Chapter 3

Production Rules and Inference

In the early 1970s, Alan Newell and Herbert Simon introduced the notion of a *production system* as a psychological model of human behaviour. In this model, part of the human knowledge is being represented in separate units called *productions* or *production rules*. These units contain information concerning actions a person has to take upon the perception of certain stimuli from the environment. Such actions may affect a person's view on the environmental reality, on the one hand because previous assumptions may have to be revised, and on the other hand because possibly new phenomena have to be explained. The model of Newell and Simon closely resembles the two-process theory of memory in cognitive psychology, where two different mechanisms for the storage of incoming sensory information are distinguished: the short-term memory, and the long-term memory, respectively. The short-term memory only contains a limited amount of rapidly decaying information. It corresponds to the part of a production system in which input and derived data are kept. The long-term memory is for permanent storage of information, and corresponds to the rule base of a production system in which the production rules are specified. The production-rule formalism has been employed by many other researchers in addition to Newell and Simon. Most of them, however, view the production-rule formalism merely as a formal language for expressing certain types of knowledge. The formalism has for example been used in the Heuristic DENDRAL system for predicting the molecular structure of compounds, as has been discussed in chapter 1. Part of the knowledge necessary for the purpose of this system has been encoded by means of production rules. The greatest success of the formalism, however, came with the building of the MYCIN and EMYCIN systems, in which the suitability of production rules for building diagnostic intelligent systems was convincingly shown. Another successful system, more directly employing the work of Newell and Simon, is OPS5, and its successor CLIPS, which will be discussed in Chapter 8.

Many present-day intelligent systems use the production-rule formalism as a knowledge representation scheme. Practical experience with production rules has proven this formalism to be particularly suitable in solving classification problems in which the available knowledge takes the form of rules of thumb. In other types of applications, such as design and planning, production rules have been applied with success as well. The suitability of the production system approach for building certain types of intelligent systems not only depends on the production-rule formalism itself, but also on the type of inference method employed for rule-based reasoning.

In the present chapter, we closely look at a number of important notions from production

systems. Section 3.1 discusses the various schemes for representing knowledge a production system offers. We proceed by discussing the two basic reasoning methods for systems with production rules in Section 3.2. To conclude, Section 3.3 pays attention to limitations of production rules.

3.1 Knowledge representation in a production system

A production system offers a number of formalisms for representing expert knowledge. The most important of these, of course, is the production-rule formalism, in which the actual problem-solving knowledge is expressed. The entire set of production rules in a production system is called its *rule base*. In addition to the production-rule formalism, a production system provides a means for defining the objects referred to in the production rules, called the *domain declaration* in this book. The rule base and the domain declaration together constitute the *knowledge base* of the production system. These and other schemes for representing knowledge will be discussed in detail in the subsequent sections.

3.1.1 Variables and facts

During a consultation of the knowledge base of a production system, information is constantly being added, removed, or modified as a result of the application of production rules, of data entered by the user, or as a result of querying some database. The facts that become known to the system during a consultation, are stored in a so-called *fact set*, also known as the *global database* or *working memory* of the system.

Factual information can be represented in a number of ways. One simple way is to represent facts by means of *variables* which can take either a single constant or a set of constants as a value. Note that the set of all variables defined in a production system together with their possible values, presents a picture of the information which is relevant in the field modelled in the system.

In general, two types of variable are discerned:

- *single-valued variables*, that is, variables which can take at most one constant value at a time, and
- *multi-valued variables*, that is, variables which can take a set of constants for a value.

Single-valued variables are used to represent information which in the case under consideration is unique; multi-valued variables are used for representing a collection of interrelated facts.

EXAMPLE 3.1

In a intelligent system, a variable with the name *capabilities* may be used for storing information about the capabilities of a certain person. This variable has to be multi-valued, because people may have more than one capability at the same time. An example of a single-valued variable is the *gender* of a person.

Properties of variables, such as whether they are single- or multi-valued, and usually also information concerning the values a variable is allowed to take, are all described in the domain declaration of a knowledge base. The following definition provides a formal description of such a domain declaration.

Definition 3.1 *Let τ denote a nonempty set of constants, called a* type. *A* typed variable declaration *is an expression of one of the following forms:*

- $x^s : \tau$, *where x^s is a single-valued variable;*
- $x^m : 2^\tau$, *where x^m is a multi-valued variable.*

Untyped variable declarations *are expressions of the form x^s, in the single-valued case, or x^m, in the multi-valued case. A set D of variable declarations for all variables occurring in the knowledge base is called the* domain declaration *of the knowledge base.*

Examples of types are the set of integer numbers, denoted by **int**, the set of real numbers, denoted by **real**, and finite sets of constants such as $\{intelligent, attractive, outspoken\}$. Note that a domain declaration is similar to a variable declaration part in, for instance, Pascal. It restricts the values a variable may take.

A variable together with the value(s) it has adopted during a consultation is called a *fact*.

Definition 3.2 *A* fact *is a statement having one of the following forms:*

- $x^s = c$, *where $c \in \tau$ if x^s is a single-valued variable declared as $x^s : \tau$;*
- $x^m = C$, *where $C \subseteq \tau$ if x^m is a multi-valued variable declared as $x^m : 2^\tau$.*

A fact set *has the following form:*

$$\{x_1^s = c_1, \ldots, x_p^s = c_p, x_1^m = C_1, \ldots, x_q^m = C_q\}$$

where c_i are constants and C_j are sets of constants. A variable may only occur once in a fact set.

EXAMPLE 3.2

Consider the following domain declaration

$$\begin{aligned} D = \; & gender : \{female, male\}, \\ & age : \mathbf{int}, \\ & personality : 2^{\{intelligent, outspoken, arrogant\}}, \\ & hire : \{yes, no\} \end{aligned}$$

of a knowledge base of an intelligent system. The following facts are typical elements of a fact set after a specific consultation:

$$\begin{aligned} gender &= male \\ age &= 27 \\ personality &= \{intelligent, outspoken\} \\ hire &= \{yes\} \end{aligned}$$

The statement '$x^s = unknown$' (or '$x^m = unknown$', respectively) is used to indicate that the variable x^s (or x^m, respectively) has not been assigned an actual value; x^s (or x^m) is then called *unknown*. The constant *unknown* has a special meaning: it expresses that the inference engine has not been able to derive one or more values for the variable. Since its meaning goes beyond (that is, *meta*) the contents of the fact set and the knowledge base, the constant *unknown* is called a *meta-constant*.

It must be emphasized that a fact set is not a part of a knowledge base, but instead is a separate component of the system. A fact set comprises information which is specific for a particular consultation, whereas a knowledge base only contains *declarations* of variables and therefore does not specify consultation-dependent values. In the following, we will frequently assume that suitable variable declarations are present in the domain declaration of a knowledge base, without explicitly referring to them.

3.1.2 Conditions and conclusions

At the beginning of this chapter, we mentioned that production rules are used for representing problem-solving knowledge from a specific problem domain. The major part of this type of knowledge takes the form of *heuristic rules* or rules of thumb, which, as we shall see, are the informal, real-life analogies of production rules. In a heuristic rule, several conditions and conclusions are interrelated, as follows:

if
    certain *conditions* are fulfilled,
then
    certain *conclusions* may be drawn.

EXAMPLE 3.3

An example of a heuristic rule, taken from the domain of the assessment of applicants for a job, is the following:

if
    the person's age less than 60, and
    the person is intelligent, and
    the person is not arrogant
then
    the person will be hired

In the process of knowledge engineering, such heuristic rules have to be transformed into their formal counterparts, that is, into production rules. A production rule just like a heuristic rule, consists of a number of conditions and conclusions. In a production rule, which unlike a heuristic rule is a formal statement, the conditions and conclusions comprise the following elements:

- symbolic and numeric constant values
- variables

- predicates and actions

An important part of the translation process therefore concerns the identification of the variables and constants that are relevant in the heuristic rules.

EXAMPLE 3.4

Consider the preceding heuristic rule once more. In the first condition a variable *age* may be identified, the second and third conditions concerns a person's *personality*, and, finally, the conclusion concerns the question whether the firm is going to *hire* the person. The following constant values may be introduced to represent the information further comprised in the heuristic rule: 60, *intelligent*, *arrogant* and *yes*.

Several syntactic forms have been devised for the representation of production rules. In the present book we employ the syntax described in the following definition.

Definition 3.3 *A* production rule *is a statement having the following form:*

$$
\begin{array}{lcl}
\langle\text{production rule}\rangle & ::= & \textbf{if } \langle\text{antecedent}\rangle \textbf{ then } \langle\text{consequent}\rangle \textbf{ fi} \\
\langle\text{antecedent}\rangle & ::= & \langle\text{disjunction}\rangle \ \{\textbf{and } \langle\text{disjunction}\rangle\}^* \\
\langle\text{disjunction}\rangle & ::= & \langle\text{condition}\rangle \ \{\textbf{or } \langle\text{condition}\rangle\}^* \\
\langle\text{consequent}\rangle & ::= & \langle\text{conclusion}\rangle \ \{\textbf{also } \langle\text{conclusion}\rangle\}^* \\
\langle\text{condition}\rangle & ::= & \langle\text{predicate}\rangle\textbf{(}\langle\text{variable}\rangle\textbf{,}\langle\text{constant}\rangle\textbf{)} \\
\langle\text{conclusion}\rangle & ::= & \langle\text{action}\rangle\textbf{(}\langle\text{variable}\rangle\textbf{,}\langle\text{constant}\rangle\textbf{)} \\
\langle\text{predicate}\rangle & ::= & \mathit{same} \mid \mathit{notsame} \mid \mathit{greaterthan} \mid \ldots \\
\langle\text{action}\rangle & ::= & \mathit{add} \mid \mathit{remove} \mid \ldots
\end{array}
$$

In the production-rule formalism it is assumed that the **or** operator has a higher precedence than the **and** operator. Note that the nesting of conjunctions and disjunctions is limited; this is typical for production systems.

A condition is built from a *predicate* and two associated arguments: a variable and a constant. By means of its predicate, a condition expresses a comparison between the specified constant value and the actual value(s) the specified variable has adopted. In the context of production systems, a predicate is a function which upon evaluation returns either the truth value *true* or the value *false*. The way predicates are evaluated is illustrated by means of the following example.

EXAMPLE 3.5

Let F be the following fact set:

$$F = \{age = 70, personality = \{intelligent, arrogant\}\}$$

where *age* is a single-valued variable and *personality* is a multi-valued one. Now consider the following condition:

$$same(personality, intelligent)$$

The predicate *same* returns upon evaluation the truth value *true* if *intelligent* is one of the constants in the set of constants adopted by the multi-valued variable *personality*; otherwise, the value *false* is returned. So, in the present example, the evaluation of the condition will return the value *true*. In case of a single-valued variable, *same* tests whether or not the constant specified as its second argument is equal to the constant which the variable mentioned as its first argument has adopted. Given the present fact set, the condition

$$lessthan(age, 60)$$

therefore yields the value *false* upon evaluation.

A conclusion is built from an *action* and two associated arguments. An action can be considered to operate on a variable. The most frequently applied action is *add*, which adds the constant specified as its second argument to the value set of the multi-valued variable mentioned in its first argument; in case of a single-valued variable, the action *add* assigns the constant value from its second argument to the specified variable.

EXAMPLE 3.6

Consider the fact set F:

$$F = \{personality = \{arrogant\}, gender = male, age = 25\}$$

in which *personality* is the only multi-valued variable. The action *add* in the conclusion

$$add(personality, outspoken)$$

adds the constant *outspoken* to the set of constant values the variable *personality* already has adopted. So, after evaluation of this conclusion we have obtained the following fact set F':

$$F' = \{personality = \{arrogant, outspoken\}, gender = male, age = 25\}$$

A production rule is built from such conditions and conclusions.

EXAMPLE 3.7

The heuristic rule informally introduced above can now be translated into the production-rule formalism, for example as follows:

if
    $lessthan(age, 60)$ **and**
    $same(personality, intelligent)$ **and**
    $notsame(personality, arrogant)$
then
    $add(hire, yes)$
fi

To conclude, the following example discusses a production rule comprising more than one conclusion.

EXAMPLE 3.8

The following heuristic rule:

if
 the person is quiet, and
 the person is calm
then
 it is likely the person is introvert, and
 it is conceivable that the person is stable

may be expressed in the production-rule formalism as follows:

if
 same(*expression*, *quiet*) **and**
 same(*emotionality*, *calm*)
then
 add(*person*, *introvert*) **also**
 add(*person*, *stable*)
fi

The only action we have considered up till now is the action *add*. If this action is the only one specified in the consequent of a production rule, then the rule closely resembles a logical implication, in which the conditions of the rule appear on the left of the implication symbol and the conclusions are specified on the right of it. This interpretation, however, is no longer valid when actions other than *add* have been specified in the consequent of a rule. Consider, for example, the action *remove* which upon execution cancels the assignment of a specific constant to a single-valued variable, or deletes a constant from the set of constants of a multi-valued variable. A production rule in which this action has been specified cannot possibly be viewed as a logical implication. More about the correspondence between the production-rule formalism and logic will be said in Section 3.1.4.

In Table 3.1 the semantics of some frequently used predicates and actions are described. From the special meaning of the constant *unknown* as discussed in the preceding section, we have that the predicates *known* and *notknown* mentioned in the table have a special meaning as well: they express knowledge concerning the derivability of values for variables, which again goes beyond the contents of the fact set and the rule base of a production system. We call such predicates *meta-predicates*. In Section 3.2.1 we will turn again to the meanings of the predicates and the effects the different actions have on a fact set.

3.1.3 Object-attribute-value tuples

The production-rule formalism introduced in the preceding section does not provide a means for specifying formal relationships between the variables of concern. In many problem do-

Table 3.1: Some predicates and actions, and their meaning

Example	*Semantics for single-valued variables*	*Semantics for multi-valued variables*
same(x, c)	$x^s = c$	$c \in x^m$
notsame(x, c)	$x^s \neq c$ and $x^s \neq \mathit{unknown}$	$c \notin x^m$ and $x^m \neq \mathit{unknown}$
lessthan(x, c)	$x^s < c$	-
greaterthan(x, c)	$x^s > c$	-
known(x)	$x^s \neq \mathit{unknown}$	$x^m \neq \mathit{unknown}$
notknown(x)	$x^s = \mathit{unknown}$	$x^m = \mathit{unknown}$
add(x, c)	$x^s \leftarrow c$	$x^m \leftarrow x^m \cup \{c\}$
remove(x, c)	$x^s \leftarrow \mathit{unknown}$	if $x^m = \{c\}$ then $x^m \leftarrow \mathit{unknown}$ else $x^m \leftarrow x^m \setminus \{c\}$

mains to be modelled however, one can often discern separate subdomains, or *objects*, which are interrelated in some way. Each subdomain then is described by a number of properties or *attributes* which are specific for that subdomain. The idea is illustrated in the following example.

EXAMPLE 3.9

In the heuristic rule from Example 3.8, a description is given of the personality of a person. Personality traits are usually described by several factors which are specific for the person. We now may view *traits* as a separate object, and *character* and *emotional* as attributes belonging to that object.

In production systems, objects are often used to explicitly group the properties which are mentioned in the heuristic rules. The objects themselves are often even further exploited for directing the inference process. If we extend the production-rule formalism to include the specification of an object, predicates and actions have to be changed from binary, in the previous case of variable-value pairs, to ternary operators. We call a tuple consisting of an object, an attribute and a constant value, an *object-attribute-value tuple*, or *o-a-v triple*. The following simple extension to the original syntax definition of production rules is thus obtained:

$$\langle\text{condition}\rangle \;::=\; \langle\text{predicate}\rangle(\langle\text{object}\rangle,\langle\text{attribute}\rangle,\langle\text{constant}\rangle)$$
$$\langle\text{conclusion}\rangle \;::=\; \langle\text{action}\rangle(\langle\text{object}\rangle,\langle\text{attribute}\rangle,\langle\text{constant}\rangle)$$

Note that an object-attribute pair always explicitly indicates that the mentioned attribute belongs to the specified object. The object-attribute pairs in an object-attribute-value tuple just act like the variables discussed in the foregoing sections. The attributes of an object are declared as being either *single-valued* or *multi-valued*, in a way resembling the declaration of variables. In the sequel, we shall write $o.a^s$ to denote the single-valued attribute a^s belonging to the object o; we use $o.a^m$ for the multi-valued case.

EXAMPLE 3.10

The following production rule

if
 same(*person*, *gender*, *male*) **and**
 same(*person*, *behaviour*, *quiet*) **and**
 same(*personality*, *emotional*, *stable*)
then
 add(*personality*, *introvert*, *yes*)
fi

concerns the objects *person* and *personality*. Attributes of the object *personality* referred to in the rule are *emotional* and *introvert*. The attributes of the object *person* referred to in the rule are *gender* and *behaviour*.

The extension of the definitions of predicates and actions with the specification of an object enables us to express relationships between objects and their attributes. However, it is still not possible to explicitly express relationships between objects amongst themselves in a natural way. Yet it may be desirable in an intelligent system to have this type of information available as well. Therefore, in addition to the rule base, often an *object schema* is present in an intelligent system; it is usually added to the domain declaration of a knowledge base. An object schema defines the interrelationships between the objects, and the relationships between the objects and their associated attributes. Figure 3.1 shows a portion of the object schema describing some features of a person. An object is represented by an ellipse. A solid line is used to represent a relationship between two objects. In Figure 3.1 the object *personality* is called a *subobject* of the object *person*. A dashed line indicates a relationship between an object and an associated attribute. We note that an object schema is frequently used for storing meta-information, for example directives to the inference engine to handle certain attributes in a special way.

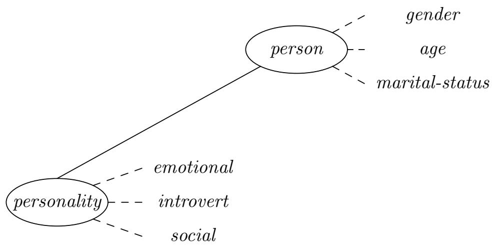

Figure 3.1: An object scheme.

3.1.4 Production rules and first-order predicate logic

In the preceding sections, we have informally discussed the meanings of predicates and actions in a production system. In general, we can say that the semantics of production rules in a

production system is described in terms of a specific inference method for applying the rules. The only semantics for a production system available therefore is procedural in nature. This is contrary to first-order logic which also has a neat declarative semantics. There is, however, a strong relationship between the production-rule formalism and the formalism of first-order logic, as we already briefly touched upon in Section 3.1.1. In most rule bases, at least part of the production rules can be translated into first-order predicate logic in a straightforward and natural way. This is fortunate, since it enables developing a rule base without precise knowledge of the working of the inference engine: the declarative readings of the corresponding logical formulas can thus be exploited.

Let us start by looking at the correspondence between conditions and conclusions in production rules on the one hand and literals in first-order logic on the other hand. Without loss of generality, we assume that the production system makes use of object-attribute-value tuples. As we discussed above, the predicates *same* and *notsame* test whether the specified attribute of some object has, respectively has not, the specified constant as a value or in its set of values. Again, we distinguish between multi-valued and single-valued attributes. A multi-valued attribute may be viewed as a relation A defined on the cartesian product of a set of objects and a set of constants, that is, $A \subseteq O \times V$, where O is the set of objects, and V is the set of constant values. Recall that in first-order predicate logic, predicate symbols may be employed for representing such relations. Conditions of the form $same(o, a^m, v)$, and $notsame(o, a^m, v)$, containing a multi-valued attribute, may therefore be translated into first-order logic in the literals $a(o, v)$, and $\neg a(o, v)$, respectively. In the single-valued case, we have to take into account that the attribute may adopt at most one value at a time. Single-valuedness is best expressed in first-order logic by using function symbols. Single-valued attributes can therefore be viewed as functions a from the set of objects O to the set of values V, that is, $a : O \to V$. Note that for expressing the meaning of the predicate *same*, we have the equality predicate $=$ at our disposal. The following translation is now rather straightforward: the condition $same(o, a^s, v)$ is translated into the literal $a(o) = v$, and the condition $notsame(o, a^s, v)$ is translated into $\neg a(o) = v$. Many other predicates used in production rules can be translated in much the same way. Table 3.2 summarizes some of these translations. Note that the semantics of first-order logic implies that the unit clause $a(o, v)$ in the presence of the unit clause $\neg a(o, v)$, leads to an inconsistency. In addition, the unit clauses $a(o) = v$ and $a(o) = w$, where $v \neq w$, are inconsistent in the presence of the equality axioms. As can be seen from Table 3.2, the action *add* is treated the same

Table 3.2: Translation of conditions and actions into first-order logic.

Example	*Logic representation for single-valued attributes*	*Logic representation for multi-valued attributes*
$same(o, a, v)$	$a(o) = v$	$a(o, v)$
$notsame(o, a, v)$	$\neg a(o) = v$	$\neg a(o, v)$
$equal(o, a, v)$	$a(o) = v$	–
$lessthan(o, a, v)$	$a(o) < v$	–
$greaterthan(o, a, v)$	$a(o) > v$	–
$add(o, a, v)$	$a(o) = v$	$a(o, v)$

way as the predicate *same* is; this reflects the transition from a procedural to a declarative semantics. The meta-predicates *known* and *notknown* are not included in Table 3.2, since it is not possible to express meta-information in standard first-order logic. A similar remark can be made concerning the action *remove*. It is noted that special non-standard logics in which such meta-predicates and non-monotonic actions can be expressed, have been developed. The subject of non-standard logic, however, goes beyond the scope of the present book.

Further translation of a production rule into a logical formula is now straightforward. The general translation scheme is as follows:

$$
\begin{array}{lll}
\textbf{if} & & \\
\quad c_{1,1} \textbf{ or } c_{1,2} \textbf{ or } \ldots \textbf{ or } c_{1,m} \textbf{ and} & & ((c'_{1,1} \vee c'_{1,2} \vee \ldots \vee c'_{1,m}) \wedge \\
\quad \ldots & & \ldots \\
\quad c_{n,1} \textbf{ or } c_{n,2} \textbf{ or } \ldots \textbf{ or } c_{n,p} & \Rightarrow & (c'_{n,1} \vee c'_{n,2} \vee \ldots \vee c'_{n,p})) \\
\textbf{then} & & \rightarrow \\
\quad a_1 \textbf{ also } a_2 \textbf{ also } \ldots \textbf{ also } a_q & & (a'_1 \wedge a'_2 \wedge \ldots \wedge a'_q) \\
\textbf{fi} & &
\end{array}
$$

where conditions $c_{i,j}$ and actions a_k are translated into literals $c'_{i,j}$ and a'_k, respectively, as prescribed by Table 3.2 From the table it can readily be seen that the kind of production rules we have looked at, are translated into *ground* logical implications.

The following example illustrates the translation scheme.

EXAMPLE 3.11

Consider the following production rule:

if
  $greaterthan(blood, systolic\text{-}pressure, 140)$ **and**
  $greaterthan(heart, rate, 100)$ **and**
  $same(person, skin, pale)$ **or**
  $same(eyes, iris, wide)$
then
  $add(person, condition, rage)$
fi

Translation of this rule into first-order logic yields the following implication:

$$
\begin{array}{l}
(systolic\text{-}pressure(blood) > 140 \wedge \\
\ rate(heart) > 100 \wedge \\
\ (skin(person, pale) \vee \\
\ \ eyes(iris, wide))) \\
\rightarrow \\
condition(person, rage)
\end{array}
$$

The formalism of first-order logic is in certain ways more flexible than the production-rule formalism is. For instance, it is not restricted to the use of *o-a-v* triples only; it allows, for example, to specify the literal $systolic\text{-}pressure(person, blood) > 140$, instead of the first literal in the implication shown above. The latter literal is more in correspondence with the intended meaning of the original heuristic rule.

To conclude, the fact set of a production system may be translated into the logic formalism much in the same way as the rule base is. A fact $o.a^s = v$ concerning a single-valued attribute a^s is translated into a unit clause $a(o) = v$. Recall that in the presence of the equality axioms single-valuedness is guaranteed. Now consider the multi-valued case. A fact concerning a multi-valued attribute a^m is translated into a set of unit clauses $a(o, v_i)$ for each v_i in the set of constant values a^m has adopted. However, it is not sufficient to only add these positive clauses: it is typical for production systems that values of multi-valued attributes not explicitly entered into the fact set, are taken implicitly by the system as being not true. This behaviour has been copied from human problem solving. For example, a medical doctor usually only records in a patient report the symptoms and signs that have actually been observed in the patient; all information not explicitly recorded for the specific patient is implicitly assumed to be negative. This aspect of problem solving is reflected in the meaning of the *notsame* predicate. Note that this way of dealing with negations is quite different from the meaning of a negative literal in first-order logic which only holds in a model in which it has actually been satisfied. For a correct meaning of the *notsame* predicate therefore we have to add explicitly unit clauses $\neg a(o, v_i)$ for the remaining constants v_i occurring in the type τ of a^m, as soon as at least one positive unit clause $a(o, v_j)$, $j \neq i$, occurs in the fact set (in case of an untyped variable, we add unit clauses $\neg a(o, v_i)$ for the remaining constants v_i mentioned in the rule base). The explicit addition of negative unit clauses is called *negation by absence*. It is a special case of the closed world assumption mentioned before in Chapter 2.

EXAMPLE 3.12

Consider the following domain declaration:

$$D = \{disorder^m : 2^{\{flu, hepatitis\}}, age^s : \textbf{int}\}$$

and the fact set $F = \{disorder^m = \{flu\}, age^s = 70\}$. We obtain the following translation into first-order logic:

$$
\begin{array}{l}
disorder(patient, flu) \\
\neg disorder(patient, hepatitis) \\
age(patient) = 70
\end{array}
$$

3.2 Inference in a production system

Several inference methods have been devised for dealing with production rules. An inference method in a production system explicitly exploits the difference between facts and rules: it operates on the fact set, which can be looked upon as the global working memory for the production rules. Note that in logic, such an explicit separation between facts (unit clauses) and rules (implications) generally is not made, although it should be noted that several inference methods, such as SLD resolution, make use of a similar distinction.

Roughly speaking, an inference method selects and subsequently applies production rules from a rule base. In applying the selected production rules, it executes the actions specified in their conclusions. Execution of such actions may cause facts to be added to, to be modified in, or to be deleted from the fact set. In this chapter we discuss the addition and deletion of facts: a discussion of the modification of facts as a result of executing actions will be postponed until Chapter 8, where we shall discuss the language OPS5 and CLIPS. Figure 3.2 shows the general idea of inference in a production system. The manipulation of production rules and facts is depicted in the figure by means of arrows.

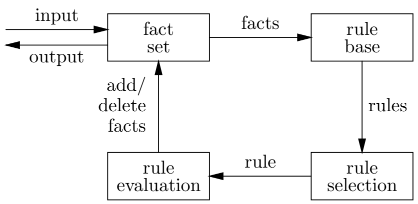

Figure 3.2: Global architecture of a production system.

The distinction between the two basic forms of inference mentioned in chapter 1 is essential when considering production systems; they yield entirely different reasoning strategies. Before discussing the two basic inference methods, top-down and bottom-up inference, in detail, we introduce them informally with the help of a simplified example, in which we abstract from the syntactical structure of conditions, conclusions, and facts. To simplify matters further, we suppose that all conditions of the form c succeed upon evaluation in the presence of the fact c. Now, consider Table 3.3.

Table 3.3: Production system before and after execution.

State	*Component*	*Top-down Inference*	*Bottom-up Inference*
Initial	*Goal*	g	–
	Facts	$\{a\}$	$\{a\}$
	Rules	R_1 : **if** b **then** g **fi**	R_1 : **if** b **then** g **fi**
		R_2 : **if** g **then** c **fi**	R_2 : **if** g **then** c **fi**
		R_3 : **if** a **then** b **fi**	R_3 : **if** a **then** b **fi**
Final	*Facts'*	$\{a, b, g\}$	$\{a, b, c, g\}$

- *Top-down inference* starts with a statement of one or more *goals* to be achieved. In our example, we have just the single goal g. A goal may match with a conclusion of one or more production rules present in the rule base. All production rules thus matching with a certain goal are selected for application. In the present case, the only rule selected is R_1. Each one of the selected production rules is subsequently applied by first considering the conditions of the rule as the new subgoals to be achieved. Roughly speaking, if there are facts present in the fact set which match with these new subgoals, then these subgoals are taken as been achieved; subgoals for which no matching facts can be found, are matched against the conclusions of the production rules from the rule base. Again, matching production rules are selected for application. In our example, we have from the rule R_1 the new subgoal b, which in turn causes the selection of the production rule R_3. This process is repeated recursively. Note that in top-down inference, production rules are applied in a backward manner. When all the subgoals, that is, the conditions of a selected production rule, have been achieved, then the actions in the conclusions of the rule are executed, possibly causing changes in the fact set. Since the subgoal a of the selected production rule R_3 matches with the fact a, we have that the condition of the rule is fulfilled. Subsequently, its action is executed, yielding the new fact b. This new fact in turn fulfills the condition b of rule R_1, which led to the selection of R_3. The inference process is terminated as soon as the initially specified goals have been achieved. Note that only production rules relevant for achieving the initially given goals are applied. This explains why the rule R_2 in table 3.3 has not been used.

- *Bottom-up inference* starts with a fact set, in our example $\{a\}$. The facts in the fact set are matched against the conditions of the production rules from the rule base. If for a specific production rule all conditions are fulfilled, then it is selected for application. The rule is applied by executing the actions mentioned in its conclusions. So, in our example the rule R_3 will be applied first. The application of the selected production rules is likely to result in changes in the fact set, thus enabling other production rules to be applicable. In the present case, after the application of rule R_3, we have obtained the new fact set $\{a, b\}$, which results in the rule R_1 now being applicable. The fact g added to the fact set as a result of executing the action of rule R_1, results in the subsequent application of R_2. We therefore conclude that the final fact set is equal to $\{a, b, c, g\}$. The inference process is terminated as soon as all applicable production rules have been processed.

Note the difference in the resulting fact sets in table 3.3 obtained from applying top-down and bottom-up inference, respectively. As a consequence of their different inference behaviour, top-down and bottom-up inference are suitable for developing different kinds of intelligent systems. Top-down inference is often used in inference engines of diagnostic intelligent systems in which the inference process is controlled by a specific goal and a small amount of data. Bottom-up inference is most suitable for applications in which the interpretation of a vast amount of data is important, and in which there are no preset goals. In the following section, we first address top-down inference before discussing bottom-up inference in Section 3.2.4.

3.2.1 Top-down inference and production rules

As we discussed before, top-down inference usually is incorporated in diagnostic intelligent systems. If it employs the variable-value representation, such a system tries to derive facts concerning one or more preset goal variables. In a typical medical diagnostic intelligent system for example, one could think of a multi-valued variable *diagnosis*, for which we want to establish for a specific patient the set of values (that is, the possible disorders). So, in general, top-down inference starts with a set of *goals* $\{G_1, G_2, \ldots, G_m\}$, $m \geq 1$, essentially being goal variables. For this purpose, in the domain-declaration part of a knowledge base, the variable declarations are extended to include a specification of whether or not they are goals. In the sequel, we shall use x_g^s to denote a single-valued goal variable, and x_g^m to denote a multi-valued goal variable. The subset of all facts concerning the goal variables in the fact set after applying the inference algorithm is called a *solution*.

The top-down inference algorithm used in this book for establishing the goals G_i from the initial set of goals, informally amounts to the following. Suppose that we are given a fixed rule base $\{R_1, R_2, \ldots, R_n\}$, $n \geq 1$, of production rules R_i. For deriving values for a multi-valued variable, the inference engine will try and apply production rules as long as there are production rules available which can extend the set of constants for the goal variable by adding new ones to it. In the single-valued case, the inference engine proceeds until a single value has been derived for the variable, and then it terminates. If the system has not been able to derive values for the variable from the rule base, for instance because applicable production rules are absent, then on some occasion it will turn to the user and ask for additional information. For this purpose a distinction is made between variables for which values may be asked from the user, these variables are called *askable variables*, and variables which may not be asked. The askability of a variable is again denoted in the domain declaration of the variables, this time by means of a subscript a; so, we have x_a^m for askable multi-valued variables and x_a^s for askable single-valued ones.

EXAMPLE 3.13

> In a medical intelligent system intended for diagnosing a patient's disorder, it is rather undesirable that the user is asked to specify the patient's disorder when the system has not been able to attain a diagnosis. Therefore, the variable *disorder* should not be askable. On the other hand, *symptom* and *gender* are typical examples of askable variables: these can usually not be derived using the production rules.

It will be evident that goal variables should never be askable.

The entire process of deriving values for a variable by applying the production rules, and possibly asking the user for values for it, is called *tracing* the variable. In the sequel, we abstract from the distinction between single- and multi-valued variables in the description of the top-down inference algorithm, since the basic structure of the algorithm is the same for both types of variable. The algorithm for tracing a variable pictured in the foregoing is described by the following procedure:

```
procedure TraceValues(variable)
  Infer(variable);
  if not established(variable) and askable(variable) then
    Ask(variable)
  fi
end
```

The function call established(variable) is used to examine whether or not a value for the variable variable has been obtained from the rule base by means of the procedure call Infer(variable); askable(variable) is used to determine whether the variable concerned is askable.

For deriving values for a variable from the rule base, called *inferring* the variable, the procedure Infer is invoked. In the procedure Infer, a subset $\{R_{i_1}, R_{i_2}, \ldots, R_{i_k}\}$ of the production rules is selected; a rule R_j will be selected if the name of the given variable occurs in one of the conclusions of the rule, in other words, if the given variable and the variable in

the conclusion *match*. The thus selected rules are then *applied*. The procedure for selecting production rules and subsequently applying them is shown below:

```
procedure Infer(variable)
  Select(rule-base, variable, selected-rules);
  foreach rule in selected-rules do
    Apply(rule)
  od
end
```

The actual selection of the relevant production rules from the rule base is described in the following procedure:

```
procedure Select(rule-base, variable, selected-rules)
  selected-rules ← ∅;
  foreach rule in rule-base do
    matched ← false;
    foreach concl in consequent(rule) while not matched do
      pattern ← variable(concl);
      if Match(variable, pattern) then
        selected-rules ← selected-rules ∪ {rule};
        matched ← true
      fi
    od
  od
end
```

The set of selected production rules is called the *conflict set*. In the **foreach** statement in the Infer procedure this conflict set is traversed: the rules from this set are applied one by one by means of a call to the procedure Apply, which will be described shortly. Note that this way the rules are applied exhaustively, that is, all rules concluding on the variable which is being traced, are applied.

Neither the order in which the production rules from the conflict set are applied, nor the order in which the conditions and the conclusions of the rules are evaluated, is fixed as yet. If the order in applying the selected rules has not been fixed, we speak of *nondeterminism of the first kind*. The evaluation order of the conditions and the conclusions in a rule not being fixed is called *nondeterminism of the second kind*. Nondeterminism of the first kind is resolved by using a so-called *conflict-resolution strategy*, which imposes some order on the rules from the conflict set. The simplest conflict-resolution strategy is, of course, just to apply the production rules in the order in which they have been selected from the rule base (which is then viewed as a sequence of production rules instead of as a set). More sophisticated conflict-resolution strategies order the conflict set using some context-sensitive criterion. An example of such a strategy is ordering the rules according to the number of conditions not yet fulfilled; this way, solutions which are 'close' to the information already available to the system generally prevail over more remote solutions. From the user's point of view, a system provided with a context-sensitive conflict-resolution strategy behaves much more intelligently, since likely solutions are explored before unlikely ones. Nondeterminism of the second kind is usually handled by evaluating the conditions and conclusions of a selected production rule in the order of their appearance. However, more sophisticated techniques are also possible. Sophisticated conflict-resolution strategies and evaluation ordering methods are seldom used in intelligent systems using top-down inference, since the goal-directed nature of top-down inference is itself an 'intelligent' control strategy, rendering additional ones less necessary. As a consequence, most systems employing top-down inference use the simplest strategies in solving the two types of nondeterminism, that is, they apply production rules, and evaluate conditions and conclusions in the order of their specification. This particular strategy will be called *backward chaining*.

The application of a selected production rule commences with the evaluation of its conditions. If upon evaluation at least one of the conditions is found to be *false*, then the rule is said to *fail*. If, on the other hand, all conditions evaluate to be *true*, then the rule is said to *succeed*. The application of a production rule is described in the following procedure:

```
procedure Apply(rule)
    EvalConditions(rule);
    if not failed(rule) then
        EvalConclusions(rule)
    fi
end
```

The procedure Apply first evaluates the condition part of the rule by calling EvalConditions. If this evaluation ends in success, then it evaluates the conclusions of the rule by means of EvalConclusions.

Let us take a closer look at the procedure EvalConditions which checks whether all conditions in the antecedent of the production rule yield the truth value *true* upon evaluation. Beginning with the first condition, the procedure traces the variable occurring in the condition by means of a recursive call to TraceValues. Subsequently, the test specified by means of the predicate in the condition is executed. In the EvalConditions procedure presented below, we have assumed, for simplicity's sake, that the antecedent of a production rule only comprises a conjunction of conditions.

```
procedure EvalConditions(rule)
    foreach condition in antecedent(rule) do
        var ← variable(condition);
        TraceValues(var); indirect recursion
        ExecPredicate(condition);
        if condition failed then
            return
        fi
    od
end
```

It should be noted that there are many ways to optimize the last procedure, several of which will be discussed below. The ExecPredicate procedure executes the test denoted by the predicate which has been specified in the condition under consideration. It compares the value the mentioned variable has with the constant specified in its second argument (if any). This test yields either the truth value *true* or *false*. We have already seen an example of the execution of such a predicate in the preceding section.

If all conditions of the production rule have been evaluated and have yielded the value *true*, then the rule succeeds, and its conclusions are subsequently evaluated. The evaluation of the conclusion part of a successful production rule merely comprises the execution of the actions specified in its conclusions:

```
procedure EvalConclusions(rule)
    foreach conclusion in consequent(rule) do
        ExecAction(conclusion)
    od
end
```

We have mentioned before that executing the action *add* results in the assignment of a constant value to a single-valued variable, or in the addition of a constant value to the set of values of a multi-valued variable. In Section 3.1.2, we also briefly discussed the action *remove*. Execution of this action results in deleting the specified constant value from the fact concerning the variable. Execution of this action can therefore disrupt the monotonicity of the reasoning process, thus rendering it difficult to reconstruct the inference steps which have been carried out. Furthermore, it is quite conceivable that the action is executed on a variable which has not been traced as yet, in which case the continuation of the inference is undefined. Therefore, in many intelligent systems this action is not allowed, in particular not in those systems employing backward chaining.

It may happen that a certain variable is specified in one of the conditions as well as in one of the conclusions of a production rule. Such a rule is called a *self-referencing production rule*. When applying a self-referencing rule during a top-down inference process as discussed above, the rule may occasion infinite recursion. For the moment, we therefore do not allow self-referencing rules in a rule base. In the following, we shall return to these rules once more in discussing some optimizations of the inference algorithm.

The procedures discussed in the foregoing together constitute the entire top-down inference algorithm. The following example demonstrates the behaviour of a system employing this form of inference.

EXAMPLE 3.14

Let $D = \{x_a^m, y^m, z_a^m, v_g^m, w_a^s\}$ be the domain declaration of some knowledge base. As can be seen, all variables except y and v are askable; v is the only goal variable. Suppose that we initially have the following fact set:

$$F = \{w = 5\}$$

Now consider the following production rules:

R_1 : **if** $same(x, a)$ **and** $same(x, b)$ **then** $add(z, f)$ **fi**
R_2 : **if** $same(x, b)$ **then** $add(z, g)$ **fi**
R_3 : **if** $same(x, d)$ **and** $greaterthan(w, 0)$ **then** $add(z, e)$ **fi**
R_4 : **if** $same(x, c)$ **and** $lessthan(w, 30)$ **then** $add(v, h)$ **fi**
R_5 : **if** $same(y, d)$ **and** $lessthan(w, 10)$ **then** $add(v, i)$ **fi**
R_6 : **if** $known(x)$ **and** $notsame(z, e)$ **then** $add(y, d)$ **fi**

The backward-chaining algorithm starts with the selection of the two production rules R_4 and R_5, since the goal variable v appears in their respective conclusions. The production rule R_4 will be the first one to be applied. Since there are no production rules concluding on the variable x occurring in the first condition of rule R_4, the user is asked to enter values for x. We suppose that the user answers by entering $x = \{c\}$. Evaluation of the first condition therefore yields the truth value *true*. It follows that R_4 succeeds, since the evaluation of the second condition, $lessthan(w, 30)$, yields the value *true* as well. The evaluation of the conclusion of the rule results in the addition of the fact $v = \{h\}$ to the fact set. Next, rule R_5 is applied. The first condition of this rule mentions the variable y. Since the variable y occurs in the conclusion of rule R_6, this rule is the next to be applied in order to obtain a value for y. The first condition of R_6 upon evaluation yields the truth value *true*. Evaluation of the second condition of rule R_6 ultimately results in a request to the user to supply values for the variable z, since the production rules R_1, R_2 and R_3 fail to infer values for it. When the user, for instance, provides the answer $z = \{i, j\}$, rule R_6 will succeed and the fact $y = \{d\}$ will be added to the fact set. We recall that rule R_6 was invoked during to the evaluation of the first condition of rule R_5. From the new fact $y = \{d\}$ we have that the first condition of rule R_5 yields the truth value *true* upon evaluation. Since the second condition is fulfilled as well, the action specified in the conclusion of the rule is executed: the value i is inserted into the fact concerning the variable v. We conclude that the following fact set F' has been obtained:

$$F' = \{x = \{c\}, y = \{d\}, z = \{i, j\}, v = \{h, i\}, w = 5\}$$

So, the solution arrived at is $\{v = \{h, i\}\}$.

An analysis of the *search space* generated by top-down inference can be instructive when developing optimizations of the algorithm. The search space of the top-down inference algorithm discussed in the foregoing is largely determined by the initial set of goals and the rule base, and has the form of a tree. We start the analysis by taking the backward-chaining strategy as a starting-point, and shall introduce several refinements to that algorithm. By means of the following example, we demonstrate how the search space is structured.

EXAMPLE 3.15

Let $D = \{x_a^m, y^m, z_g^s, u_a^m, v^m, w_a^m\}$ be the domain declaration of a production system. Note that the single-valued variable z is the only goal variable. The fact set initially is empty. Consider the following set of production rules:

R_1 : **if** $same(w, a)$ **and** $same(x, b)$ **then** $add(v, c)$ **fi**
R_2 : **if** $same(w, d)$ **and** $same(v, c)$ **then** $add(y, e)$ **fi**
R_3 : **if** $same(v, c)$ **then** $add(z, k)$ **fi**
R_4 : **if** $same(x, j)$ **and** $same(y, e)$ **then** $add(z, h)$ **fi**
R_5 : **if** $same(u, f)$ **and** $same(x, g)$ **then** $add(z, i)$ **fi**

The inference engine starts with the construction of the conflict set: $\{R_3, R_4, R_5\}$. The rule R_3 is the first to be applied. The variable v mentioned in the first condition of

rule R_3 occurs in the conclusion of rule R_1. This is the only rule in the new conflict set. So, rule R_1 is the next rule to be evaluated. Suppose that rule R_1 fails; as a consequence, no value will be inferred for v. The reader can easily verify that given this set of production rules the variables w, x, and u will be asked from the user. The search

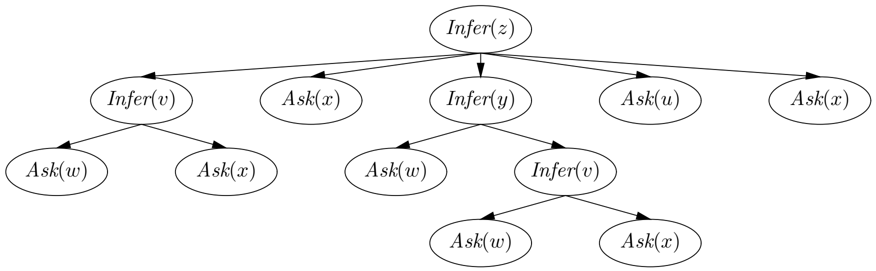

Figure 3.3: A search space generated using backward chaining.

space generated from the set of goals and the rules using backward chaining, takes the form of a tree, as shown in Figure 3.3. The label *Infer* indicates that the variable is inferred from the rule base; *Ask* indicates that the variable is asked from the user. Now note that the mere presence of a fact concerning a specific variable in the fact set, does not guarantee that the variable has actually been traced by exhaustively applying the production rules: the fact may have been entered into the fact set as a side-effect of the application of a production rule having more than one conclusion when tracing another variable. Therefore, the figure shows that the variable v has to be traced twice, even if a fact concerning v already occurs in the fact set. Furthermore, it indicates that several variables have to be asked from the user more than once.

In the preceding example, we showed that the process of tracing a specific variable may be repeated unnecessarily: it will be evident that optimization is required here. We introduce the notion of a 'traced' variable. A variable is marked as *traced* as soon as the process of tracing the variable has been performed, independent of whether it has yielded a value for the variable or not. We now modify the TraceValues procedure as follows:

```
procedure TraceValues(variable)
    Infer(variable);
    if not established(variable) and askable(variable) then
        Ask(variable)
    fi;
    traced(variable) ← true
end
```

Recall that in the foregoing, this procedure was invoked from the EvalConditions procedure. The last-mentioned procedure is now modified in such a way that the TraceValues procedure is invoked for a given variable only if the variable has not yet been marked as traced. This simple refinement has a dramatic effect on the structure of the search space, as is shown in the following example.

EXAMPLE 3.16

Consider once again the production rules from the preceding example, and also the search space for the set of goals and the rule base shown in Figure 3.3. If we exploit the refinement discussed in the foregoing and mark a variable as traced as soon as it has been traced exhaustively, then several inference steps and several questions to the user will have become superfluous. The resulting search space is considerably smaller than the one depicted in Figure 3.3, since now it suffices to represent each vertex only once. The resulting search space therefore has the form of a graph; it is depicted in Figure 3.4. Note that it indicates the dependencies between the variables during the inference.

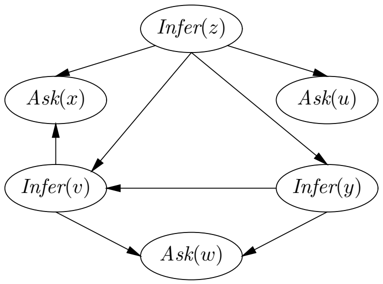

Figure 3.4: A search graph for backward chaining.

If a production rule contains two or more conclusions concerning different variables, then the rule may be applied more than once by the top-down inference algorithm. However, if we assume that production rules do not contain actions removing or modifying facts, then it suffices to apply such a rule only once: applying such a rule a second time cannot result in the addition of new facts to the fact set. To prevent a rule from being applied more than once, each rule will be marked as been *used* as soon as the inference algorithm applies it.

```
procedure Infer(variable)
    Select(rule-base, variable, selected-rules);
    foreach rule in selected-rules with not used(rule) do
        used(rule) ← true;
        Apply(rule)
    od
end
```

Moreover, marking a production rule before it is actually applied, has the further advantage of preventing infinite recursion in case of self-referencing rules.

In the foregoing we have introduced two refinements of the basic backward-chaining (and top-down) algorithm: the marking of a variable after its examination by the inference engine as 'traced', and the marking of a production rule as 'used' upon its application. The last

refinement we pay attention to is the so-called *look-ahead facility*. This facility yields under certain circumstances a remarkable increase in efficiency, by pruning the search space in an effective way. The general structure of the search space, however, is not changed. Let us first study the following example illustrating the need for the look-ahead facility.

EXAMPLE 3.17

Consider the production rule:

if
  $same(x, a)$ **and**
  $same(y, b)$ **and**
  $notsame(z, c)$ **and**
  $lessthan(v, 40)$ **and**
  $greaterthan(w, 90)$
then
  $add(u, 6)$
fi

in which v and w are the only single-valued variables. Now, suppose that the fact $v = 80$ is present in the fact set. This single fact provides us with enough information to deduce that this rule will certainly fail due to its fourth condition. However, the backward-chaining algorithm as introduced in the foregoing, will detect the failure only when at least one (and possibly all three) of the variables x, y, and z from the first three conditions has been traced.

The look-ahead facility now amounts to examining all conditions from a selected production rule before it is actually applied. If a condition is encountered which already fails beforehand using the information from the fact set, then the entire rule will fail; the inference engine proceeds with the next rule. A possible algorithm for the look-ahead facility is shown below:

```
function LookAhead(rule)
    foreach condition in antecedent(rule) do
        var ← variable(condition);
        if traced(var) then
            ExecPredicate(condition);
            if condition failed then
                return(false)
            fi
        fi
    od;
    return(true)
end
```

Note that in the look-ahead facility only conditions specifying a traced variable are examined; it just skips the other conditions. If look ahead returns with success, then the top-down inference algorithm continues as usual. This function LookAhead is called from the procedure

Infer just before calling Apply. The procedure Apply then is invoked only if LookAhead has succeeded.

In the following diagram, the most important procedure and function calls are shown, indicating the level at which they are introduced in the top-down inference algorithm:

```
TraceValues
  Infer
    Select
    LookAhead
      ExecPredicate
    Apply
      EvalConditions
        TraceValues
        ExecPredicate
      EvalConclusions
        ExecAction
  Ask
```

This diagram depicts once more the indirect recursive call to the procedure TraceValues from EvalConditions.

Note that only relatively slight modifications of the discussed algorithm are needed to extend it to an object-attribute-value representation. We only have to add a procedure for tracing the *goal attributes* of an object, a notion similar to the notion of a goal variable:

```
procedure Activate(object)
  foreach attr in attributes(object) do
    if goal(attr) then
      TraceValues(object, attr)
    fi
  od
end
```

The other procedures only need slight alteration by adding the object as an extra argument to the procedures, and by taking for a variable an attribute of the given object.

3.2.2 Bottom-up inference and production rules

Broadly speaking, bottom-up inference with production rules differs from top-down inference only by being controlled by the fact set instead of by goals and subgoals. As in top-down inference, each time a set of relevant production rules is selected from a rule base; the resulting set of applicable production rules in bottom-up inference again is called the conflict set. Recall that in top-down inference, only production rules having a particular (sub)goal variable in one of their conclusions were included in the conflict set. In bottom-up inference, however, a production rule is entered into the conflict set if its conditions are fulfilled using the information from the given fact set. In applying the rules from the conflict set, there is also a difference between the two forms of inference. In top-down inference all production rules from the conflict set are applied exhaustively. In bottom-up inference, however, generally only one of them is applied. This difference arises from the action *remove* being

applied frequently in production systems employing bottom-up inference, in contrast with top-down inference systems. It will be evident that the evaluation of this action may have as an effect that certain conditions, which were true before evaluation of that specific action took place, do not longer hold after its evaluation. As a consequence, all rules in the conflict set have to be reconsidered, since some may specify conditions which fail upon evaluation using the altered fact set. Furthermore, the changes in the fact set may render other rules being successful, which should then be added to the conflict set. Therefore, after applying a single rule from a specific conflict set, a new conflict set is selected. In practice, however, many of the rules previously present in the conflict set will appear again in the new conflict set in the next inference cycle. The entire process is repeated again and again, until some predefined termination criterion is met; a frequently employed criterion is the emptiness of the set of applicable rules. The inference is started just by the presence of initial facts in the fact set. The general approach of bottom-up inference in a production system is described in the following procedure:

```
procedure Infer(rule-base, fact-set)
  rules ← Select(rule-base, fact-set);
  while rules ≠ ∅ do
    rule ← ResolveConflicts(rules);
    Apply(rule);
    rules ← Select(rule-base, fact-set)
  od
end
```

The function Select is applied for selecting the applicable production rules from the rule base. The function ResolveConflicts subsequently chooses from the resulting conflict set a single rule for application: this function implements a conflict-resolution strategy, several of which will be discussed below. The selected production rule is then applied by means of a call to the procedure Apply. In bottom-up inference, the procedure Apply just evaluates the actions of the conclusions specified in the consequent of the rule.

The selection of the relevant production rules from the rule base is described in the following function Select. Again, for ease of exposition, we have assumed that the antecedent of a production rule only comprises a conjunction of conditions:

```
function Select(rule-base, fact-set)
  selected-rules ← ∅;
  foreach rule in rule-base do
    failed ← false;
    foreach cond in antecedent(rule) while not failed do
      EvalCondition(cond, fact-set, failed)
    od;
    if not failed then
      selected-rules ← selected-rules ∪ {rule}
    fi
  od;
  return(selected-rules)
end
```

The procedure EvalCondition evaluates a single condition of a production rule. It returns failure if the variable mentioned in the condition does not occur in the fact set, or if the specified predicate upon evaluation yields the truth value *false*; otherwise the procedure returns success. Note that this evaluation differs from the evaluation described for top-down inference, since here the evaluation of a condition does not lead to the generation of a new subgoal.

The basic bottom-up inference algorithm is very simple. However, much more is still to be said about incorporating control strategies into the basic bottom-up inference scheme. The order in which the rule base is traversed nor the order in which the conditions and conclusions of a selected rule are evaluated has been fixed in the procedure Select. As in Section 3.2.1, we again call the non-fixed order in which production rules are selected and applied, nondeterminism of the first kind, and the non-fixed order in which conditions and conclusions are evaluated, nondeterminism of the second kind. Nondeterminism of the first kind is again resolved by means of a conflict-resolution strategy. These strategies are much more often applied in systems with bottom-up inference than in systems using top-down inference. If rules, and conditions and conclusions of rules, are evaluated in the order in which they have been specified we speak of *forward chaining*; this is the simplest possible form of conflict-resolution. So, in forward chaining the first successful production rule encountered will be selected for application. Note that the choice of the order in which the conditions of the rules from the rule base are evaluated in bottom-up inference has no effect on the resulting behaviour of the system. Only the order in which the rules are applied, and the order in which the conclusions of the rules are evaluated, are of importance.

Many conflict-resolution strategies have been developed for bottom-up inference; here we only discuss three more of them in addition to the earlier mentioned forward chaining. The main reason for augmenting the basic inference algorithm with a conflict-resolution strategy which is more sophisticated than simple forward chaining, is to obtain a more context-sensitive and problem-directed reasoning behaviour, that is, to better control the inference. Since bottom-up inference lacks the 'intelligent' goal-directed nature of top-down inference, conflict-resolution strategies evidently are much more important in bottom-up inference than in top-down inference. Possible conflict resolution strategies differing from forward chaining are:

- conflict resolution by *prioritization*, which for selecting a rule for application uses priorities of the production rules which have been indicated explicitly by the knowledge engineer;
- conflict resolution by *specificity*, which causes the system to prefer more strongly stated production rules over weaker ones;
- conflict resolution by *recency*, which uses the most recently derived facts in selecting a production rule for application, thus causing the system to pursue a single line of reasoning.

Conflict resolution by production rule prioritization has the same advantages as forward chaining: it is easy to implement and use, while being effective in many applications. However, an obvious disadvantage of this strategy is the burden it places on the knowledge engineer who has to impose an ordering on the production rules from the rule base explicitly.

Conflict resolution by specificity is based on some measure of specificity for production rules, such as for example the number of tests specified in the conditions of the rules. A

production rule R_1 is considered to be more specific than a production rule R_2 if R_1 contains at least the same conditions as R_2.

EXAMPLE 3.18

Consider the following two production rules:

R_1 : **if** same(programming-language, symbol-manipulation)
then add(language, AI) **fi**

R_2 : **if** same(programming-language, symbol-manipulation)
and same(syntax, parentheses)
then add(language, Lisp) **fi**

Production rule R_2 is more specific than production rule R_1, since it contains the condition from R_1 as well as some additional conditions. Furthermore, the conclusion of R_2 is more specific than the one from R_1. It will be obvious that success of R_2 will yield a stronger result than success of rule R_1. A specificity strategy will therefore choose R_2 from the conflict set R_1, R_2 for application.

The use of a specificity strategy increases the extensibility of the rule base: a rule base can easily be enlarged by adding new, more specific rules to it without our having to worry too much about older rules, since more specific production rules prevail over the more general ones. Note that most humans exhibit a similar behaviour. For example, when a person encounters his friend in the street, he will not be inclined to think that this other person is a mammal, but he will think instead that it is his friend John: he just applies the most specific knowledge he has.

The last conflict-resolution strategy we pay attention to, the recency strategy, is undoubtedly the most complicated of the ones we have mentioned. This conflict-resolution strategy requires that each fact in the fact set is supplemented with a so-called time tag, a unique number indicating the 'time' the fact was derived. In the following definition, the notion of a fact is redefined for the case of bottom-up inference using the recency strategy. We will only deal with single-valued variables; the extension to multi-valued ones is straightforward.

Definition 3.4 *A* fact *is an expression of the following form:*

$$t : x^s = c$$

where $t \in \mathbb{N}$ *is a* time tag *which uniquely identifies the fact;* x^s *is a single-valued variable, and* c *is a constant. A* fact set *has the following form:*

$$\{t_1 : x_1^s = c_1, \ldots, t_n : x_m^s = c_m\}$$

Constants and variables may now occur more than once in the fact set. However, a time-tag variable pair is unique. Each fact added to the fact set as a result of applying a production rule, is assigned a new time tag $t + 1$ where t is the last assigned one.

EXAMPLE 3.19

Consider the following fact set:

$$\{1 : x = a, 2 : x = b, 3 : y = c, 4 : z = d\}$$

The variable x occurs twice amongst these facts, with different time tags. This should be interpreted as follows: at time 1 the variable x has taken the value a, and at time 2 it has obtained the value b. Therefore, x has two values, one at time 1 and another one at time 2.

There are various ways in which time tags may be interpreted in the representation of facts. Time tags for example may be taken to monitor progress in time of some parameter.

EXAMPLE 3.20

Consider the following fact set:

$$\{1 : temp = 36.2, 2 : temp = 37, 3 : temp = 38\}$$

Herein, time tags are used to indicate the change of the body temperature of some person in time; each time tag for example indicates a day.

We recall that here we introduced time tags in order to enable conflict resolution by recency; this is the main usage of time tags. After the applicable production rules have been selected from the rule base, it is possible to order the conflict set using the time tags associated with the individual facts from the fact set. With each rule in the conflict set a sequence of time tags is associated, where each time tag originates from a fact matching with a condition of the specific rule. These time tags are then sorted in decreasing order. Each thus obtained sequence of time tags is padded with as many as zero time tags as required to make all sequences of equal length. This way, the production rules in the conflict set may be compared to each other.

EXAMPLE 3.21

Consider the following fact set:

$$\{1 : x = a, 2 : x = b, 3 : y = c, 4 : z = d, 5 : w = e, 6 : z = f\}$$

Now, suppose that the conflict set consists of the following three production rules:

R_1: **if** $same(z, f)$ **then** $add(x, e)$ **fi**
R_2: **if** $same(x, b)$ **and** $same(z, d)$ **then** $add(y, f)$ **fi**
R_3: **if** $same(x, a)$ **and** $same(y, c)$ **and** $same(w, e)$ **then** $add(x, d)$ **fi**

Rule R_3 has the largest number of conditions, namely three. So, with each rule we associate a sequence of time tags having length three:

R_1:	6	0	0
R_2:	4	2	0
R_3:	5	3	1

The production rules in the conflict set are now ordered according to the lexicographical order of their associated sequences of time tags: in the ordering, a rule R_1 precedes a rules R_2 if the sequence of time tags associated with R_1, read from left to right, is larger than the one associated with rule R_2. The order relation between members of the conflict set is denoted by the symbol $\geq$. The relation $\geq$ is a total ordering, and has therefore the following four properties:

- *Reflexivity:* for each production rule R we have $R \geq R$;
- *Transitivity:* for each three production rules R_1, R_2, and R_3, satisfying $R_1 \geq R_2$ and $R_2 \geq R_3$, we have that $R_1 \geq R_3$;
- *Anti-symmetry:* for each pair of rules satisfying $R_1 \geq R_2$ and $R_2 \geq R_1$, we have that $R_1 = R_2$;
- *Totality:* for each pair of production rules R_1 and R_2, we have either $R_1 \geq R_2$ or $R_2 \geq R_1$.

EXAMPLE 3.22

Consider the following sequences of time tags, belonging to four production rules R_1, R_2, R_3 and R_4, respectively:

R_1:	6	0	0
R_2:	4	3	2
R_3:	5	3	1
R_4:	6	1	0

For these rules we have that $R_4 \geq R_1 \geq R_3 \geq R_2$.

This ordering of the rules from a conflict set enables us to give an algorithm for conflict resolution based on recency; it is described in the following function ResolveConflicts:

```
function ResolveConflicts(rules)
  if rules = ∅ then return(∅)
  else
    r ← Max-Time-tag-Subset(rules);
    if r is singleton then return(r)
    else
      result ← ResolveConflicts(r)
    fi;
    if result = ∅ then return(first(r))
    else
      return(result)
    fi
  fi
end
```

The function Max-Time-tag-Subset called from ResolveConflicts selects from the conflict set rules the rules with the highest time tag. If the resulting set r contains more than one element, then, after the earlier examined time tags have been skipped, the function ResolveConflicts is called recursively for this set r. Note that contrary to what has been described before, the conflict set is not ordered entirely before it is examined: each time only a subset of the rules relevant for conflict resolution is selected. The presented algorithm does not always yield a single production rule, since it is possible to have two or more (different) production rules, having the same sequence of time tags. In this case, on arbitrary grounds the first specified production rule is returned as a result.

As has been discussed above, a variable specified in a condition of a production rule may have more than one occurrence in the fact set, although with different time tags. As a consequence, a condition may match with more than one fact. So, a production rule may be applied more than once, using different facts.

EXAMPLE 3.23

Consider the following fact set:

$$F = \{1 : x = a\}$$

and the conflict set consisting of the following two production rules:

R_1: **if** $same(x, a)$ **then** $add(y, b)$ **fi**
R_2: **if** $same(y, b)$ **then** $add(y, b)$ **fi**

Then, the application of rule R_1 in the first inference step results in the following modified fact set:

$$F' = \{1 : x = a, 2 : y = b\}$$

Subsequent application of rule R_2 yields the following fact set:

$$F'' = \{1 : x = a, 2 : y = b, 3 : y = b\}$$

Rule R_2 can now be applied again. In this example, the inference will not terminate; rule R_2 will be applied forever.

In the preceding example, we have shown that a production rule may be applied more than once, using different facts. It is therefore necessary to specify in the conflict set all possible *applications* of a rule, instead of the rule itself. For this purpose, we introduce the notion of a rule instance.

Definition 3.5 *Let F be a fact set, and R a production rule. Let $M \subseteq F$ be a subset of facts, such that each element $f \in M$ matches with a condition of R, and each condition of the production rule R matches with an element from M. Then, the pair (R, M) is called a* rule instance.

In other words, a rule instance consists of a production rule and the facts matching with its conditions. It will be evident that although production rules may be applied several times, it is undesirable that rule instances are applied more than once. Note that the conflict set should now be taken as a set of rule instances. The basic algorithm for bottom-up inference discussed in the foregoing has to be altered for dealing with such rule instances. Recall that so far, we have treated four procedures which together constitute the bottom-up inference algorithm:

- the procedure Infer, which described the global inference process;
- the function Select, which was used for the selection of applicable rules from the rule base;
- the function ResolveConflicts, which specified the conflict-resolution method;
- the procedure Apply, which applied the once selected production rule.

First, we reconsider the procedure Infer. Here, we have to record the rule instances which have been applied, to prevent rule instances from being applied more than once:

```
procedure Infer(fact-set, rule-base)
  applied-instances ← ∅;
  instances ← Select(rule-base, applied-instances, fact-set);
  while instances ≠ ∅ do
    instance ← ResolveConflicts(instances);
    Apply(instance);
    applied-instances ← applied-instances ∪ {instance};
    instances ← Select(rule-base, applied-instances, fact-set)
  od
end
```

The function Select now has to generate rule instances from the production rules in the rule base instead of the production rules themselves:

```
function Select(rule-base, applied, fact-set)
  selected-instances ← ∅;
  foreach rule in rule-base do
    failed ← false;
    rule-instances ← (rule, ∅);
    foreach cond in antecedent(rule) while not failed do
      ModifyInstances(cond, rule-instances, fact-set, failed)
    od;
    if not failed then
      selected-instances ← selected-instances ∪
                           (rule-instances \ applied)
    fi
  od;
  return(selected-instances)
end
```

Note that the second argument of Select now contains a set of rule instances. The procedure ModifyInstances called from Select, evaluates a given condition cond using the fact set. Each different matching fact gives rise to the creation of a new rule instance. Of course, if a condition fails then no rule instance will be created. Since the rule instances are built recursively, evaluation of subsequent conditions of a production rule may lead to discarding rule instances under construction from the set rule-instances:

```
procedure ModifyInstances(condition, rule-instances, fact-set, failed)
  relevant-facts ← EvalCondition(condition, fact-set);
  failed ← relevant-facts = ∅;
  new-instances ← ∅;
  if not failed then
    foreach fact in relevant-facts do
      foreach rule-inst in rule-instances do
        new-instances ← new-instances ∪ Add(rule-inst, fact)
      od
    od
  fi;
  rule-instances ← new-instances
end
```

After the conflict set has been created, the next step in Infer is to select a single rule instance from it by conflict resolution. This is achieved by means of the procedure ResolveConflicts which already has been discussed. Finally, the procedure Apply is called to evaluate the conclusions of the selected rule instance. Recall that evaluation of the action *add* adds a new fact to the fact set, which is assigned a new time tag. Evaluation of the action *remove* deletes a fact from the fact set; the fact to be deleted is selected either by explicitly referring to its time tag, or simply by matching.

We conclude this section with an example.

EXAMPLE 3.24

Consider the following fact set F:

$$F = \{1 : x = a, 2 : x = b, 3 : y = 4\}$$

Furthermore, let us have the following set of production rules:

R_1: **if** $same(x, a)$ **and** $same(x, b)$ **then** $add(z, e)$ **fi**
R_2: **if** $same(z, e)$ **and** $same(w, g)$ **then** $add(z, f)$ **fi**
R_3: **if** $lessthan(y, 10)$ **and** $same(x, a)$ **or** $same(x, b)$ **then** $add(w, g)$ **fi**

The given fact set F gives rise to the creation of the following rule instances, together constituting the conflict set:

$(R_1, \{1 : x = a, 2 : x = b\})$
$(R_3, \{3 : y = 4, 1 : x = a\})$
$(R_3, \{3 : y = 4, 2 : x = b\})$

Note that two rule instances of R_3 have been created. Using the recency conflict-resolution strategy, the second instance of R_3 is selected for evaluation, since the time tag of its second matching fact is larger than the time tag of the second matching fact of the first instance of R_3. Evaluation of the instance $(R_3, \{3 : y = 4, 2 : x = b\})$ causes the fact $4 : w = g$ to be added to the fact set, resulting in:

$$F' = \{1 : x = a, 2 : x = b, 3 : y = 4, 4 : w = g\}$$

The inference is now repeated; the instance $(R_3, \{3 : y = 4, 2 : x = b\})$ however is no longer applicable.

The algorithm discussed in this section provides a more or less complete description of the bottom-up inference method. However, an inference engine implementing this algorithm will be quite inefficient, since at every inference step all instances are created all over again. A first step towards improving the efficiency of the algorithm is to save the instances between two consecutive inference steps. Such an algorithm already exists, it is called the *rete algorithm*, and has been developed by C.L. Forgy as part of the system OPS5. We shall return to this rete algorithm and OPS5 in Chapter 8.

3.3 Production rules as a representation formalism

In this chapter we have discussed several forms of knowledge representation and inference used in production systems. Various attempts in the past in using production rules for building intelligent systems have proven the production system approach to be a flexible one, and is suitable for many problem areas. In fact, many of the intelligent systems mentioned in chapter 1 are examples of rule-based systems. In addition, several large intelligent systems have been and still are being developed using the techniques discussed in this chapter. However, some disadvantages and restrictions of the formalism have also been recognized:

- Descriptive knowledge cannot be represented in the formalism in a natural way. An example of descriptive knowledge has been given in chapter 1 where we described the cardiovascular system. We shall see in the following chapter that the frame formalism is much more suitable for representing this type of knowledge.

- The different types of knowledge encountered in a problem area, such as problem-dependent knowledge, problem-independent knowledge, and knowledge used for exerting control on the inference process (often called meta-knowledge) have to be expressed using one and the same formalism and therefore cannot be distinguished explicitly.

- The production rule formalism has a strong operational flavour. As a consequence, some knowledge of the underlying execution model of the inference engine is required for adequately representing a problem domain in a knowledge base. Compare this situation with the one in logic, where no knowledge concerning inference is required for specifying a correct knowledge base; familiarity with the declarative semantics of logic suffices in this case.

- A more involved application generally leads to a large rule base, which is difficult to develop and maintain. So, for developing large applications the necessity of partitioning a large rule base into smaller modules arises. However, the production rule formalism offers no direct means for explicitly indicating and exploiting such a modularization.

Some of these problems may be solved by combining production rules with some other formalism, such as for example a frame formalism, or with other programming paradigms such as object-oriented programming. Chapter 8 discusses several solutions that have been proposed and incorporated in actual systems.

Exercises

(3.1) Consider the knowledge base containing the following domain declaration

$$D = \{x_a^s, y^s, z_a^s, w_g^m\}$$

and the following rule base:

$\{R_1$: **if** *lessthan*$(z, 20)$ **and** *notknown*(y) **then** *add*(w, b) **fi**,
R_2: **if** *same*(x, c) **and** *known*(y) **then** *add*(w, d) **fi**,
R_3: **if** *notsame*(x, b) **and** *greaterthan*$(z, 100)$ **then** *add*(y, f) **fi**$\}$

The variables x_a^s, y^s and z_a^s are single-valued, and the variable w_g^m is multi-valued. As can be seen, w_g^m is a goal variable, and the variables x_a^s and z_a^s are askable. Furthermore, let the following fact set be given:

$$F = \{x = c, z = 5\}$$

(a) Determine the fact set F' which results from applying backward chaining on this set of production rules. Which production rules will have succeeded, and which ones have failed?

(b) Suppose that the following self-referencing production rule is added to the three rules listed above:

R_4: **if** *notknown*(y) **then** *add*(y, A) **fi**

Again we employ backward chaining, starting with the same initial fact set F as given above. Which fact set F'' do we now obtain?

(3.2) Has the choice of the conflict-resolution strategy employed in top-down inference with the look-ahead facility, any effect on the values inferred for a multi-valued goal variable? Motivate your answer. Now answer the same question for an arbitrary (subgoal) variable.

(3.3) Consider the following knowledge base, containing the domain declaration

$$D = \{x^s, y^s, z^s, u^s\}$$

and the rule base:

$\{R_1$: **if** *same(x,a)* **and** *known*(y) **then** *add*(y, b) **fi**,
R_2: **if** *same*(x, c) **and** *lessthan*$(z, 15)$ **then** *add*(u, d) **fi**,
R_3: **if** *same*(y, b) **and** *lessthan*$(z, 5)$ **then** *add*(u, f) **fi**$\}$

Furthermore, consider the following fact set:

$$F = \{1 : x = a, 2 : y = b, 3 : z = 10, 4 : x = c\}$$

Bottom-up inference is employed to derive new facts from the given rule base and the facts present in the fact set.

(a) Give all rule instances created by matching the initial fact set F and the rule base given above. This set of instances is the conflict set.

(b) Order the conflict set obtained in (a), using conflict resolution by recency. Which one of the rule instances will then be selected for application?

(c) Give the fact set which results after evaluation of the rule instance chosen in (b). Will the inference eventually terminate? Motivate your answer by giving the successive changes which take place in the fact set.

(3.4) Consider the following knowledge base, containing the domain declaration

$$D = \{x^s, y^s, z^s\}$$

and the rule base:

$\{R_1$: **if** *same*(x, a) **and** *equal*$(y, 10)$ **then** *add*(z, b) **fi**,
R_2: **if** *same*(z, c) **and** *lessthan*$(y, 20)$ **then** *add*(x, a) **fi**$\}$

Furthermore, consider the following fact set:

$$F = \{1 : x = A, 2 : z = C, 3 : y = 10, 4 : z = C\}$$

Bottom-up inference is employed for inferring new facts from the given rule base and the fact set.

(a) Which rule instances will be created in the first inference step, and which one of these will be selected for application if we apply conflict resolution by recency? Give the new fact set obtained after evaluation of the chosen rule instance. Will the inference eventually terminate?

(b) Suppose that we add the following production rule to the ones shown above, before consulting the knowledge base.

R_3: **if** *same*(z, b) **then** *add*(z, c) **fi**

We start with the same inititial fact set F as in (a). Give the fact set that eventually results. Will the inference terminate? Explain your answer by comparing the results with those of (a).

(3.5) Recall that two types of nondeterminism are distinguished in production systems due to the need to specify: (1) the order in which applicable rules are selected from the rule base, and (2) the order in which conditions and conclusions are evaluated. If we take a particular conflict-resolution strategy in bottom-up inference, which choice(s) for resolving nondeterminism do(es) influence the behaviour of the system?