is limited. He can distinguish animals from other objects, but he cannot use the fact that a particular animal has a long neck to conclude that he is looking at a giraffe.

Plainly, Robbie will enjoy the visit more if he can identify the individual animals. Accordingly, Robbie decides to build ZOOKEEPER, an identification-oriented deduction system.

Robbie could build ZOOKEEPER by creating one if–then rule for each kind of animal in the zoo. The consequent side of each rule would be a simple assertion of animal identity, and the antecedent side would be a bulbous enumeration of characteristics sufficiently complete to reject all incorrect identifications.

Robbie decides, however, to build ZOOKEEPER by creating rules that produce intermediate assertions. The advantage is that the antecedent–consequent rules involved need have only a few antecedents, making them easier for Robbie to create and use. Using this approach, ZOOKEEPER produces chains of conclusions leading to the identification of the animal that Robbie is currently examining.

Now suppose that Robbie's local zoo contains only seven animals: a cheetah, a tiger, a giraffe, a zebra, an ostrich, a penguin, and an albatross. This assumption simplifies ZOOKEEPER, because only a few rules are needed to distinguish one type of animal from another. One such rule, rule Z1, determines that a particular animal is a mammal:

```
Z1    If      ?x has hair
      then    ?x is a mammal
```

Note that antecedents and consequents are patterns that contain *variables*, such as x, marked by question-mark prefixes. Whenever a rule is considered, its variables have no values initially, but they acquire values as antecedent patterns are matched to assertions.

Suppose that a particular animal, named Stretch, has hair. Then, if the working memory contains the assertion *Stretch has hair*, the antecedent pattern, *?x has hair*, matches that assertion, and the value of x becomes *Stretch*. By convention, when variables become identified with values, they are said to be **bound** to those values and the values are sometimes called **bindings**. Thus, x is bound to *Stretch* and *Stretch* is x's binding.

Once a variable is bound, that variable is replaced by its binding wherever the variable appears in the same or subsequently processed patterns. Whenever the variables in a pattern are replaced by variable bindings, the pattern is said to be **instantiated**. For example, the consequent pattern, *?x is a mammal* becomes *Stretch is a mammal* once instantiated by the variable binding acquired when the antecedent pattern was matched.

Now let us look at other ZOOKEEPER rules. Three others also determine biological class:

Z2	If	*?x* gives milk
	then	*?x* is a mammal

Z3	If	*?x* has feathers
	then	*?x* is a bird

Z4	If	*?x* flies
		?x lays eggs
	then	*?x* is a bird

The last of these rules, Z4, has two antecedents. Although it does not really matter for the small collection of animals in ZOOKEEPER's world, some mammals fly and some reptiles lay eggs, but no mammal or reptile does both.

Once ZOOKEEPER knows that an animal is a mammal, two rules determine whether that animal is carnivorous. The simpler rule has to do with catching the animal in the act of having its dinner:

Z5	If	*?x* is a mammal
		?x eats meat
	then	*?x* is a carnivore

If Robbie is not at the zoo at feeding time, various other factors, if available, provide conclusive evidence:

Z6	If	*?x* is a mammal
		?x has pointed teeth
		?x has claws
		?x has forward-pointing eyes
	then	*?x* is a carnivore

All hooved animals are ungulates:

Z7	If	?*x* is a mammal
		?*x* has hoofs
	then	?*x* is an ungulate

If Robbie has a hard time looking at the feet, ZOOKEEPER may still have a chance because all animals that chew cud are also ungulates:

Z8	If	?*x* is a mammal
		?*x* chews cud
	then	?*x* is an ungulate

Now that Robbie has rules that divide mammals into carnivores and ungulates, it is time to add rules that identify specific animal identities. For carnivores, there are two possibilities:

Z9	If	?*x* is a carnivore
		?*x* has tawny color
		?*x* has dark spots
	then	?*x* is a cheetah

Z10	If	?*x* is a carnivore
		?*x* has tawny color
		?*x* has black strips
	then	?*x* is a tiger

Strictly speaking, the basic color is not useful because both of the carnivores are tawny. However, there is no need for information in rules to be minimal. Moreover, antecedents that are superfluous now may become essential later as new rules are added to deal with other animals.

For the ungulates, other rules separate the total group into two possibilities:

Z11	If	?*x* is an ungulate
		?*x* has long legs
		?*x* has long neck
		?*x* has tawny color
		?*x* has dark spots
	then	?*x* is a giraffe

Z12	If	?*x* is an ungulate
		?*x* has white color
		?*x* has black stripes
	then	?*x* is a zebra

Three more rules are needed to handle the birds:

Z13	If	?x is a bird
		?x does not fly
		?x has long legs
		?x has long neck
		?x is black and white
	then	?x is an ostrich

Z14	If	?x is a bird
		?x does not fly
		?x swims
		?x is black and white
	then	?x is a penguin

Z15	If	?x is a bird
		?x is a good flyer
	then	?x is an albatross

Now that you have seen all the rules in ZOOKEEPER, note that the animals evidently share many features. Zebras and tigers have black stripes; tigers, cheetahs, and giraffes have a tawny color; giraffes and ostriches have long legs and a long neck; and ostriches and penguins are black and white.

To learn about how forward chaining works, suppose that Robbie is at the zoo and is about to analyze an unknown animal, Stretch, using ZOOKEEPER. Further suppose that the following six assertions are in working memory:

Stretch has hair.
Stretch chews cud.
Stretch has long legs.
Stretch has a long neck.
Stretch has tawny color.
Stretch has dark spots.

Because Stretch has hair, rule Z1 fires, establishing that Stretch is a mammal. Because Stretch is a mammal and chews cud, rule Z8 establishes that Stretch is an ungulate.

At this point, all the antecedents for rule Z11 are satisfied. Evidently, Stretch is a giraffe.

Rule-Based Systems Use a Working Memory and a Rule Base

As you have seen in the ZOOKEEPER system, one of the key representations in a rule-based system is the working memory:

A **working memory** is a representation

In which

▷ Lexically, there are application-specific symbols and pattern symbols.

▷ Structurally, assertions are lists of application-specific symbols, and patterns are lists of application-specific symbols and pattern symbols.

▷ Semantically, the assertions denote facts in some world.

With constructors that

▷ Add an assertion to working memory

With readers that

▷ Produce a list of the matching assertions in working memory, given a pattern

---

Another key representation is the rule base:

A **rule base** is a representation

In which

▷ Lexically, there are application-specific symbols and pattern symbols.

▷ Structurally, patterns are lists of application-specific symbols and pattern symbols, and rules consist of patterns. Some of these patterns constitute the rule's *if* patterns; the others constitute the rule's *then* pattern.

▷ Semantically, rules denote constraints that enable procedures to seek new assertions or to validate a hypothesis.

With constructors that

▷ Construct a rule, given an ordered list of *if* patterns and a *then* pattern

With readers that

▷ Produce a list of a given rule's *if* patterns

▷ Produce a list of a given rule's *then* patterns

Thus, ZOOKEEPER uses instances of these representations that are specialized to animal identification. ZOOKEEPER itself can be expressed in procedural English, as follows:

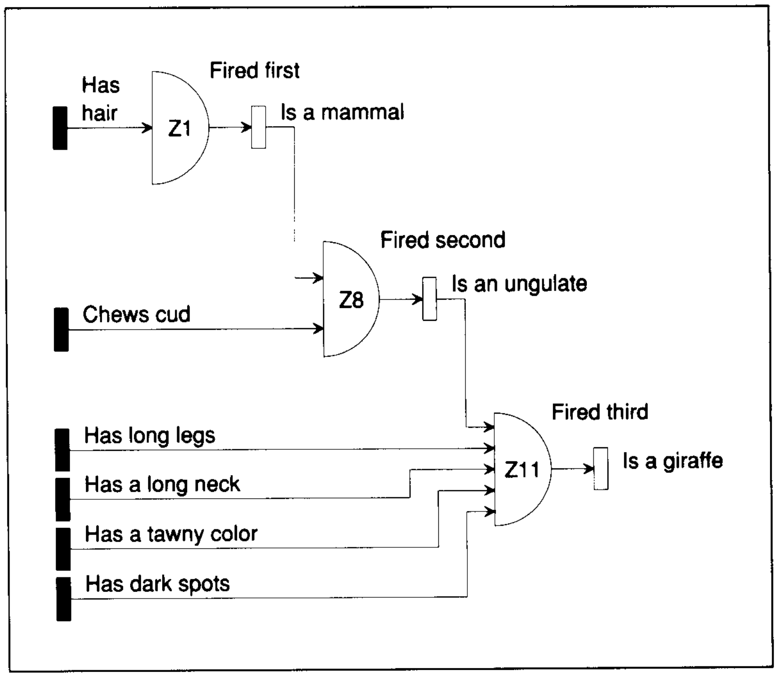


Figure 7.2 Knowing something about an unknown animal enables identification via forward chaining. Here, the assertions on the left lead to the conclusion that the unknown animal is a giraffe.

```
To identify an animal with ZOOKEEPER (forward-chaining version),
▷ Until no rule produces a new assertion or the animal is identified,
  ▷ For each rule,
    ▷ Try to support each of the rule's antecedents by matching it to known facts.
    ▷ If all the rule's antecedents are supported, assert the consequent unless there is an identical assertion already.
    ▷ Repeat for all matching and instantiation alternatives.
```

Thus, assertions flow through a series of antecedent–consequent rules from given assertions to conclusions, as shown in the history recorded in figure 7.2. In such diagrams, sometimes called **inference nets**, the D-shaped objects represent rules, whereas vertical bars denote given assertions and vertical boxes denote deduced assertions.

Deduction Systems May Run Either Forward or Backward

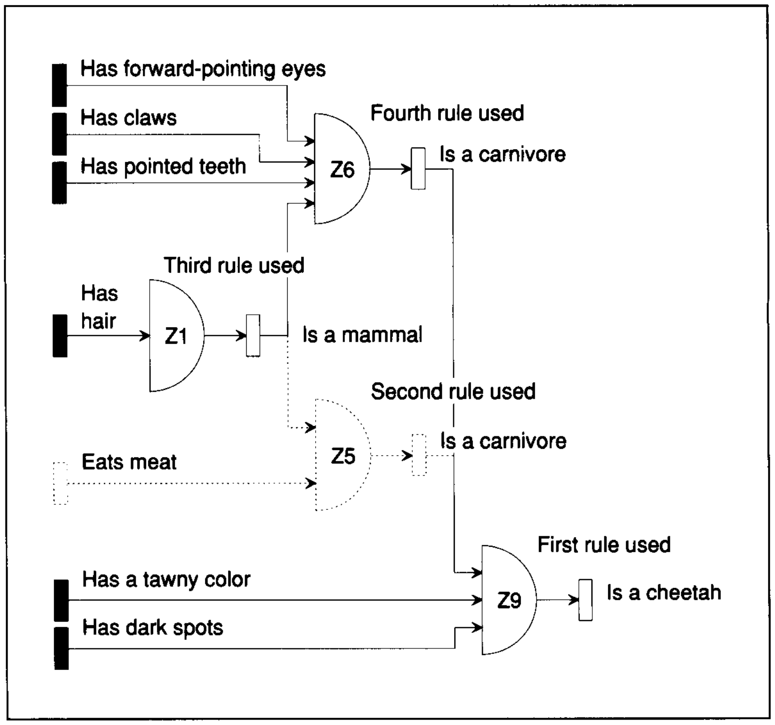
So far, you have learned about a deduction-oriented rule-based system that works from given assertions to new, deduced assertions. Running this way, a system exhibits forward chaining. *Backward chaining* is also possible: A rule-based system can form a hypothesis and use the antecedent–consequent rules to work backward toward hypothesis-supporting assertions.

For example, ZOOKEEPER might form the hypothesis that a given animal, Swifty, is a cheetah and then reason about whether that hypothesis is viable. Here is a scenario showing how things work out according to such a backward-chaining approach:

- ZOOKEEPER forms the hypothesis that Swifty is a cheetah. To verify the hypothesis, ZOOKEEPER considers rule Z9, which requires that Swifty is a carnivore, that Swifty has a tawny color, and that Swifty has dark spots.
- ZOOKEEPER must check whether Swifty is a carnivore. Two rules may do the job, namely rule Z5 and rule Z6. Assume that ZOOKEEPER tries rule Z5 first.
- ZOOKEEPER must check whether Swifty is a mammal. Again, there are two possibilities, rule Z1 and rule Z2. Assume that ZOOKEEPER tries rule Z1 first. According to that rule, Swifty is a mammal if Swifty has hair.
- ZOOKEEPER must check whether Swifty has hair. Assume ZOOKEEPER already knows that Swifty has hair. So Swifty must be a mammal, and ZOOKEEPER can go back to working on rule Z5.
- ZOOKEEPER must check whether Swifty eats meat. Assume ZOOKEEPER cannot tell at the moment. ZOOKEEPER therefore must abandon rule Z5 and try to use rule Z6 to establish that Swifty is a carnivore.
- ZOOKEEPER must check whether Swifty is a mammal. Swifty is a mammal, because this was already established when trying to satisfy the antecedents in rule Z5.
- ZOOKEEPER must check whether Swifty has pointed teeth, has claws, and has forward-pointing eyes. Assume ZOOKEEPER knows that Swifty has all these features. Evidently, Swifty is a carnivore, so ZOOKEEPER can return to rule Z9, which started everything done so far.
- Now ZOOKEEPER must check whether Swifty has a tawny color and dark spots. Assume ZOOKEEPER knows that Swifty has both features. Rule Z9 thus supports the original hypothesis that Swifty is a cheetah, and ZOOKEEPER therefore concludes that Swifty is a cheetah.

Thus, ZOOKEEPER is able to work backward through the antecedent–consequent rules, using desired conclusions to decide for what assertions it should look. A backward-moving chain develops, as dictated by the following procedure:

Figure 7.3 Knowing something about an unknown animal enables identification via backward chaining. Here, the hypothesis that Swifty is a cheetah leads to assertions that support that hypothesis.



To identify an animal with ZOOKEEPER (backward-chaining version),

```
▷ Until all hypotheses have been tried and none have been
  supported or until the animal is identified,
    ▷ For each hypothesis,
        ▷ For each rule whose consequent matches the current
          hypothesis,
            ▷ Try to support each of the rule's antecedents by
              matching it to assertions in working memory or by
              backward chaining through another rule, creating
              new hypotheses. Be sure to check all matching and
              instantiation alternatives.
            ▷ If all the rule's antecedents are supported, announce
              success and conclude that the hypothesis is true.
```

In the example, backward chaining ends successfully, verifying the hypothesis, as shown in figure 7.3. The chaining ends unsuccessfully if any required antecedent assertions cannot be supported.

The Problem Determines Whether Chaining Should Be Forward or Backward

Many deduction-oriented antecedent–consequent rule systems can chain either forward or backward, but which direction is better? This subsection describes several rules of thumb that may help you to decide.

Most important, you want to think about how the rules relate facts to conclusions. Whenever the rules are such that a typical set of facts can lead to many conclusions, your rule system exhibits a high degree of **fan out**, and a high degree of fan out argues for backward chaining. On the other hand, whenever the rules are such that a typical hypothesis can lead to many questions, your rule system exhibits a high degree of **fan in**, and a high degree of fan in argues for forward chaining.

- If the facts that you have or may establish can lead to a large number of conclusions, but the number of ways to reach the particular conclusion in which you are interested is small, then there is more fan out than fan in, and you should use backward chaining.
- If the number of ways to reach the particular conclusion in which you are interested is large, but the number of conclusions that you are likely to reach using the facts is small, then there is more fan in than fan out, and you should use forward chaining.

Of course, in many situations, neither fan out nor fan in dominates, leading you to other considerations:

- If you have not yet gathered any facts, and you are interested in only whether one of many possible conclusions is true, use backward chaining.

Suppose, for example, that you do not care about the identity of an animal. All you care about is whether it is a carnivore. By backward chaining from the carnivore hypothesis, you ensure that all the facts you gather are properly focused. You may ask about the animal's teeth, but you will never ask about the animal's color.

- If you already have in hand all the facts you are ever going to get, and you want to know everything you can conclude from those facts, use forward chaining.

Suppose, for example, that you have had a fleeting glimpse of an animal that has subsequently disappeared. You want to know what you can deduce about the animal. If you were to backward chain, you would waste time pursuing hypotheses that lead back to questions you can no longer answer because the animal has disappeared. Accordingly, you are better off if you forward chain.

RULE-BASED REACTION SYSTEMS

In deduction systems, the *if* parts of some if–then rules specify combinations of assertions, and the *then* part specifies a new assertion to be deduced

Mycin Diagnoses Bacterial Infections of the Blood

ZOOKEEPER is based on MYCIN, a well-known rule-based deduction system. Historically, the MYCIN system played a major role in stimulating research interest in rule-based expert systems.

MYCIN's expertise lies in the domain of bacterial infections. Physicians usually must begin antibiotic treatment for patients who have bacterial infections without knowing exactly which organism is the culprit. There is no time to wait for definitive laboratory culture evidence, which accumulates too slowly. For the desperately sick, therapy must begin at once—not 2 days from at once. This requirement leaves two choices: the physician can either prescribe a broad-spectrum drug that covers all possibilities, or she can prescribe a better, disease-specific drug.

MYCIN helps the physician to prescribe disease-specific drugs. MYCIN informs itself about particular cases by requesting information from the physician about a patient's symptoms, general condition, history, and laboratory-test results that can be obtained easily and quickly. At each point, the question MYCIN asks is determined by MYCIN's current hypothesis and the answers to all previous questions. Thus, the questions start as though taken from a checklist, but the questions then vary as evidence builds. Here is an excerpt from a sample session:

```
>   What is the patient's name?
    John Doe.
>   Male or female?
    Male.
>   Age?
    He is 55.
>   Have you obtained positive cultures indicating general type?
    Yes.
>   What type of infection is it?
    Primary bacteremia.
>   When did symptoms first appear?
    May 5.
>   Let's call the most recent positive culture C1. From what site was C1 taken?
    From the blood.
>   When?
    May 9.
>   Let's call the first significant organism from this culture U1. Do you know the
    identity of U1?
    No.
>   Is U1 a rod or a coccus or something else?
    Rod.
>   What is the gram stain of U1?
    Gram-negative.
>   Have there been positive cultures from sites that are not associated with the infec-
    tions about which you are seeking advice?
    No.
>   Did U1 grow aerobically?
    Yes.

⋮
```

When it is satisfied that it has a reasonably good grasp of the situation, MYCIN announces its diagnosis:

> \> My recommendation will be based on the opinion that the identity of U1 may be
> 1. Pseudomonas-aeruginosa
> 2. Klebsiella-pneumoniae
> 3. E. coli
> 4. Bacteroides-fragilis
> 5. Enterobacter
> 6. Proteus-nonmirabilis.

After asking a few more questions about the patient, MYCIN announces its recommendation:

> \> On a scale of 0 to 4, where higher numbers indicate greater severity, how would you rate the patient's degree of sickness?
> 3.
>
> \> Does the patient have a clinically significant allergic reaction to any antimicrobial agent?
> No.
>
> \> How much does the patient weigh?
> 70 kilograms.
>
> \> My first recommendation is as follows: to cover for items 1, 2, 3, 5, and 6, give gentamycin using a dose of 119 mg (1.7 mg/kg) q8h IV [or IM] for 10 days. Modify dose in renal failure. Also, to cover for item 4, give clindamycin using a dose of 595 mg (8.5 mg/kg) q6h IV [or IM] for 14 days. If diarrhea or other gastrointestinal symptoms develop, check for pseudomembranous colitis.

MYCIN's pool of knowledge consists of approximately 500 antecedent–consequent rules, which give MYCIN the ability to recognize about 100 causes of bacterial infections. The following rule is typical:

```
M88   If     ?x's type is primary bacteremia
             the suspected entry point of ?x is the gastrointestinal tract
             the site of the culture of ?x is one of the sterile sites
      then   there is evidence that ?x is bacteroides
```

MYCIN is a backward-chaining system, because physicians prefer to think about one hypothesis at a time. By sticking with the questions that are relevant to a particular hypothetical conclusion, the questioning is guaranteed to remain relevant to that hypothesis. A forward-running system can jump around, working first toward one conclusion and then toward another, seemingly at random.

Another reason why MYCIN was designed to be a backward-chaining system is that backward chaining simplifies the creation of an English-language interface. The interface needs to deal only with answers to specific questions, rather than with free-form, imaginative text.

directly from the triggering combination. In reaction systems, which are introduced in this section, the *if* parts specify the *conditions* that have to be satisfied and the *then* part specifies an *action* to be undertaken. Sometimes, the action is to *add* a new assertion; sometimes it is to *delete* an existing assertion; sometimes, it is to execute some procedure that does not involve assertions at all.

A Toy Reaction System Bags Groceries

Suppose that Robbie has just been hired to bag groceries in a grocery store. Because he knows little about bagging groceries, he approaches his new job by creating BAGGER, a rule-based reaction system that decides where each item should go.

After a little study, Robbie decides that BAGGER should be designed to take four steps:

1 The check-order step: BAGGER analyzes what the customer has selected, looking over the groceries to see whether any items are missing, with a view toward suggesting additions to the customer.
2 The bag-large-items step: BAGGER bags the large items, taking care to put the big bottles in first.
3 The bag-medium-items step: BAGGER bags the medium items, taking care to put frozen ones in freezer bags.
4 The bag-small-items step: BAGGER bags the small items.

Now let us see how this knowledge can be captured in a rule-based reaction system. First, BAGGER needs a working memory. The working memory must contain assertions that capture information about the items to be bagged. Suppose that those items are the items listed in the following table:

Item	Container type	Size	Frozen?
Bread	plastic bag	medium	no
Glop	jar	small	no
Granola	cardboard box	large	no
Ice cream	cardboard carton	medium	yes
Potato chips	plastic bag	medium	no
Pepsi	bottle	large	no

Next, BAGGER needs to know which step is the current step, which bag is the current bag, and which items already have been placed in bags. In the following example, the first assertion identifies the current step as the check-order step, the second identifies the bag as Bag1, and the remainder indicate what items are yet to be bagged:

Step is check-order
Bag1 is a bag
Bread is to be bagged
Glop is to be bagged
Granola is to be bagged
Ice cream is to be bagged
Potato chips are to be bagged

Note that working memory contains an assertion that identifies the step. Each of the rules in BAGGER's rule base tests the step name. Rule B1, for example, is triggered only when the step is the check-order step:

```
B1    If      step is check-order
              potato chips are to be bagged
              there is no Pepsi to be bagged
      then    ask the customer whether he would like a bottle of Pepsi
```

The purpose of rule B1 is to be sure the customer has something to drink to go along with potato chips, because potato chips are dry and salty. Note that rule B1's final condition checks that a particular pattern *does not* match any assertion in working memory.

Now let us move on to a rule that moves BAGGER from the check-order step to the bag-large-items step:

```
B2    If      step is check-order
      then    step is no longer check-order
              step is bag-large-items
```

Note that the first of rule B2's actions deletes an assertion from working memory. Deduction systems are assumed to deal with static worlds in which nothing that is shown to be true can ever become false. Reaction systems, however, are allowed more freedom. Sometimes, that extra freedom is reflected in the rule syntax through the breakup of the action part of the rule, marked by *then*, into two constituent parts, marked by *delete* and *add*. When you use this alternate syntax, rule B2 looks like this:

```
B2 (add–delete form)
   If       step is check-order
   delete   step is check-order
   add      step is bag-large-items
```

The remainder of BAGGER's rules are expressed in this more transparent **add–delete syntax**.

At first, rule B2 may seem dangerous, for it looks as though it could prevent rule B1 from doing its legitimate and necessary work. There is no problem, however. Whenever you are working with a reaction system, you adopt a suitable *conflict-resolution procedure* to determine which rule to fire among many that may be triggered. Bagger uses the simplest conflict-resolution strategy, *rule ordering*, which means that the rules are arranged in a list, and the first rule triggered is the one that is allowed to fire. By placing rule B2 after rule B1, you ensure that rule B1 does its job before rule B2 changes the step to bag-large-items. Thus, rule B2 changes the step only when nothing else can be done.

Use of the rule-ordering conflict resolution helps you out in other ways as well. Consider, for example, the first two rules for bagging large items:

```
B3    If        step is bag-large-items
                a large item is to be bagged
                the large item is a bottle
                the current bag contains < 6 large items
      delete    the large item is to be bagged
      add       the large item is in the current bag

B4    If        step is bag-large-items
                a large item is to be bagged
                the current bag contains < 6 large items
      delete    the large item is to be bagged
      add       the large item is in the current bag
```

Big items go into bags that do not have too many items already, but the bottles—being heavy—go in first. The placement of rule B3 before rule B4 ensures this ordering.

Note that rules B3 and B4 contain a condition that requires counting, so Bagger must do more than assertion matching when looking for triggered rules. Most rule-based systems focus on assertion matching, but provide an escape hatch to a general-purpose programming language when you need to do more than just match an antecedent pattern to assertions in working memory.

Evidently, Bagger is to add large items only when the current bag contains fewer than six items.[†] When the current bag contains six or more items, Bagger uses rule B5 to change bags:

```
B5    If        step is bag-large-items
                a large item is to be bagged
                an empty bag is available
      delete    the current bag is the current bag
      add       the empty bag is the current bag
```

Finally, another step-changing rule moves Bagger to the next step:

[†]Perhaps a better Bagger system would use volume to determine when bags are full; to deal with volume, however, would require general-purpose computation that would make the example unnecessarily complicated, albeit more realistic.

B6	If	step is bag-large-items
	delete	step is bag-large-items
	add	step is bag-medium-items

Let us simulate the result of using these rules on the given database. As we start, the step is check-order. The order to be checked contains potato chips, but no Pepsi. Accordingly, rule B1 fires, suggesting to the customer that perhaps a bottle of Pepsi would be nice. Let us assume that the customer goes along with the suggestion and fetches a bottle of Pepsi.

Inasmuch as there are no more check-order rules that can fire, other than rule B2, the one that changes the step to bag-large-items, the step becomes bag-large-items.

Now, because the Pepsi is a large item in a bottle, the conditions for rule B3 are satisfied, so rule B3 puts the Pepsi in the current bag. Once the Pepsi is in the current bag, the only other large item is the box of granola, which satisfies the conditions of rule B4, so it is bagged as well, leaving the working memory in the following condition:

Step is bag-medium-items
Bag1 contains Pepsi
Bag1 contains granola
Bread is to be bagged
Glop is to be bagged
Ice cream is to be bagged
Potato chips are to be bagged

Now it is time to look at rules for bagging medium items.

B7	If	step is bag-medium-items
		a medium item is frozen, but not in a freezer bag
	delete	the medium item is not in a freezer bag
	add	the medium item is in a freezer bag
B8	If	step is bag-medium-items
		a medium item is to be bagged
		the current bag is empty or contains only medium items
		the current bag contains no large items
		the current bag contains < 12 medium items
	delete	the medium item is to be bagged
	add	the medium item is in the current bag
B9	If	step is bag-medium-items
		a medium item is to be bagged
		an empty bag is available
	delete	the current bag is the current bag
	add	the empty bag is the current bag

Note that the fourth condition that appears in rule B8 prevents Bagger from putting medium items in a bag that already contains a large item. If there is a bag that contains a large item, rule B9 starts a new bag.

Also note that rule B7 and rule B8 make use of the rule-ordering conflict-resolution procedure. If both rule B7 and rule B8 are triggered, rule B7 is the one that fires, ensuring that frozen things are placed in freezer bags before bagging.

Finally, when there are no more medium items to be bagged, neither rule B7 nor rule B8 is triggered; instead, rule B10 is triggered and fires, changing the step to bag-small-items:

B10	If	step is bag-medium-items
	delete	step is bag-medium-items
	add	step is bag-small-items

At this point, after execution of all appropriate bag-medium-item rules, the situation is as follows:

Step is bag-small-items
Bag1 contains Pepsi
Bag1 contains granola
Bag2 contains bread
Bag2 contains ice cream (in freezer bag)
Bag2 contains potato chips
Glop is to be bagged

Note that, according to simple rules used by Bagger, medium items do not go into bags with large items. Similarly, conditions in rule B11 ensure that small items go in their own bag:

B11	If	step is bag-small-items
		a small item is to be bagged
		the current bag contains no large items
		the current bag contains no medium items
		the bag contains < 18 small items
	delete	the small item is to be bagged
	add	the small item is in the current bag

Bagger needs a rule that starts a new bag:

B12	If	step is bag-small-items
		a small item is to be bagged
		an empty bag is available
	delete	the current bag is the current bag
	add	the empty bag is the current bag

Finally, Bagger needs a rule that detects when bagging is complete:

```
B13   If       step is bag-small-items
      delete   step is bag-small-items
      add      step is done
```

After all rules have been used, everything is bagged:

Step is done
Bag1 contains Pepsi
Bag1 contains granola
Bag2 contains bread
Bag2 contains ice cream (in freezer bag)
Bag2 contains potato chips
Bag3 contains glop

Reaction Systems Require Conflict Resolution Strategies

Forward-chaining deduction systems do not need strategies for conflict resolution because every rule presumably produces reasonable assertions, so there is no harm in firing all triggered rules. But in reaction systems, when more than one rule is triggered, you generally want to perform only one of the possible actions, thus requiring a **conflict-resolution strategy** to decide which rule actually fires. So far, you have learned about rule ordering:

- *Rule ordering.* Arrange all rules in one long prioritized list. Use the triggered rule that has the highest priority. Ignore the others.

Here are other possibilities:

- *Context limiting.* Reduce the likelihood of conflict by separating the rules into groups, only some of which are active at any time.
- *Specificity ordering.* Whenever the conditions of one triggered rule are a superset of the conditions of another triggered rule, use the superset rule on the ground that it deals with more specific situations.
- *Data ordering.* Arrange all possible assertions in one long prioritized list. Use the triggered rule that has the condition pattern that matches the highest priority assertion in the list.
- *Size ordering.* Use the triggered rule with the toughest requirements, where *toughest* means the longest list of conditions.
- *Recency ordering.* Use the least recently used rule.

Of course, the proper choice of a conflict resolution strategy for a reaction system depends on the situation, making it difficult or impossible to rely on a fixed conflict resolution strategy or combination of strategies. An alternative is to think about which rule to fire as another problem to be solved. An elegant example of such problem solving is described in Chapter 8 in the introduction of the SOAR problem solving architecture.

PROCEDURES FOR FORWARD AND BACKWARD CHAINING

In this section, you learn more about rule-based systems. The focus is on how to do forward and backward chaining using well-known methods for exploring alternative variable bindings.

Depth-First Search Can Supply Compatible Bindings for Forward Chaining

One simple way to do forward chaining is to cycle through the rules, looking for those that lead to new assertions once the consequents are instantiated with appropriate variable bindings:

```
To forward chain (coarse version),
▷ Until no rule produces a new assertion,
  ▷ For each rule,
    ▷ For each set of possible variable bindings determined
      by matching the antecedents to working memory,
      ▷ Instantiate the consequent.
      ▷ Determine whether the instantiated consequent is
        already asserted. If it is not, assert it.
```

For an example, let us turn from the zoo to the track, assuming the following assertions are in working memory:

Comet	is-a	horse
Prancer	is-a	horse
Comet	is-a-parent-of	Dasher
Comet	is-a-parent-of	Prancer
Prancer	is	fast
Dasher	is-a-parent-of	Thunder
Thunder	is	fast
Thunder	is-a	horse
Dasher	is-a	horse

Next, let us agree that a horse who is the parent of something fast is valuable. Translating this knowledge into an if–then rule produces the following:

```
Parent Rule
    If     ?x is-a horse
           ?x is-a-parent-of ?y
           ?y is fast
    then   ?x is valuable
```

APPLICATION

XCON Configures Computer Systems

BAGGER is based on XCON, a well-known rule-based deduction system. Historically, the XCON system played a major role in stimulating commercial interest in rule-based expert systems.

XCON's domain is computer-system components. When a company buys a big mainframe computer, it buys a central processor, memory, terminals, disk drives, tape drives, various peripheral controllers, and other paraphernalia. All these components must be arranged sensibly along input–output buses. Moreover, all the electronic modules must be placed in the proper kind of cabinet in a suitable slot of a suitable backplane.

Arranging all the components is a task called *configuration.* Doing configuration can be tedious, because a computer-component family may have hundreds of possible options that can be organized in an unthinkable number of combinations.

To do configuration, XCON uses rules such as the following:

X1	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 that the sbi module occupies is known
		there is space available for a power supply
		there is no available power supply
		the voltage and frequency of the components are known
	then	add an appropriate power supply
X2	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 the sbi module occupies is known
		there is space available for a power supply
		there is an available power supply
	then	put the power supply in the cabinet in the available space

The first rule, X1, acts rather like the one in BAGGER that asks the customer whether he wants a bottle of Pepsi if the order contains potato chips but no beverage. The second rule, X2, is a typical insertion rule. The context mentioned in both rules is a combination of the top-level step and a substep. The context is changed by rules such as the following:

X3	If	the current context is x
	then	deactivate the x context
		activate the y context

Rule X3 has the effect of deleting one item from the context designation and adding another. It fires only if no other rule associated with the context triggers.

XCON has nearly 10,000 rules and knows the properties of several hundred component types for VAX computers, made by Digital Equipment Corporation. XCON routinely handles orders involving 100 to 200 components. It is representative of many similar systems for marketing and manufacturing.

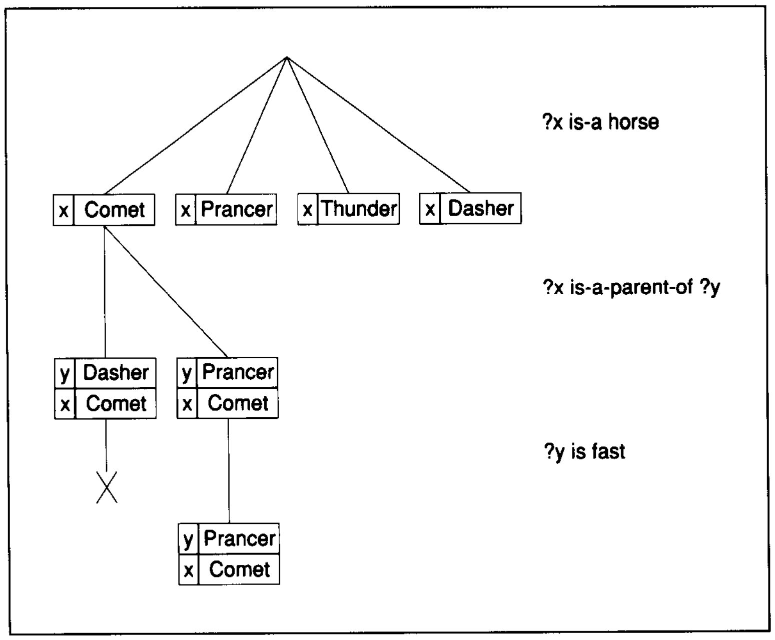


Figure 7.4 During forward chaining, binding commitments can be arranged in a tree, suggesting that ordinary search methods can be used to find one or all of the possible binding sets. Here, the parent rule's first antecedent leads to four possible bindings for *x*, and the rule's second antecedent, given that *x* is bound to Comet, leads to two possible bindings for *y*.

Now, if there is a binding for x and a binding for y such that each antecedent corresponds to an assertion when the variables are replaced by their bindings, then the rule justifies the conclusion that the thing bound to x is valuable. For example, if x is bound to Comet and y is bound to Prancer, then each of the antecedents corresponds to an assertion—namely *Comet is-a horse*, *Comet is-a-parent-of Prancer*, and *Prancer is fast*. Accordingly, Comet must be valuable.

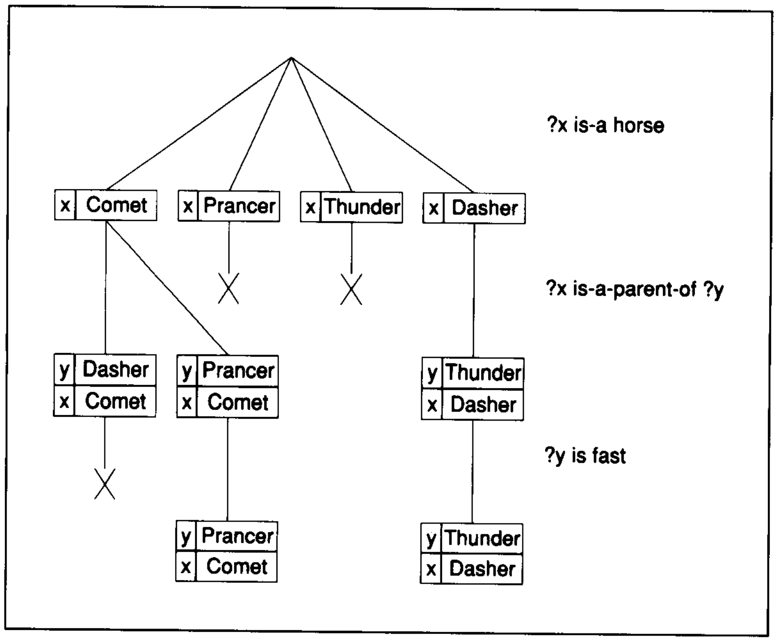
To conduct a search for binding pairs, you can start by matching the first antecedent against each assertion. As shown in figure 7.4, there are four matches and four corresponding binding choices for x in the antecedent *?x is-a horse*, because Comet, Prancer, Thunder, and Dasher are all horses.

Next, proceeding in the depth-first search style, assume that x's binding should be Comet, which is the binding produced by the first match. Then, with x bound to Comet, the second assertion, after instantiation, is *Comet is-a-parent-of ?y*. Matching this second instantiated antecedent against each assertion produces two matches, because Comet is a parent of both Dasher and Prancer. Thus, there are two binding choices for y given that x is bound to Comet.

Figure 7.4 show how the x and y choices fit together. Evidently, each of the two antecedents examined so far produces binding choices that can be arranged in a search tree.

Traveling along the leftmost branch, with x bound to Comet and y bound to Dasher, you proceed to the third antecedent, which becomes *Dasher is fast* when instantiated with y's binding. This instantiated an-

Figure 7.5 Two paths extend from the root down through the levels corresponding to three rule antecedents. Evidently, there are two binding sets that satisfy the rule.



tecedent fails to match any assertion, however, so you have to look farther for an acceptable combination. You do not have to look far, because the combination with x bound to Comet, as before, and y bound to Prancer leads to the instantiation of the third antecedent as *Prancer is fast*, which does match an assertion. Accordingly, you can conclude that the combination with x bound to Comet and y bound to Prancer is a combination that jumps over all the antecedent hurdles. You can use this combination to instantiate the consequent, producing *Comet is valuable.*

As shown in figure 7.4, there are three other choices for x bindings. Among these, if x is bound to Prancer or Thunder, then the second assertion, once instantiated, becomes *Prancer is-a-parent-of ?y* or *Thunder is-a-parent-of ?y*, both of which fail to match any assertion. If Dasher is the proper binding, then *Dasher is-a-parent-of ?y* matches just one assertion, *Dasher is-a-parent-of Thunder*, leaving only Thunder as a choice for y's binding. With x bound to Dasher and y bound to Thunder, the third instantiated antecedent is *Thunder is fast*, which matches an assertion, leading to the conclusion, as shown in figure 7.5, that the Dasher–Thunder combination also jumps over all the hurdles, suggesting that Dasher is valuable too.

From this example, several points of interest emerge. First, you can see that each path in the search tree corresponds to a set of binding commitments. Second, each antecedent matches zero or more assertions given the bindings already accumulated along a path, and each successful match produces a branch. Third, the depth of the search tree is always equal to the number of antecedents. Fourth, you have a choice, as usual, about how you search the tree. Exhaustive, depth-first, left-to-right search is the usual method when the objective is to find all possible ways that a rule can be deployed. This method is the one exhibited in the following procedure:

```
To forward chain (detailed version),
▷ Until no rule produces a new assertion,
  ▷ For each rule,
    ▷ Try to match the first antecedent with an existing asser-
      tion. Create a new binding set with variable bindings
      established by the match.
    ▷ Using the existing variable bindings, try to match the
      next antecedent with an existing assertion. If any new
      variables appear in this antecedent, augment the exist-
      ing variable bindings.
    ▷ Repeat the previous step for each antecedent, accumu-
      lating variable bindings as you go, until,
      ▷ There is no match with any existing assertion using
        the binding set established so far. In this case, back
        up to a previous match of an antecedent to an asser-
        tion, looking for an alternative match that produces
        an alternative, workable binding set.
      ▷ There are no more antecedents to be matched. In this
        case,
        ▷ Use the binding set in hand to instantiate the con-
          sequent.
        ▷ Determine if the instantiated consequent is already
          asserted. If not, assert it.
        ▷ Back up to the most recent match with unexplored
          bindings, looking for an alternative match that pro-
          duces a workable binding set.
      ▷ There are no more alternatives matches to be explored
        at any level.
```

Depth-First Search Can Supply Compatible Bindings for Backward Chaining

You learned that forward chaining can be viewed as searching for variable-binding sets such that, for each set, all antecedents correspond to assertions once their variables are replaced by bindings from the set.

Backward chaining can be treated in the same general way, but there are a few important differences and complications. In particular, you start by matching a hypothesis both against existing assertions and against rule consequents.

Suppose, for example, that you are still working with horses using the same rules and assertions in working memory as before. Next, suppose that you want to show that Comet is valuable; in other words, suppose that you want to verify the hypothesis, *Comet is valuable.* You fail to find a match for *Comet is valuable* among the assertions, but you succeed in matching the hypothesis with the rule consequent, *?x is valuable.* The success leads you to attempt to match the antecedents, presuming that x is bound to Comet.

Happily, the instantiated first antecedent, *Comet is-a horse*, matches an assertion, enabling a search for the instantiated second antecedent, *Comet is-a-parent-of y.* This second antecedent leads to two matches, one with the assertion *Comet is-a-parent-of Dasher* and one with the assertion *Comet is-a-parent-of Prancer.* Accordingly, the search branches, as shown in figure 7.6.

Along the left branch, y is bound to Dasher, leading to a futile attempt to match the third antecedent *Dasher is fast* to an assertion. Along the right branch, however, y is bound to Prancer, leading to a successful attempt to match *Prancer is fast* to an assertion.

Evidently, the hypothesis, *Comet is valuable*, is supported by the combination of the given rule and the given assertions because a binding set, discovered by search, connects the hypothesis with the assertions via the rule.

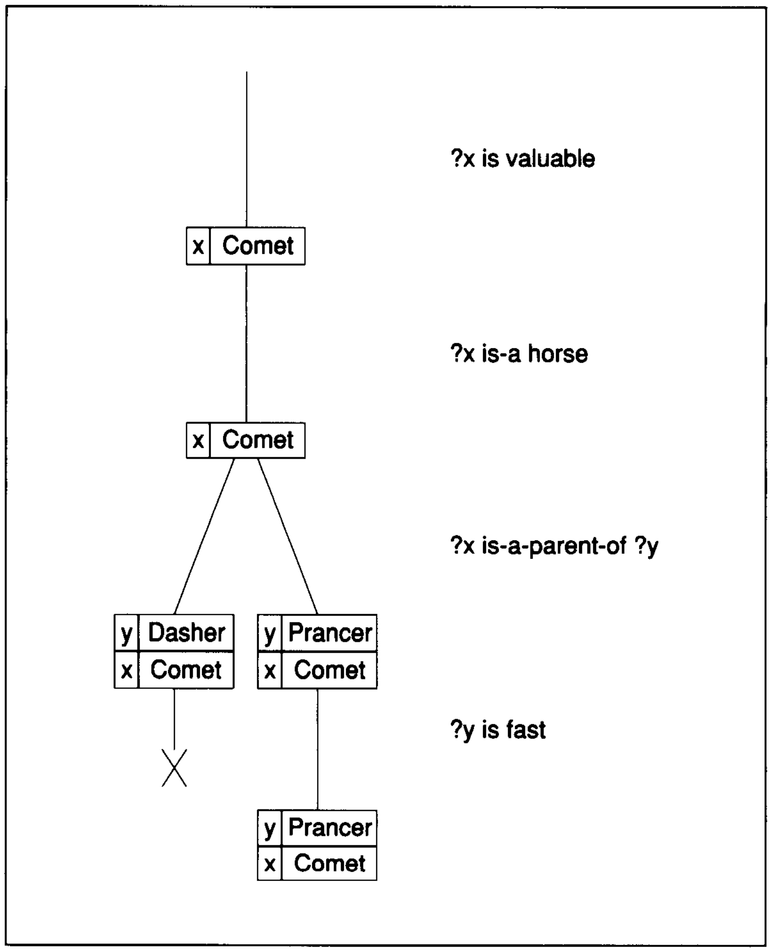
The search is more complicated, however, when the hypothesis itself contains a variable. Suppose that the question is "Who is valuable?" rather than "Is Comet valuable?" Then, the hypothesis itself, *?z is valuable*, contains a variable, z.

This new hypothesis, like the hypothesis *Comet is valuable*, matches no assertions but does match the consequent, *?x is valuable.* Now, however, you have a match between two variables, z and x, instead of a constant, Comet, and a variable, x.

Accordingly, now that it is time to match the first antecedent with the assertions, you go into the match with z bound to x. The variable x is not bound to anything, however, so the match of the first antecedent proceeds unfettered, as though the chaining were forward. There are four possible matches of the first antecedent to assertions, with x bound to any one of Comet, Prancer, Thunder, or Dasher. Then, assuming x's binding should be Comet, and working through the bindings allowed by the next two assertions, you are led to one of the results shown in figure 7.7, with z bound to x, x bound to Comet, and y bound to Prancer.

The fact that z, the variable in the hypothesis, matches x, a variable in a rule, need cause you no serious pause. The only additional task you need to perform is to instantiate all the way to constants whenever you have an

Figure 7.6 During backward chaining, as during forward chaining, binding commitments can be arranged in a tree, but the first binding commitments are established by the consequent, rather than by the first antecedent. Here, the consequent establishes one binding for *x*, and the second antecedent establishes two bindings for *y*. The binding for *x* and one of the two bindings for *y* establish that Comet is valuable.



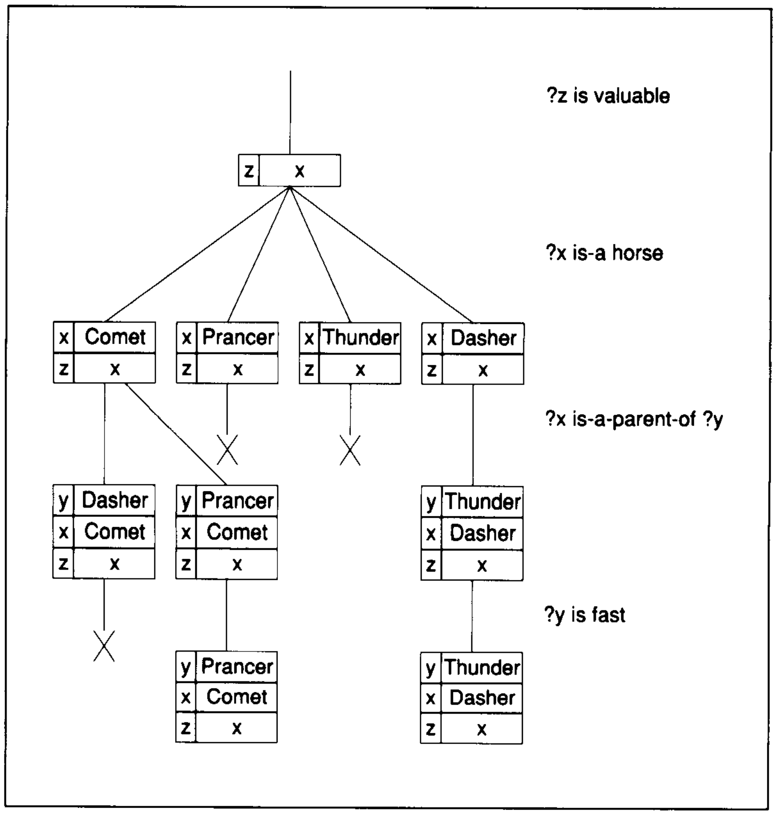
option to continue instantiating. Thus, you first replace *z* by *x*, and then you replace *x* by Comet, producing an instantiated hypothesis of *Comet is valuable.*

At this point, you could, if you wished, continue to look for other ways to bind variables so as to find other valuable horses.

The search is still more complicated when more than one rule can provide a variable binding. Suppose, for example, that you have the hypothesis with the variable, *?z is valuable*, but that you now add a new rule and two new assertions:

Winner Rule
    If   *?w* is-a winner
    then *?w* is fast

Figure 7.7 During backward chaining, hypothesis variables can be bound not only to assertion constants but also to consequent variables. Here, the hypothesis variable *z* is bound to the consequent variable *x*. Ultimately, two binding sets are found, establishing that two horses are valuable.



Dasher	Is-a	Winner
Prancer	Is-a	Winner

Now, the search proceeds as before, as shown in figure 7.8, until it is time to find a match for the third antecedent, *?y is fast*. The first time, with *y* bound to Dasher, there is no matching assertion, but there is a match between the second rule's consequent *w is fast* and the instantiated first-rule antecedent, *Dasher is fast*. Consequently, *w* becomes bound to Dasher, and an effort is made to find a match between the second rule's instantiated antecedent, *Dasher is-a winner*, against an assertion. Because there is a match with one of the two new assertions, you can conclude that Dasher is indeed fast, which means that the original hypothesis, *?z is valuable*, can be connected via rules to assertions using the binding set with *w* and *y* bound to Dasher, *x* bound to Comet, and *z* bound to *x*. To instantiate the hypothesis with this binding set, you first replace *z* with *x*, and then replace *x* with Comet.

Note that you can gain nothing by trying to find a second match for the instantiated antecedent, *Comet is-a-parent-of y*, because the ultimate conclusion, that Comet is valuable, has already been reached. Nevertheless,

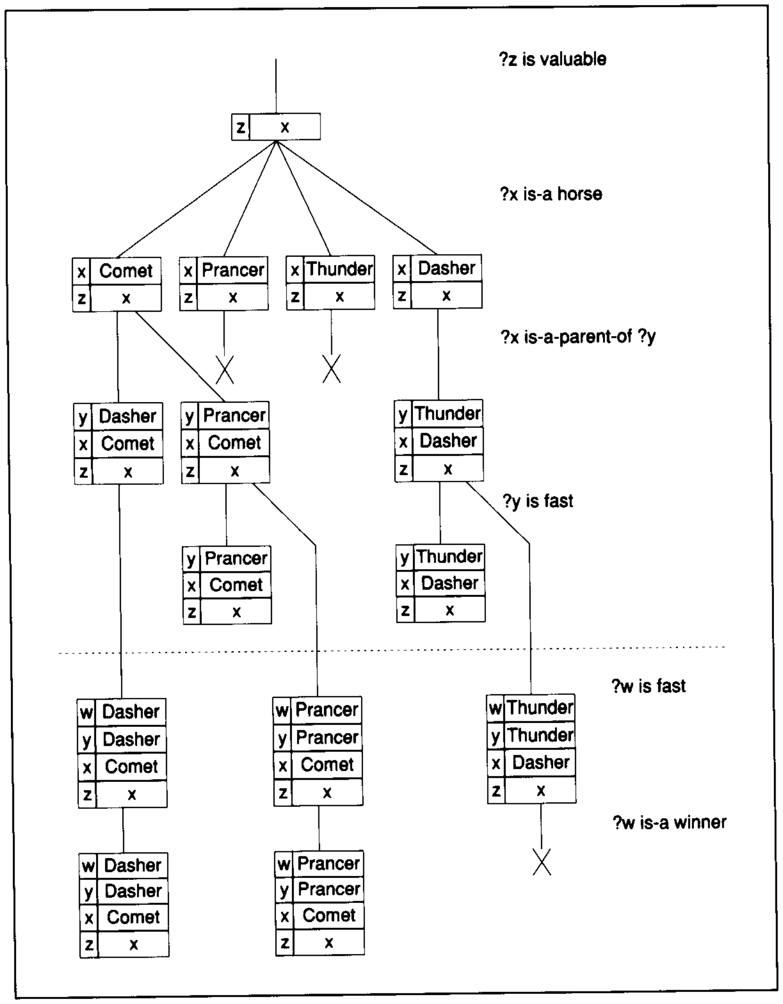


Figure 7.8 During backward chaining, rules chain together whenever the antecedent of one matches the consequent of another. Here, an antecedent, *?y is fast*, in one rule matches a consequent, *?w is fast* in another rule.

most binding-set programs are not smart enough to realize that nothing is to be gained, so they look for other ways to bind *y* using *Comet is-a-parent-of y*.

As shown in figure 7.8, *y* can be bound to Prancer, which leads to an attempt to match the third antecedent, *Prancer is fast* with an assertion; the match succeeds, reaffirming, with a different binding set, that Comet is valuable.

Displaying even more energy, most binding programs not only note that the instantiated antecedent, *Prancer is fast*, is in the database; they also note, as shown in figure 7.8, that there is a rule that links *Prancer is fast* to the assertion *Prancer is-a winner*, thus reaffirming, for a third time, with a binding set that includes a binding for *w*, that Comet is valuable.

Similarly, when searching for evidence that Dasher is valuable, the instantiated antecedent, *Thunder is fast*, is not only matched to an assertion, it is matched to the consequent, *w is fast*, in the second rule. This time, however, the instantiated antecedent in the second rule, *w is-a winner*, does not match any assertion, so there remains just one way of showing that Dasher is valuable.

In summary, the backward-chaining procedure moves from the initial hypothesis, through rules, to known facts, establishing variable bindings in the process. When the initial hypothesis matches the consequent of a rule, you create a binding set. Additional bindings are added to the initial binding set as the backward-chaining procedure works on the antecedents, and still more bindings are added when the procedure chains through an antecedent to the consequent of another rule. The following procedure summarizes:

To backward chain,

- ▷ Find a rule whose consequent matches the hypothesis (or antecedent) and create a binding set (or augment the existing binding set).
- ▷ Using the existing binding set, look for a way to deal with the first antecedent,
  - ▷ Try to match the antecedent with an existing assertion.
  - ▷ Treat the antecedent as an hypotheses and try to support it by backward chaining through other rules using the existing binding set.
- ▷ Repeat the previous step for each antecedent, accumulating variable bindings, until,
  - ▷ There is no match with any existing assertion or rule consequent using the binding set established so far. In this case, back up to the most recent match with unexplored bindings, looking for an alternative match that produces a workable binding set.
  - ▷ There are no more antecedents to be matched. In this case, the binding set in hand supports the original hypothesis.
    - ▷ If all possible binding sets are desired, report the current binding set, and back up, as if there were no match.
    - ▷ If only one possible binding set is desired, report the current binding set and quit.
  - ▷ There are no more alternative matches to be explored at any level.

Relational Operations Support Forward Chaining

Now it is time to look at another approach to forward chaining. First, you learn how relational database operations can handle the bookkeeping required for forward chaining. Then, you learn how the relational database operations can be arranged to produce high-speed operation.

All that you need to know about relational databases in this section is introduced as you need it. If you have not studied relational databases elsewhere, and find the introduction in this section to be too brief, read the appendix, which describes relational databases in more detail.

Now consider the Parent Rule and assertions previously used to demonstrate the search-oriented approach. Here, again, is the Parent Rule.

```
Parent Rule
   If      ?x is-a horse
           ?x is-a-parent-of ?y
           ?y is fast
   then    ?x is valuable
```

Now think of the assertions as though they were part of a table. In the language of relations, the assertions are recorded in a **relation**, named Data, whose columns are labeled with **field names**—namely First, Second, and Third:

First	Second	Third
Comet	is-a	horse
Prancer	is-a	horse
Comet	is-a-parent-of	Dasher
Comet	is-a-parent-of	Prancer
Prancer	is	fast
Dasher	is-a-parent-of	Thunder
Thunder	is	fast
Thunder	is-a	horse
Dasher	is-a	horse

To determine what values of x and y trigger the rule, you first determine which of the relation's records match the first antecedent in the rule, *?x is-a horse*. In the language of relations, you need to find those records whose Second field value is *is-a* and whose Third field value is *horse*. Conveniently, relational database systems include an access procedure, SELECT, that extracts records with specified field values from one relation to produce a new relation with fewer records. You can ask SELECT to pick the horses out of the Data relation, for example:

SELECT Data with Second = is-a and Third = horse

The result is the new relation:

First	Second	Third
Comet	is-a	horse
Prancer	is-a	horse
Thunder	is-a	horse
Dasher	is-a	horse

All you really want to know, however, is which bindings of x produce matches. Accordingly, you use another relational database access procedure, PROJECT, to isolate the appropriate field:

PROJECT Result over First

At this point, the field named First is renamed X to remind you that it consists of bindings for the x variable. The result, a single-field relation, is as follows:

A1

X
Comet
Prancer
Thunder
Dasher

Next, you determine which of the records in the data relation match the second antecedent in the rule, *?x is-a-parent-of ?y*. You need to select those records whose Second field value is *is-a-parent-of*. Then you project the results over the First and Third fields:

PROJECT [SELECT Data with Second = is-a-parent-of]
over First and Third

After renaming the field named First to X and the field named Third to Y, you have the following table:

A2

X	Y
Comet	Dasher
Comet	Prancer
Dasher	Thunder

Finally, you need to determine which of the records in the data relation match the third antecedent in the rule, *?y is fast*. Accordingly, you select those records whose Second field value is *is* and whose Third field value is *fast*, and you project the result over the First field:

PROJECT [SELECT Data with Second = is and Third = fast]
over First

After renaming the field named First to Y, reflecting the fact that the field values are possible bindings for y, you have the following table:

A3

Y
Prancer
Thunder

You now have three new relations–A1, A2, and A3—corresponding to the three antecedents in the rule. The next question is, What bindings of x satisfy both the first and second antecedents? Or, What field values are found both in A1's X field and in A2's X field?

The JOIN operation builds a relation with records constructed by concatenating records, one from each of two source tables, such that the records match in prescribed fields. Thus, you can join A1 and A2 over their X fields to determine which values of x are shared. Here is the required JOIN operation:

JOIN A1 and A2 with X = X

The result is a relation in which field-name ambiguities are eliminated by concatenation of ambiguous field names with the names of the relations that contribute them:

B1 (preliminary)

X.A1	X.A2	Y
Comet	Comet	Dasher
Comet	Comet	Prancer
Dasher	Dasher	Thunder

All you really want, of course, is to find the pairs of bindings for x and y that satisfy the first two antecedents. Accordingly, you can project the preliminary B1 relation over, say, X.A1 and Y, with the following result, after renaming of the fields:

B1

X	Y
Comet	Dasher
Comet	Prancer
Dasher	Thunder

B1 now contains binding pairs that simultaneously satisfy the first two antecedents in the rule. Now you can repeat the analysis to see which of these binding pairs also satisfy the third antecedent.

To begin, you join A3 and B1 over their Y fields to determine which values of y are shared:

JOIN A3 and B1 with Y = Y

The result is as follows:

B2 (preliminary)

Y.A3	X	Y.B1
Prancer	Comet	Prancer
Thunder	Dasher	Thunder

Now you project to determine the pairs of bindings for x and y that satisfy not only the first two antecedents, but also the third:

B2

X	Y
Comet	Prancer
Dasher	Thunder

At this point, you know that there are two binding pairs that simultaneously satisfy all three antecedents. Inasmuch as the *then* part of the rule uses only the binding of x, you project B2 over the X field:

B2

X
Comet
Dasher

Thus, the parent rule is triggered in two ways: once with x bound to Comet, and once with x bound to Dasher. In a deduction system, both binding sets can be used. In a reaction system, a conflict-resolution procedure would be required to select the next action.

The only problem with the procedure that you just learned about is that it consumes a large amount of computation. If a rule has n antecedents, then it takes n SELECT and n PROJECT operations to produce the A relations along with $n - 1$ JOIN and $n - 1$ PROJECT operations to produce the B relations. If there happen to be m rules, and if you check out each rule whenever a new assertion is added to the data relation, then you have to perform mn SELECTs, $m(2n - 1)$ PROJECTs, and most alarmingly, $m(n - 1)$ expensive JOINs each time a new assertion is added. Fortunately, there is another way to search for variable bindings that does not use so many operations.

The Rete Approach Deploys Relational Operations Incrementally

You have just learned how to use relational operations to find binding sets, but the method described is an expensive way to do forward chaining, because a great deal of work has to be done to trigger a rule. Now you are ready to learn that the relational operations can be performed incrementally, as each new assertion is made, reducing both the total amount of work and the time it takes to trigger a rule once all the triggering assertions are in place.

Ordinarily, the word *rete* is an obscure synonym for net, found only in large dictionaries. In the context of forward chaining, however, the word **rete procedure** names a procedure that works by moving each new assertion, viewed as a relational record, through a rete of boxes, each of which performs a relational operation on one relation or on a few, but never on all the relations representing accumulated assertions.

The arrangement of the rete for the valuable-horse example is shown in figure 7.9.

As a new assertion is made, it becomes a single-record relation. That single-record relation is then examined by a family of SELECT operations, each of which corresponds to a rule antecedent.

In the example, the first assertion is *Comet is-a horse.* Accordingly, the following single-record relation is constructed:

New-assertion

First	Second	Third
Comet	is-a	horse

This relation is examined by three SELECT operations:

SELECT new-assertion with Second = is-a and Third = horse
SELECT new-assertion with Second = is-a-parent-of
SELECT new-assertion with Second = is and Third = fast

Next, whenever the record in the single-record relation makes it past a SELECT operation, the single-record relation endures a PROJECT operation that picks off the field or fields that contain bindings.

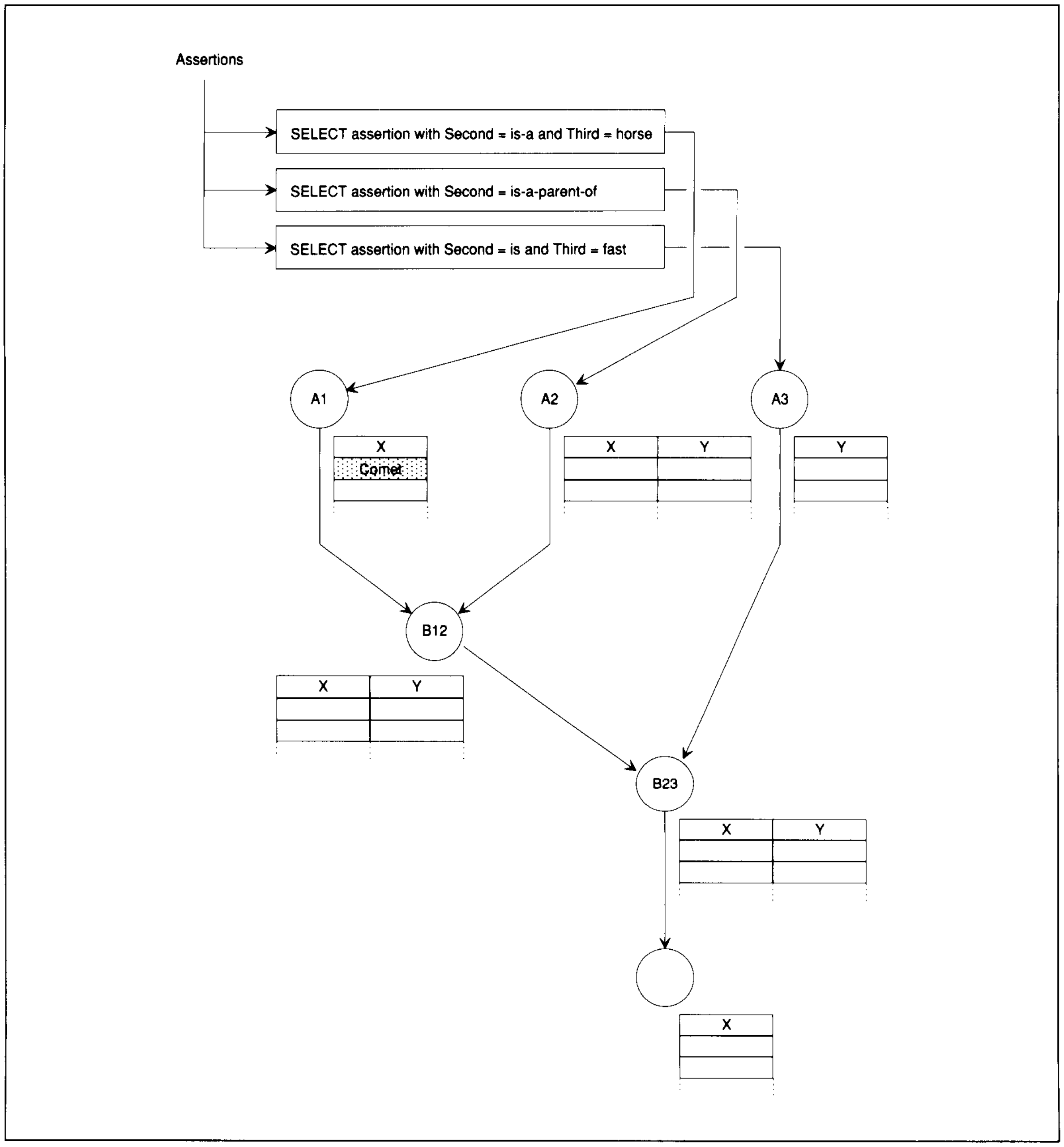

Assertions
SELECT assertion with Second = is-a and Third = horse
SELECT assertion with Second = is-a-parent-of
SELECT assertion with Second = is and Third = fast
A1
A2
A3
X
Comet
X Y
Y
B12
X Y
B23
X Y
X

Figure 7.9 The rete for a simple rule about horses with fast offspring. Here the state of the rete is captured just following the addition made in response to the first assertion. In this and other figures, the most recent changes are shown shaded.

In the example, the record makes it past the first of the three SELECT operations, whereupon the PROJECT and renaming operations produce the following:

New-assertion

X
Comet

Once a record has gone past the SELECT, PROJECT, and renaming operations, it is added to a relation associated with a rule antecedent. Each antecedent-specific relation is located at an **alpha node** created specifically for the antecedent. Each alpha-node relation accumulates all the assertions that make it through the corresponding, filterlike SELECT operation. In the example, the selected, projected, and renamed record is added to the A1 node—the one attached to the first antecedent.

The second assertion, *Prancer is-a horse*, follows the first through the rete, and also ends up as a record in the relation attached to the A1 node. Then, the third assertion, Comet is-a-parent-of Dasher, following a different route, ends up as a record in the relation attached to the A2 node.

Each addition to an alpha node's relation inspires an attempt to join the added record, viewed as a single-record relation, with another relation. In particular, an addition to either A1's relation or A2's relation leads to joining of the added record, viewed as a single-record relation, with the relation attached to the other alpha node. Importantly, the JOIN operation is done with a view toward determining whether the variable binding expressed in the added record corresponds to a variable binding already established in the other relation.

In the example, the added record—the one added to A2's relation—produces the following single-record relation:

X	Y
Comet	Dasher

Meanwhile, A1's relation has accumulated two records:

A1

X
Comet
Prancer

Joining the two relations over the X field and projecting to eliminate one of the redundant X fields yields a one-record, two-field relation:

X	Y
Comet	Dasher

This new relation is then added to a relation attached to a **beta node**—the B12 node—so named because it is the JOIN of the A1 and A2 relations. B12's relation contains a single record that records a pair of bindings for x and y that satisfies the first and second antecedents.

Thus, an addition to either A1's relation or A2's relation leads to a JOIN operation that may add one or more records to B12's relation reflecting variable bindings that satisfy the first two rule antecedents simultaneously.

The next assertion—the fourth—*Comet is-a-parent-of Prancer*, produces the wave of activity in the rete shown by the shading in figure 7.10.

The wave starts with the addition of a second record to A2's relation. This new record, viewed as a single-record relation, is joined to A1's relation, producing a second record for B12's relation.

Because it is tiresome to append the phrase, *viewed as a relation*, each time a record, viewed as a relation, is joined with another relation, let us agree to speak of joining records with a relation, even though, strictly speaking, only relations are joined.

Next, the fifth assertion, *Prancer is fast*, initiates a wave of additions to the records in the rete and leads to a record in A3's relation. In general, an addition to A3's relation leads to joining the added record with B12's relation. This JOIN operation is done with a view toward determining whether the variable binding expressed in the added record corresponds to a variable binding already established in B12's relation.

The result is added to B23's relation. In this example, the JOIN operation is over the Y fields, and the JOIN operation produces—after doing a PROJECT to eliminate one of the redundant Y fields—an initial record for B23's relation:

B23

X	Y
Comet	Prancer

Projecting this new record over the X field yields a possible binding for x in the rule's *then* part:

Parent-Rule Bindings B23

X
Comet

Thus, an addition to A3's relation has led to joining the added record with B12's relation. Symmetrically, of course, any new records added to B12's

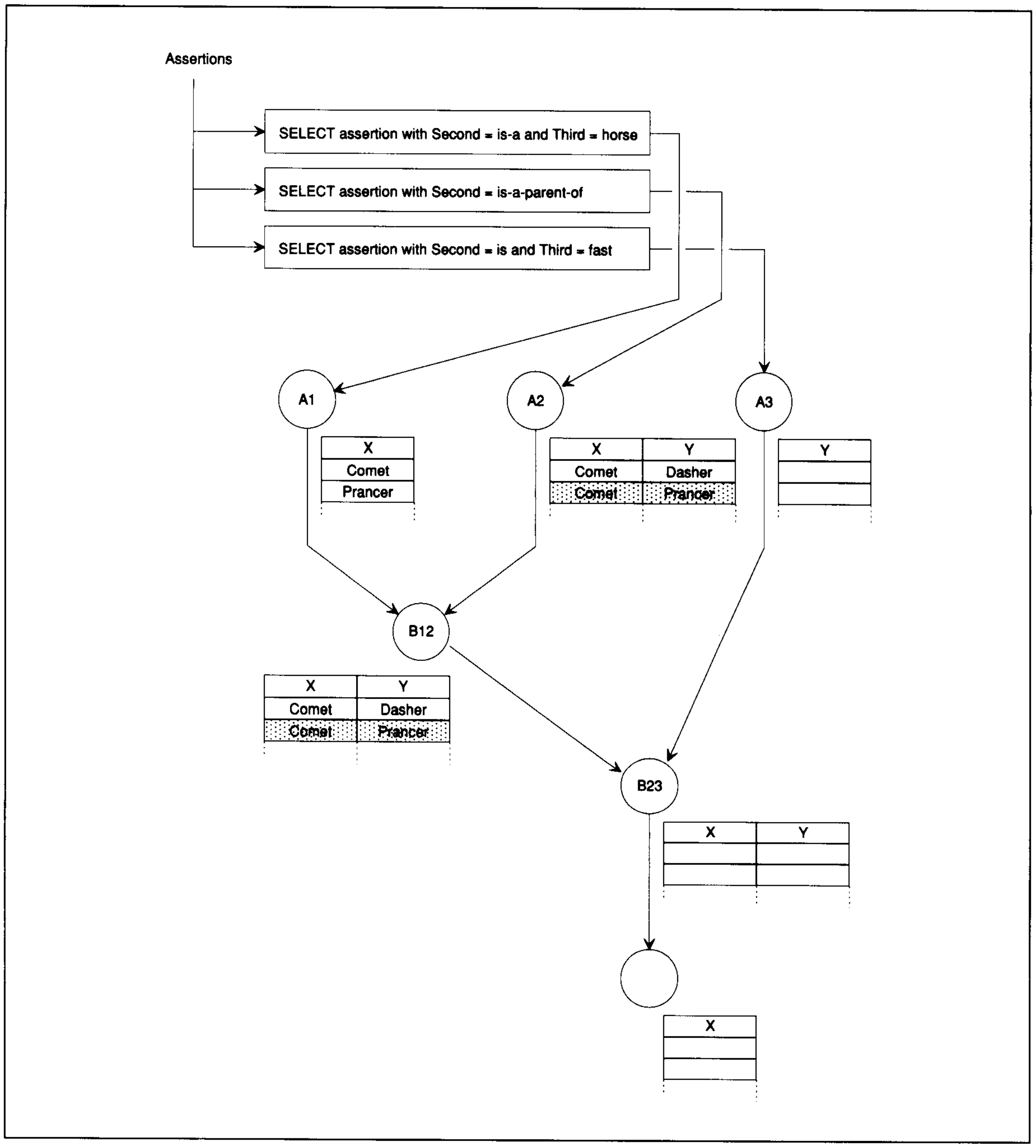

Assertions
SELECT assertion with Second = is-a and Third = horse
SELECT assertion with Second = is-a-parent-of
SELECT assertion with Second = is and Third = fast
A1
X
Comet
Prancer
A2
X
Y
Comet
Dasher
Comet
Prancer
A3
Y
B12
X
Y
Comet
Dasher
Comet
Prancer
B23
X
Y
X

Figure 7.10 Here, the state of the rete is captured just following the additions made in response to the fourth assertion, *Comet is-a-parent-of Prancer.* The additions are shown shaded.

relation are joined to A3's relation. As before, the JOIN operation is done to determine whether the variable bindings expressed in the added records correspond to variable bindings already expressed in the other relation involved in the JOIN operation.

Now consider the state of the rete after you add three more assertions—*Dasher is-a-parent-of Thunder*, *Thunder is fast*, and *Thunder is-a horse.* A1's relation indicates that there are three horses:

A1

X
Comet
Prancer
Thunder

A2's relation indicates that Comet is a parent of two children, and Dasher is a parent of one child:

A2

X	Y
Comet	Dasher
Comet	Prancer
Dasher	Thunder

A3's relation indicates that Prancer and Thunder are fast:

A3

Y
Prancer
Thunder

Next, the information in the alpha-node relations is joined to form the beta-node relations:

B12

X	Y
Comet	Dasher
Comet	Prancer

B23

X	Y
Comet	Prancer

Next, the ninth assertion, *Dasher is-a horse*, initiates another wave of additions to the records in the rete—the additions indicated by the shading in figure 7.11.

The first of these additions is the new record in A1's relation. This new record is joined to A2's relation, producing a new record in B12's relation:

B12 (increment)

X	Y
Dasher	Thunder

But now this new B12 record is joined to A3's relation producing a new record for B23's relation:

B23 (increment)

X	Y
Dasher	Thunder

Projection of this new record over the X field yields another possible binding for x in the rule's consequent:

Parent-Rule Bindings (increment) B23

X
Dasher

Thus, after all nine assertions are processed, the possible bindings for x in the rule's consequent are given by the following relation:

Parent-Rule Bindings B23

X
Comet
Dasher

As you have seen, adding a new relation produces a wavelike phenomenon that continues through the rete as long as JOIN operations produce new records. Note that all the relational operations involve only small relations containing a few assertions; they never involve the entire accumulated database of assertions.

Although the example may seem complicated, the procedures for building and using a rete are straightforward:

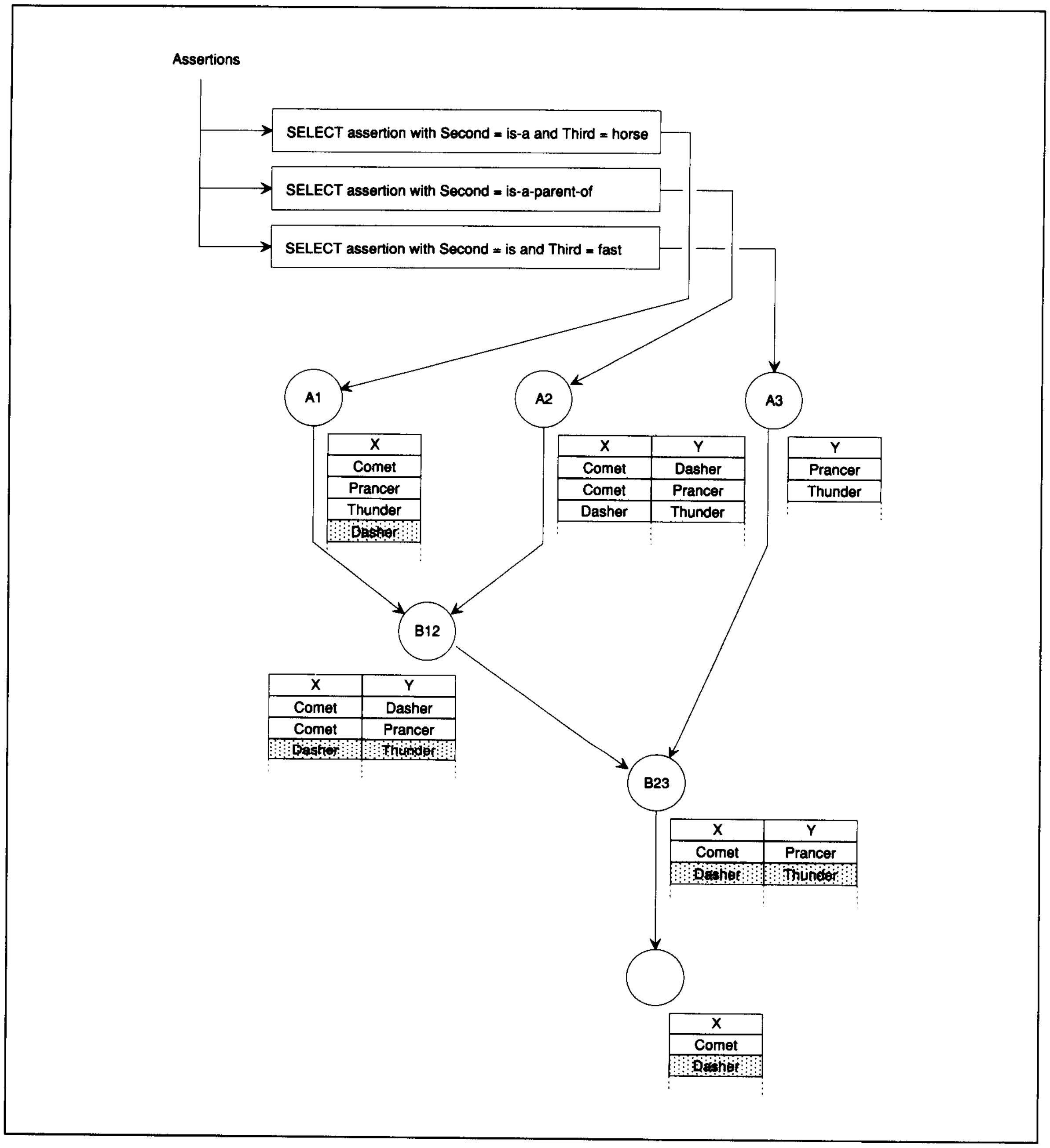


Figure 7.11 Here, the state of the rete is captured just following the additions made in response to the ninth assertion, *Dasher is-a horse*. The most recent changes are shown shaded.

To construct a rete,

- ▷ For each antecedent pattern that appears in the rule set, create a SELECT operation that examines new assertions.
- ▷ For each rule,
  - ▷ For each antecedent,
    - ▷ Create an alpha node and attach it to the corresponding SELECT operation, already created.
- ▷ For each alpha node, except the first,
  - ▷ Create a beta node.
  - ▷ If the beta node is the first beta node, attach it to the first and second alpha nodes.
  - ▷ Otherwise, attach the beta node to the corresponding alpha node and to the previous beta node.
- ▷ Attach a PROJECT operation to the final beta node.

To use a rete,

- ▷ For each assertion, filter the assertion through the SELECT operations, passing the assertion along the rete to the appropriate alpha nodes.
- ▷ For each alpha node receiving an assertion, use the PROJECT operation to isolate the appropriate variable bindings. Pass these new bindings, if any, along the rete to the appropriate beta nodes.
- ▷ For each beta node receiving new variable bindings on one of its inputs, use the JOIN operation to create new variable binding sets. Pass these new variable binding sets, if any, along the rete to the next beta node or to the final PROJECT operation.
- ▷ For each rule, use the PROJECT operation to isolate the variable bindings needed to instantiate the consequent.

SUMMARY

- ■ Rule-based systems were developed to take advantage of the fact that a great deal of useful knowledge can be expressed in simple if–then rules.
- ■ Many rule-based systems are deduction systems. In these systems, rules consist of antecedents and consequents. In one example, a toy deduction system identifies animals.

- Deduction systems may chain together rules in a forward direction, from assertions to conclusions, or backward, from hypotheses to questions. Whether chaining should be forward or backward depends on the problem.
- Many rule-based systems are reaction systems. In these systems, rules consist of conditions and actions. A toy reaction system bags groceries.
- Reaction systems require conflict-resolution strategies to determine which of many triggered rules should be allowed to fire.
- Depth-first search can supply compatible bindings for both forward chaining and backward chaining.
- Relational operations support breadth-first search for compatible bindings during forward chaining. The rete procedure performs relational operations incrementally as new assertions flow through a rule-defined rete.

BACKGROUND

The rete procedure was developed by C. L. Forgy [1982].

MYCIN was developed by Edward Shortliffe and colleagues at Stanford University [1976].

XCON was developed to configure the Digital Equipment Corporation's VAX computers by John McDermott and other researchers working at Carnegie Mellon University, and by Arnold Kraft, Dennis O'Connor, and other developers at the Digital Equipment Corporation [McDermott 1982].