```
(person (name ?first ?last))
```

expects two fields for the *name* slot, but the pattern

```
(person (name ?first ? last))
```

expects three fields for the *name* slot, and the last field must be the symbol *last*.

When a single-field slot is left unspecified in a pattern, CLIPS automatically adds a single-field wildcard check for that slot to the pattern. For example, the pattern

```
(person (name ?first ?last))
```

is converted to

```
(person (name ?first ?last)
        (social-security-number ?))
```

8.6 BLOCKS WORLD

To demonstrate variable bindings, we will build a program to move blocks in a simple blocks world. This type of program is analogous to the classic blocks world program in which the knowledge domain is restricted to blocks (Firebaugh 88). This is a good example of planning and might be applied to automated manufacturing, where a robot arm manipulates parts.

The second restriction will be that any goal must not already have been achieved. That is, the goal cannot be to move block x on top of block y if block x is already on top of block y. This is a rather simple condition to check; however, the appropriate syntax to test for this condition will not be introduced until Chapter 9.

To begin to solve this problem, it will be useful to set up a configuration of blocks that can be used for testing the program. Figure 8.1 shows the configuration that will be used. There are two stacks in this configuration. The first stack has block A on top of block B on top of block C. The second stack has block D on top of block E on top of block F.

To determine which type of rules would be effective in solving the problem, a step-by-step completion of a blocks world goal is useful. What steps must be taken to move block C on top of block E? The easiest solution for this problem would be to directly move block C on top of block E. However, this rule could only be applied if both block C and block E had no blocks on top of them. The pseudocode for this rule would be

```
RULE MOVE-DIRECTLY
IF   The goal is to move block ?upper on top of
       block ?lower and
     block ?upper is the top block in its stack and
     block ?lower is the top block in its stack,
THEN Move block ?upper on top of block ?lower.
```

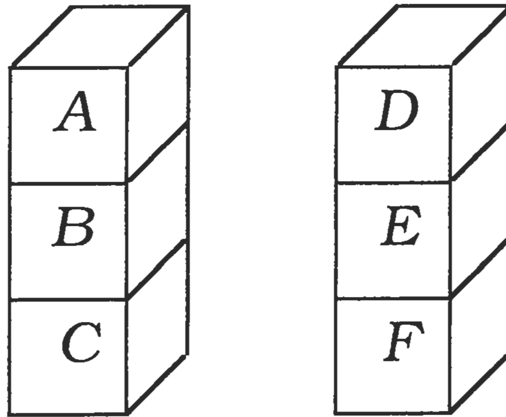

Figure 8.1 Blocks World Initial Configuration

The only things of interest in a blocks world are blocks. A single block may be stacked on another block. The goal of a complex blocks world program is to rearrange the stacks of blocks into a goal configuration with the minimum number of moves. For this example, a number of simplifying restrictions will be made. The first of these restrictions is that only one initial goal is allowed, and this goal can only be to move one block on top of another. With this restriction it is rather trivial to determine the optimal moves to achieve the goal. If the goal is to move block x on top of block y, then move all blocks (if any) on top of block x to the floor and all blocks (if any) on top of block y to the floor and then move block x on top of block y.

The *move-directly* rule cannot be used in this case since blocks A and B are on top of block C, and block D is on top of block E. In order to allow the *move-directly* rule to move block C on top of block E, blocks A, B, and D must be moved to the floor. Since this is the easiest step to take to get them out of the way, that is what happens. The simple blocks world does not require that the blocks be restacked and only a single initial goal is allowed, so there is no need to stack blocks when they are moved out of the way. This rule can be expressed as two pseudocode rules: one rule to clear blocks off the block to be moved and one rule to clear blocks off the block to be stacked on.

```
RULE CLEAR-UPPER-BLOCK
IF   The goal is to move block ?x and
     block ?x is not the top block in its stack and
     block ?above is on top of block ?x,
THEN The goal is to move block ?above to the floor

RULE CLEAR-LOWER-BLOCK
IF   The goal is to move another block on top of
       block ?x and
     block ?x is not the top block in its stack and
     block ?above is on top of block ?x,
THEN The goal is to move block ?above to the floor
```

The *clear-upper-block* rule will work to clear the blocks off block C. It will first determine that block B needs to be moved to the floor. In order to move block

B to the floor, this same rule will determine that block A needs to be moved to the floor. Similarly, the *clear-lower-block* rule will determine that block D needs to be moved to the floor in order to move something on top of block E.

Now there are subgoals to move blocks A, B, and D to the floor. Blocks A and D can be moved directly to the floor. If written properly, the *move-directly* rule might be able to handle moving blocks on top of the floor as well as on other blocks. Since the floor is really not a block, it may be necessary to treat the floor differently. The following pseudocode rule will handle the special case of moving a block to the floor:

```
RULE MOVE-TO-FLOOR
IF   The goal is to move block ?upper on top of the
       floor and
     block ?upper is the top block in its stack,
THEN Move block ?upper on top of the floor.
```

The *move-to-floor* rule can now move blocks A and D to the floor. Once block A is moved to the floor, the *move-to-floor* rule can be activated to move block B to the floor. With blocks A, B, and D on the floor, blocks C and E are now the top blocks in their stacks and it is possible to use the *move-directly* rule to move block C on top of block E.

Now that the rules have been written using pseudocode, the facts to be used by the rules should be determined. The types of facts needed cannot always be determined without some prototyping. In this case, the pseudocode rules point out several types of facts that will be needed. For example, the information about which blocks are on top of other blocks is crucial. This information could be described with the following deftemplate:

```
(deftemplate on-top-of
   (slot upper)
   (slot lower))
```

and the facts described by this template would be

```
(on-top-of (upper A) (lower B))
(on-top-of (upper B) (lower C))
(on-top-of (upper D) (lower E))
(on-top-of (upper E) (lower F))
```

Since it is also important to know which blocks are at the top and bottom of a stack, it would be useful to include the following facts:

```
(on-top-of (upper nothing) (lower A))
(on-top-of (upper C) (lower floor))
(on-top-of (upper nothing) (lower D))
(on-top-of (upper F) (lower floor))
```

The words *nothing* and *floor* have special meaning in these facts. The facts (on-top-of (upper nothing) (lower A)) and (on-top-of (upper nothing) (lower D)) indicate that A and D are the top blocks in their stacks. Similarly, the facts (on-top-of (upper C) (lower floor)) and (on-top-of (upper F) (lower floor)) indicate that blocks C and F are the bottom blocks in their stacks. Including these facts

does not necessarily solve the problem of determining the top and bottom blocks in a stack. If the rules are not written correctly, the words *floor* and *nothing* might be mistaken as the names of blocks. Facts that indicate the names of the blocks might be useful. The following facts using the implied deftemplate *build* can be used to identify the blocks from the special words *nothing* and *floor*:

```
(block A)
(block B)
(block C)
(block D)
(block E)
(block F)
```

Finally, a fact is needed to describe the block-moving goals that are being processed. These goals could be described with the deftemplate

```
(deftemplate goal (slot move) (slot on-top-of))
```

and the initial goal using this deftemplate would be

```
(goal (move C) (on-top-of E))
```

With the facts and deftemplates now defined, the initial configuration of the blocks world can be described by the following deffacts:

```
(deffacts initial-state
   (block A)
   (block B)
   (block C)
   (block D)
   (block E)
   (block F)
   (on-top-of (upper nothing) (lower A))
   (on-top-of (upper A) (lower B))
   (on-top-of (upper B) (lower C))
   (on-top-of (upper C) (lower floor))
   (on-top-of (upper nothing) (lower D))
   (on-top-of (upper D) (lower E))
   (on-top-of (upper E) (lower F))
   (on-top-of (upper F) (lower floor))
   (goal (move C) (on-top-of E)))
```

The *move-directly* rule is written as follows:

```
(defrule move-directly
   ?goal <- (goal (move ?block1)
                  (on-top-of ?block2))
   (block ?block1)
   (block ?block2)
   (on-top-of (upper nothing) (lower ?block1))
```

```
   ?stack-1 <- (on-top-of (upper ?block1)
                          (lower ?block3))
   ?stack-2 <- (on-top-of (upper nothing)
                          (lower ?block2))
   =>
   (retract ?goal ?stack-1 ?stack-2)
   (assert (on-top-of (upper ?block1)
                      (lower ?block2))
           (on-top-of (upper nothing)
                      (lower ?block3)))
   (printout t ?block1 " moved on top of " ?block2
               "." crlf))
```

The first three patterns determine that there is a goal to move a block on top of another block. Patterns two and three ensure that a goal to move a block onto the floor will not be processed by this rule. The fourth and sixth patterns check that the blocks are the top blocks in their stacks. The fifth and sixth patterns match against information necessary to update the stacks that the moving block is being taken from and moved to. The actions of the rule update the stack information for the two stacks and print a message. The block beneath the moved block is now the top block in that stack, and the block being moved is now the top block in the stack to which it was moved.

The *move-to-floor* rule is implemented as follows:

```
(defrule move-to-floor
   ?goal <- (goal (move ?block1) (on-top-of floor))
   (block ?block1)
   (on-top-of (upper nothing) (lower ?block1))
   ?stack <- (on-top-of (upper ?block1)
                        (lower ?block2))
   =>
   (retract ?goal ?stack)
   (assert (on-top-of (upper ?block1)
                      (lower floor))
           (on-top-of (upper nothing)
                      (lower ?block2)))
   (printout t ?block1 " moved on top of floor."
               crlf))
```

This rule is similar to the *move-directly* rule with the exception that it is not necessary to update some information about the floor since it is not a block.

The *clear-upper-block* rule is implemented as follows:

```
(defrule clear-upper-block
   (goal (move ?block1))
   (block ?block1)
   (on-top-of (upper ?block2) (lower ?block1))
   (block ?block2)
   =>
```

```
      (assert (goal (move ?block2)
                    (on-top-of floor))))
```

The *clear-lower-block* rule is implemented as follows:

```
(defrule clear-lower-block
   (goal (on-top-of ?block1))
   (block ?block1)
   (on-top-of (upper ?block2) (lower ?block1))
   (block ?block2)
   =>
   (assert (goal (move ?block2)
                 (on-top-of floor))))
```

The program is now complete with the *move-directly*, *move-to-floor*, *clear-upper-block*, and *clear-lower-block* rules, the *goal* and *on-top-of* deftemplates, and the *initial-state* deffacts. The following output shows a sample run of this blocks world program:

```
CLIPS> (unwatch all)↵
CLIPS> (reset)↵
CLIPS> (run)↵
A moved on top of floor.
B moved on top of floor.
D moved on top of floor.
C moved on top of E.
CLIPS>
```

Blocks A and B are first moved to the floor to clear block C. Block D is then moved to the floor to clear block E. Finally, block C can be moved on top of block E to solve the initial goal.

This example has demonstrated how to build a program using a step-by-step method. First, pseudorules were written using English-like text. Second, the pseudorules were used to determine the types of facts that would be required. Deftemplates describing the facts were designed, and the initial knowledge for the program was coded using these deftemplates. Finally, the pseudorules were translated to CLIPS rules using the deftemplates as a guide for translation.

The development of an expert system typically requires a great deal more prototyping and iterative development than in this example. It is not always possible to determine the best method for representing facts or the types of rules that will be needed to build an expert system. Following a consistent methodology, however, can aid in the development of an expert system even when a great deal of prototyping and iteration need to be performed.

8.7 MULTIFIELD WILDCARDS AND VARIABLES

Multifield Wildcards

Multifield wildcards and variables can be used to match against zero or more fields of a pattern. The **multifield wildcard** is indicated by a dollar sign followed