does not allow clauses with variables ranging over an infinite number of predicates, functions, assertions and sentences (e.g., “All unary functions are boring” cannot be stated directly). This problem may be ameliorated by a notational trick; the situations under which predicates are true are indicated with a Holds predicate. Thus instead of writing On(block1, surface, situation1), write Holds (On(block1, surface), situation1). This notation allows inferences about many situations with only one added axiom. The “situational calculus” reappears in Section 12.3.1. Another useful notational trick is a Diff relation, which holds between two terms if they are syntactically different. There are infinitely many axioms asserting that terms are different; the actual system can be made to incorporate them implicitly in a well-defined way. The Diff relation is also used in Section 12.3.1.

3. The frame problem (so called for historical reasons and not related to the frames described in Section 10.3.1) is a classic bugbear of problem-solving methods including predicate logic. One aspect of this problem is that for technical reasons, it must be explicitly stated in axioms that describe actions (in a general sense a visual test is an action) that almost all assertions were true in a world state remain true in the new world state after the action is performed. The addition of these new axioms causes a huge increase in the “bureaucratic overhead” necessary to maintain the state of the world. Currently, no really satisfactory way of handling this problem has been devised. The most common way to attack this aspect of the frame problem is to use explicit “add lists” and “delete lists” ([Fikes 1977], Chapter 13) which attempt to specify exactly what changes when an action occurs. New true assertions are added and those that are false after an action must be deleted. This device is useful, but examples demonstrating its inadequacy are readily constructed. More aspects of the frame problem are given in Chapter 13.

4. There are several sorts of reasoning performed by human beings that predicate logic does not pretend to address. It does not include the ability to describe its own formulae (a form of “quotation”), the notion of defaults, or a mechanism for plausible reasoning. Extensions to predicate logic, such as modal logic, are classically motivated. More recently, work on extensions addressing the topics above have begun to receive attention [McCarthy 1978; Reiter 1978; Hayes 1977]. There is still active debate as to whether such extensions can capture many important aspects of human reasoning and knowledge within the model-theoretic system. The contrary view is that in some reasoning, the very process of reasoning itself is an important part of the semantics of the representation. Examples of such extended inference systems appear in the remainder of this chapter, and the issues are addressed in more detail in the next section.

12.2 COMPUTER REASONING

Artificial intelligence in general and computer vision in particular must be concerned with efficiency and plausibility in inference [Winograd 1978]. Computer-based knowledge representations and their accompanying inference processes often sacrifice classical formal properties for gains in control of the inference process and for flexibility in the sorts of “truth” which may be inferred.
Automated inference systems usually have inference methods that achieve efficiency through implementational, computation-based, inference criteria. For example, truth may be defined as a successful lookup in a data base, falsity as the failure to find a proof with a given allocation of computational resources, and the establishment of truth may depend on the order in which deductions are made.

The semantics of computer knowledge representations is intimately related to the inference process that acts on them. Therefore, it is possible to define knowledge representations and interpreters in computers whose properties differ fairly radically from those of classical representations and proof procedures, such as the first-order predicate calculus. For instance, although the systems are deterministic, they may not be formally consistent (loosely, they may contain contradictory information). They may not be complete (they cannot derive all true theorems from the givens); it may be possible to prove $P$ from $Q$ but $\neg P$ from $Q$ and $R$. The set of provable theorems may not be recursively enumerable [Reiter 1978]. Efforts are being made to account for the “extended inference” needed by artificial intelligence using more or less classical logic [McCarthy 1978; Reiter 1978; Hayes 1977; 1978a; 1978b; Kowalski 1974, 1979]. In each case, the classical view of logic demands that the deductive process and the deducible truths be independent. On the other hand, it is reasonable to devote attention to developing a nonclassical semantics of these inference processes; this topic is in the research stage at this writing.

Several knowledge representations and inference methods using them are “classical” in the artificial intelligence world; that is, they provide paradigmatic methods of dealing with the issues of computational inference. They include STRIPS [Fikes and Nilsson 1971], the situational calculus [McCarthy and Hayes 1969], PLANNER and CONNIVER [Hewitt 1972; Sussman and McDermott 1972], and semantic net representations [Hendrix 1979; Brachman 1979].

To illustrate the issue of consistency, and to illustrate how various sorts of propositions can be represented in semantic nets, we address the question of how the order of inference can affect the set of provable theorems in a system.

Consider the semantic net of Fig. 12.3. The idea is that in the absence of specific information to the contrary, one should assume that railroad bridges are narrow. There are exceptions, however, such as Bridge02 (which has a highway bridge above the rail bridge, say). The network is clearly inconsistent, but trouble is avoided if inferences are made “from specific to general.” Such ordering implies that the system is incomplete, but in this case incompleteness is an advantage.

Simple ordering constraints are possible only with simple inferential powers in the system [Winograd 1978]. Further, there is as yet little formal theory on the effects of ordering rules on computational inference, although this has been an active topic [Reiter 1978].

12.3 PRODUCTION SYSTEMS

The last section explored why the process of inference itself could be an important part of the semantics of a knowledge representation system. This idea is an impor-
tant part of production systems. Perceived limitations in logic inference mechanisms and the seductive power of arbitrary algorithmic processes for inference has spawned the development of rule-based systems which differ from first-order logic in the following respects:

- Arbitrary additions and deletions to the clausal data base are allowed.
- An interpreter that controls the inference process in special ways is usually an integral part of the system.

Early examples of systems with the first addition are STRIPS [Fikes and Nilsson 1971] and PLANNER [Hewitt 1972]. Later examples of systems with both additions are given in [Waterman and Hayes-Roth 1978]. The virtues of trying to control inferences may be appreciated after our brief introduction to clausal automatic theorem proving, where there are no very good semantic heuristics to guide inferences. However, the price paid for restricting the inference process is the loss of formal properties of consistency and correctness of the system, which are not guaranteed in rule-based systems. We shall look in some detail at a particular form of rule-based inference system called production systems.

A production system supports a general sort of "inference." It has in common with resolution that matching is needed to identify which inference to make. It is different in that the action upon finding a matching data item is less constrained. Actions of arbitrary complexity are allowed. A production system consists of an explicit set of situation-action nodes, which can be applied against a data base of situations. For example, in a very constrained visual domain the rule

\[(\text{Green (Region } X\text{)}) \rightarrow (\text{Grass (Region } X\text{)})\]  \hspace{1cm} (12.11)

could infer directly the interpretation of a given region. Segmentation rules can also be developed; the following example merges two adjacent green regions into a single region.
\[(\text{Green(Region } X) \land \text{Green(Region } Y) \land \text{Adjacent(Region } X, \text{Region } Y)) \rightarrow (\text{Green(Region } Z) \land ((\text{Region } Z) := \text{Union(Region } X, \text{Region } Y)))\]

These examples highlight several points. The first is that the basic idea of production systems is simple. The rules are easy to “read” by both the programmer and the program, and new rules are easily added. Although it is imaginable that “situations” could extend down to the pixel level, and production systems could be used (for instance) to find lines, the system overhead would render such an approach impractical. In the visual domain, the production system usually operates on the segmented image (Chapters 4 and 5) or with the high-level internal model. In the rules above, X and Y are variables that must be bound to specific instances of regions in a database. This process of binding variables or matching can become very complex, and is one of the two central issues of this kind of inference. The other is how to choose rules from a set of whose situations match the current situation to some degree.

### 12.3.1 Production System Details

In its simplest form a production system has three basic components:

1. A database
2. A set of rules
3. An interpreter for the rules

The vision database is usually a set of facts that are known about the visual environment. Often the rules are considered to be themselves a manipulable part of the database. Examples of some visual facts may be

\[(\text{ABOVE (Region } 5) \text{ (Region } 10))\]

\[(\text{SIZE (Region } 5) 300)\]

\[(\text{SKY (Region } 5))\]

\[(\text{TOP (Region } 5) 255)\]  \hspace{2cm} (12.12)

The database is the sole storage medium for all state variables of the system. In particular, unlike procedurally oriented languages, there is no provision for separate storage of control state information—no separate program counter, push-down stack, and so on [Davis and King 1975].

A rule is an ordered pair of patterns with a left-hand side and a right-hand side. A pattern may involve only database primitives but usually will have variables and special forms as subpatterns which are matched against the database by the interpreter. For example, applying the following rule to a database which includes (12.12),
(TOP (Region X) (GreaterThan 200))

→

(SKY (Region X))

region 5 can be inferred to be sky. The left-hand side matches a set of data-base facts and this causes (SKY (Region 5)) to be added to the data base. This example shows the kinds of matching that the interpreter must do: (1) the primitive TOP in the data base fact matches the same symbol in the rule, (2) (Region X) matched (Region 5) and X is bound to 5 as a side effect, and (3) (GreaterThan 200) matches 255. Naturally, the user must design his own interpreter to recognize the meaning of such operational subpatterns.

However, even the form of the rules outlined so far is relatively restrictive. There is no reason why the right-hand side cannot do almost arbitrary things. For instance, the application of a rule may result in various productions being deleted or added from the set of productions; the data base of productions and assertions thus can be adaptive [Waterman and Hayes-Roth 1978]. Also, the right-hand side may specify programs to be run which can result in facts being asserted into the data base or actions performed.

Control in a basic production system is relatively simple: Rules are applied until some condition in the data base is reached. Rules may be applied in two distinct ways: (1) a match on the left-hand side of a rule may result in the addition of the consequences on the right-hand side to the data base, or (2) a match on the right-hand side may result in the addition of the antecedents in the left-hand side to the data base. The order of application of rules in the first case is termed *forward chaining* reasoning, where the objective is to see if a set of consequences can be derived from a given set of initial facts. The second case is known as *backward chaining*; the objective is to determine a set of facts that could have produced a particular consequence.

### 12.3.2 Pattern Matching

In the process of matching rules against the data base, several problems occur:

- Many rule situations may match data base facts
- Rules designed for a specific context may not be appropriate for larger context
- The pattern matching process may become very expensive
- The data base or set of rules may become unmanageably large.

The problem of multiple matches is important. Early systems simply resolved it by scanning the data base in a linear fashion and choosing the first match, but this is an ineffective strategy for large data bases, and has conceptual problems as well. Accordingly, strategies have evolved for dealing with these conflicts. Like most inference-controlling heuristics, their effectiveness can be domain-dependent, they can introduce incompleteness into the system, and so on.

On the principle of *least commitment*, when there are many chances of errors, one strategy is to apply the most general rule, defined by some metric on the com-
ponents of the pattern. One simple such metric is the number of elements in a pattern. Antithetical to this strategy is the heuristic of applying the most specific pattern. This may be appropriate where the likelihood of making a false inference is small, and where specific actions may be indicated (match (MAD DOG) with (MAD DOG), not with (DOG)). Another popular but inelegant technique is to exercise control over the order of production application by using state markers which are inserted into the data base by right-hand sides and looked for by left-hand sides.

1. \( A \rightarrow B \land <\text{marker 1}>. \)
2. \( A \rightarrow B \land <\text{marker 2}>. \)
3. \( B \land <\text{marker 1}> \rightarrow C. \)
4. \( B \land <\text{marker 2}> \rightarrow D. \)

Here if rule 1 is executed, "control goes to rule 3," i.e., rule 3 is now executable, whereas if rule 2 is applied, "control goes to rule 4." Similarly, such control paradigms as subroutining, iteration and co-routining may be implemented with production systems [Rychner 1978].

The use of connectives and special symbols can make matching become arbitrarily complex. Rules might be interpreted as allowing all partial matches in their antecedent clauses [Bajcsy and Joshi 1978]. Thus

\[(A\ B\ C) \rightarrow (D)\]

is interpreted as

\[(ABC) \lor (BC) \lor (AB) \lor (AC) \lor (A) \lor (B) \lor (C) \rightarrow (D)\]

where the leftmost actual match is used to compare the rule to others in the case of conflicts.

The problem of large data bases is usually overcome by structuring them in some way so that the interpreter applies the rules only to a subset of the data base or uses a subset of the rules. This structuring undermines a basic principle of pure rule-based systems: Control should be dependent on the contents of the data base alone. Nevertheless, many systems divide the data base into two parts: an active smaller part which functions like the original data base but is restricted in size, and a larger data base which is inaccessible to the rule set in the active smaller part. "Meta-rules" have actions that move situation-action rules and facts from the smaller data base to the larger one and vice versa. The incoming set of rules and facts is presumably that which is applicable in the context indicated by the situation triggering the meta-rule. This two-level organization of rules is used in "blackboard" systems, such as Hearsay for speech-understanding [Erman and Lesser 1975]. The meta-rules seem to capture our idea of "mental set," or "context," or "frame" (Section 10.3.1, [Minsky 1975]). The two data bases are sometimes referred to as short-term memory and long term memory, in analogy with certain models of human memory.
12.3.3 An Example

We shall follow the actions of a production system for vision [Sloan 1977; Sloan and Bajcsy 1979]. The intent here is to avoid a description of all the details (which may be found in the References) and concentrate on the performance of the system as reflected by a sample of its output. The program uses a production system architecture in the domain of outdoor scenes. The goal is to determine basic features of the scene, particularly the separation between sky and ground. The interpreter is termed the "observer" and the memory has a two-tiered structure: (1) short term memory (STM) and (2) long term memory (LTM), a data base of all facts ever known or established, structured to prefer access to the most recently used facts. The image to be analyzed is shown in Fig. 12.4, and the action may be followed in Fig. 12.5. The analysis starts with the initialization command

*(look 100000 100 nil)

This command directs the Observer to investigate all regions that fall in the size range 100 to 100000, in decreasing order of size. The LTM is initialized to NIL.

our first look at (region 11)

\[
x \quad y \quad r-g \quad y-b \quad w-b \quad \text{size} \quad \text{top} \quad \text{bottom} \quad \text{left} \quad \text{right} \\
35 \quad 2 \quad 24 \quad 29 \quad 6 \quad 2132 \quad 35 \quad 97 \quad 2 \quad 127
\]

This report is produced by an image-processing procedure that produces assertions about (region 11). This region is shown highlighted in Fig. 12.5c.

__________ Progress Report __________

regions on this branch:
(11)
context stack:

![Outdoor scene to be analyzed with production system.](image)

Fig. 12.4 Outdoor scene to be analyzed with production system.
Fig. 12.5 Images corresponding to steps in production system analysis. (a) Texture in the scene. (b) Region 11 outlined. (c) Sky-Ground separation. (d) Skyline.

nil
contents of short term memory:
((far-left (region 11)) (far-right (region 11))
(right (region 11) 127) (left (region 11) 2)
(bottom (region 11) 97) (top (region 11) 35)
(w-b (region 11) minus) (y-b (region 11) zero)
(r-g (region 11) zero) (size (region 11) 2132))

end of progress report

Note that gray-level information is represented as a vector in opponent color space (Chapter 2), where the components axes are WHITE-BLACK \((w-b)\), RED-GREEN \((r-g)\), and YELLOW-BLUE \((y-b)\). Three values (plus, zero, minus) are used for each component. The display above is generated once after every iteration of the Observer. The report shows that \((REGION 11)\) is being investigated; there is no known context for this investigation; the information about \((REGION 11)\) created by the image-processing apparatus has been placed in STM. The context stack is for information only, and shows a trace of activated sets of rules.
i think that (far-left (region 11))
i think that (far-right (region 11))
i think that (right (region 11) 127)
i think that (left (region 11) 2)
i think that (bottom (region 11) 97)
i think that (top (region 11) 35)
i think that (size (region 11) 2132)

This portion of the trace shows assertions moving from STM to LTM. They are reported because this is the first time they have been REMEMBERed (a special procedure in the Observer).

---------- Progress Report ----------

regions on this branch:
(11)
context stack:
nil
contents of short term memory:
((color (region 11) black))

---------- end of progress report ----------

The assertions created from the region data structure have been digested, and lead only to the conclusion that (REGION 11) is BLACK, based on a production that looks like:

\[(w-b (region x) \text{ minus}) \wedge (r-w (region x) \text{ zero})
\wedge (b-w (region x) \text{ zero}) \rightarrow (color (region x) \text{ black})\]

---------- Progress Report ----------

regions on this branch:
(11)
context stack:
nil
contents of short term memory:
((ground (region 11)) (shadow (region 11)))

---------- end of progress report ----------

The observer knows that things that are black are GROUND and SHADOW. The facts it deduces about region 11 are again stored in the LTM.

Having discovered a piece of ground, the Observer has activated the GROUND-RULES, and changed context. It now investigates the neighbors of (REGION 11).

our first look at (region 16)

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>r-g</td>
<td>y-b</td>
<td>w-b</td>
<td>size</td>
<td>top</td>
<td>bottom</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>------</td>
<td>-----</td>
<td>--------</td>
</tr>
<tr>
<td>58</td>
<td>2</td>
<td>23</td>
<td>30</td>
<td>3</td>
<td>1833</td>
<td>57</td>
<td>119</td>
</tr>
</tbody>
</table>

Sec. 12.3  Production Systems  403
(REGION 16) is a neighbor of (REGION 11), and the observer is trying to determine whether or not they are sufficiently similar, in both color and texture, to justify merging them.

_______ Progress Report _______

regions on this branch:
(16 11)
context stack:
(ground)
contents of short term memory:
( ((texture-difference (region 16) (region 11)))
  (color-similar (region 16) (region 11))
  (distance (region 16) near) (ground (region 16))
  (color (region 16 black))

_________ end of progress report _________

The Observer decides that (REGION 16) is ground because it is at the bottom of the picture.

The ground-growing process continues, until finally one of the neighbors of a ground region is a piece of sky. The Observer will not immediately recognize this region as sky, but will see that a depth discontinuity exists and that the border between these two regions represents a section of three dimensional skyline.

our first look at (region 8)

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>r-g</th>
<th>y-b</th>
<th>w-b</th>
<th>size</th>
<th>top</th>
<th>bottom</th>
<th>left</th>
<th>right</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>2</td>
<td>13</td>
<td>13</td>
<td>33</td>
<td>394</td>
<td>15</td>
<td>38</td>
<td>2</td>
<td>57</td>
</tr>
</tbody>
</table>

_______ Progress Report _______

regions on this branch:
(8 13 16 11)
context stack:
(ground ground ground)
contents of short term memory:
( (new-neighbor (region 800) (far-left (region 8))
  (right (region 8) 57) (left (region 8) 2) (bottom (region 8) 38)
  (top (region 8) 15) (w-b (region 8) zero) (y-b (region 8) minus)
  (r-g (region 8) minus) (size (region 8) 394))

_________ end of progress report _________

texture descriptors for (region 8) are (54 50)
texture descriptors for (region 13) are (44 51)

Texture measurement is appropriate in the context of ground areas.
regions on this branch:
(8 13 16 11)
context stack:
(ground ground ground)
contents of short term memory:

((texture-similar (region 8) (region 13)) (color-difference
(region 8) (region 13)) (color (region 8) blue-green))

------------- end of progress report -------------

(REGION 8) passes the texture similarity test, but fails the color match.

regions on this branch:
(8 13 16 11)
context stack:
(ground ground ground)
contents of short term memory:

((darker (region 13) (region 8)) (brighter (region 8) (region 13))
yellower (region 13) (region 8)) (bluer (region 8) (region 13))
(redder (region 13) 13)
(below (region 13) (region 8)) (above (region 8) (region 13)))

------------- end of progress report -------------

checking the border between (region 13) and (region 8)

------------- Progress Report -------------

regions on this branch:
(8 13 16 11)
context stack:
(skyline ground ground ground)
contents of short term memory:

((segments built) (skyline-segment ((117 42)) (region 13)
(region 8)) (skyline-segment ((14 40) (13 40)) (region 13)
(region 8)))

------------- end of progress report -------------

------------- Progress Report -------------

regions on this branch:
(8 13 16 11)
context stack:
(skyline ground ground ground)
contents of short term memory:
((peak (14 40)) (peak (17 42)))

_________ end of progress report _________

Two local maxima have been discovered in the skyline. On the basis of a depth judgment, these peaks are correctly identified as treetops.

The analysis continues until all the major regions have been analyzed. The sky-ground separation is shown in Fig. 12.5a and skyline in Fig. 12.5e.

In most cases, complete analysis of the image follows from the context established by the first (largest) region. This implies that initial scanning of such scenes can be quite coarse, and very simple ideas about gross context are enough to get started. Once started, inferences about local surroundings lead the Observer’s attention over the entire scene, often returning many times to the same part of the image, each time with a bit more knowledge.

### 12.3.4 Production System Pros and Cons

In their pure form, the productions of production systems are completely “modular,” and are themselves independent of the control process. The data base of facts, or situations, is unordered set accessed in undetermined order to find one matching some rule. The rule is applied, and the system reports the search for a matching situation and situation-action pair (rule). This completely unstructured organization of knowledge could be a model for the human learning of “facts” which become available for use by some associative mechanism that finds relevant facts in our memories. The hope for pure production systems is that performance will degrade noncatastrophically from the deletion of rules or facts, and that the rules can interact in synergistic and surprising ways. A learning curve may be simulated by the addition of productions. Thus one is encouraged to experiment with how knowledge may best be broken up into disjoint fragments that interact to produce intelligent behavior.

Together with the modularity of productions in a simple system, there is a corresponding simplicity in the overall control program. The pure controller simply looks at the data base and somehow finds a matching situation (left-hand side) among the productions, applies the rule, and cycles. This simple structure remains constant no matter how the rules change, so any nondeterminism in the performance arises from the matcher, which may find different left hand side matches for sets of assertions in the data base.

The productions usually have a syntax that is machine-readable. Their semantics is similarly constrained, and so it begins to seem hopeful that a program (perhaps fired up by a production) could reason about the rules themselves, add them, modify them, or delete them. This is in contrast to the situation with procedurally embedded knowledge (Section 10.1.3), because it is difficult or impossible for programs to answer general questions about other programs. Thus the claim is that a production system can more easily reason about itself than can many other knowledge representation systems.
Productions often interact in ways that are not foreseen. This can be an advantage or a drawback, depending on the behavior desired. The pattern-matching control structure allows knowledge to be used whenever it is relevant, not only when the original designer thought that it might be. Symbiotic interaction of knowledge may also produce unforeseen insights. Production systems are a primary tool of knowledge engineering, an enterprise that attempts to encode and use expert knowledge at such tasks as medical diagnosis and interpretation of mass spectrograms [Lindsay et al. 1980; Buchanan and Mitchell 1978; Buchanan and Feigenbaum 1978; Shortliffe 1976; Aikins 1980].

There are many who are not convinced that production systems really offer the advantages they initially seem to. They use the following sorts of arguments.

The pure form of production system is almost never seen doing anything useful. In particular, the production system is most naturally a forward-chaining inference system, and one must exercise restraints and guidelines on it to keep it from running away and deducing lots of irrelevant facts instead of doing useful work. Of course, production systems may be written to do backward chaining by hypothesizing a RHS and seeing which LHS must be true for the desired RHS to occur (the process may be iterated to any depth). In practical systems based on production systems, there is implicit or explicit ordering of production rules so the matcher tries them in some order. Often the ordering is determined in a rather complex and dynamic manner, with groups of related rules being more likely to be applied together, the most recently used rule not allowed to be reapplied immediately, and so on. In fact, many production systems's controllers have all the control structure tricks mentioned above (and more) built into them; the simple and elegant "bag of rules" ideal is inadequate for realistic examples. When the rules are explicitly written with an idiosyncratic control structure in mind, the system can become unprincipled and inexplicable.

On the same lines, notice how difficult it is to specify a time-ordered sequence of actions by a completely modular set of rewriting rules. It is unnatural to force knowledge about processes that may contain iteration, tests, and recursion into the form of independent situation-action rules. A view that is more easily defensible is that knowledge about procedures for perception should be encoded as (embedded in) computer procedures, not assertions or rules. The causal chain that dictates that some actions are best performed before others is implicit in the sequential execution of procedures, and the language constraints, such as iterate and test, test and branch, or subroutine invocation, are all fairly natural ways to think about solving certain problems. Production systems can in fact be made to perform all these procedural-like functions, but only through an abrogation of the ideal of modular, unordered, matching-oriented rule invocation which is the production system ideal. The question turns into one of aesthetics; how to use productions in a good style, and to work with their philosophy instead of against it.

To summarize the previous two objections: Production-based knowledge systems may in practice be no more robust, easily modified, modular, extensible, understandable, or self-understanding than any other (say, procedural) system unless great care is taken. After a certain level of complexity is reached, they are
likely to be as opaque as any other scheme because of the control-structuring methods that must be imposed on the pure production system form.

12.4 SCENE LABELING AND CONSTRAINT RELAXATION

The general computational problem of assigning labels consistently to objects is sometimes called the "labeling problem," and arises in many contexts, such as graph and automata homomorphism, graph coloring, Latin square generation, and of course, image understanding [Davis and Rosenfeld 1976; Zucker 1976; Haralick and Shapiro 1979]. "Relaxation labeling," "constraint satisfaction," and "cooperative algorithms" are natural implementations for labeling, and their potential parallelism has been a very influential development in computer vision. As should any important development, the relaxation paradigm has had an impact on the conceptualization as well as on the implementation of processes.

Cooperating algorithms to solve the labeling problem are useful in low level vision (e.g., line finding, stereopsis) and in intermediate-level vision (e.g., line-labeling, semantics-based region growing). They may also be useful for the highest-level vision programs, those that maintain a consistent set of beliefs about the world to guide the vision process.

Section 12.4.1 presents the main concepts in the labeling problem. Section 12.4.2 outlines some basic forms that "discrete labeling" algorithms can take. Section 12.4.3 introduces a continuing example, that of labeling lines in a line drawing, and gives a mathematically well-behaved probabilistic "linear operator" labeling method. Section 12.4.4 modifies the linear operator to be more in accord with our intuitions, and Section 12.4.5 describes relaxation as linear programming and optimization, thereby gaining additional mathematical rigor.

12.4.1 Consistent and Optimal Labelings

All labeling problems have the following notions.

1. A set of **objects**. In vision, the objects usually correspond to entities to be labeled, or assigned a "meaning."

2. A finite set of **relations** between objects. These are the sorts of relations we saw in Chapter 10; in vision, they are often geometric or topological relations between segments in a segmented image. Properties of objects are simply unary relations. An input scene is thus a relational structure.

3. A finite set of **labels**, or symbols associated with the "meanings" mentioned above. In the simplest case, each object is to be assigned a single label. A **labeling** assigns one or more labels to (a subset of) the objects in a relational structure. Labels may be weighted with "probabilities"; a (label, weight) pair can indicate something like the "probability of an object having that label."

4. **Constraints**, which determine what labels may be assigned to an object and what sets of labels may be assigned to objects in a relational structure.