CHAPTER 2

Interactive Activation and Competition

Our own explorations of parallel distributed processing began with the use of interactive activation and competition mechanisms of the kind we will examine in this chapter. We have used these kinds of mechanisms to model visual word recognition (McClelland & Rumelhart, 1981; Rumelhart & McClelland, 1982) and to model the retrieval of general and specific information from stored knowledge of individual exemplars (McClelland, 1981), as described in *PDP:1*. In this chapter, we describe some of the basic mathematical observations behind these mechanisms, and then we introduce the reader to a specific model that implements the retrieval of general and specific information using the "Jets and Sharks" example discussed in *PDP:1* (pp. 25-31). (The interactive activation model of word perception is presented in Chapter 7.)

After describing the specific model, we will introduce the program in which this model is implemented: the **iac** program (for interactive activation and competition). The description of how to use this program will be quite extensive; it is intended to serve as a general introduction to the entire package of programs since the user interface and most of the commands and auxiliary files are common to all of the programs. After describing how to use the program, we will present several exercises, including an opportunity to work with the Jets and Sharks example and an opportunity to explore an interesting variant of the basic model, based on dynamical assumptions used by Grossberg (e.g., Grossberg, 1978).

BACKGROUND

The study of interactive activation and competition mechanisms has a long history. They have been extensively studied by Grossberg. A useful introduction to the mathematics of such systems is provided in Grossberg

(1978). Related mechanisms have been studied by a number of other investigators, including Levin (see Levin, 1976), whose work was instrumental in launching our exploration of PDP mechanisms.

An interactive activation and competition network (hereafter, *IAC network*) consists of a collection of processing units organized into some number of competitive pools. There are excitatory connections among units in different pools and inhibitory connections among units within the same pool. The excitatory connections between pools are generally bidirectional, thereby making the processing *interactive* in the sense that processing in each pool both influences and is influenced by processing in other pools. Within a pool, the inhibitory connections are usually assumed to run from each unit in the pool to every other unit in the pool. This implements a kind of competition among the units such that the unit or units in the pool that receive the strongest activation tend to drive down the activation of the other units.

The units in an IAC network take on continuous activation values between a maximum and minimum value, though their output—the signal that they transmit to other units—is not necessarily identical to their activation. In our work, we have tended to set the output of each unit to the activation of the unit minus the *threshold* as long as the difference is positive; when the activation falls below threshold, the output is set to 0. Without loss of generality, we can set the threshold to 0; we will follow this practice throughout the rest of this chapter. A number of other output functions are possible; Grossberg (1978) describes a number of other possibilities and considers their various merits.

The activations of the units in an IAC network evolve gradually over time. In the mathematical idealization of this class of models, we think of the activation process as completely continuous, though in the simulation modeling we approximate this ideal by breaking time up into a sequence of discrete steps.

Units in an IAC network change their activation based on a function that takes into account both the current activation of the unit and the net input to the unit from other units or from outside the network. The net input to a particular unit (say, unit i) is the same in almost all the models described in this volume: it is simply the sum of the influences of all of the other units in the network plus any external input from outside the network. The influence of some other unit (say, unit j) is just the product of that unit's output, $output_j$, times the strength or weight of the connection to unit i from unit j. Thus the net input to unit i is given by

$$net_i = \sum_j w_{ij} output_j + extinput_i. \tag{1}$$

In the IAC model, $output_j = [a_j]^+$. Here, a_j refers to the activation of unit j, and the expression $[a_j]^+$ has value a_j for all $a_j > 0$; otherwise its value is 0. The index j ranges over all of the units with connections to unit i. In

general the weights can be positive or negative, for excitatory or inhibitory connections, respectively.

Once the net input to a unit has been computed, the resulting *change* in the activation of the unit is as follows:

If $(net_i > 0)$,
$$\Delta a_i = (max - a_i)net_i - decay\,(a_i - rest).$$

Otherwise,
$$\Delta a_i = (a_i - min)net_i - decay\,(a_i - rest).$$

Note that in this equation, *max*, *min*, *rest*, and *decay* are all parameters. In general, we choose $max = 1$, $min \leqslant rest \leqslant 0$, and *decay* between 0 and 1. Note also that a_i is assumed to start, and to stay, within the interval [*min*, *max*].

Suppose we imagine the input to a unit remains fixed and examine what will happen across time in the equation for Δa_i. For specificity, let's just suppose the net input has some fixed, positive value. Then we can see that Δa_i will get smaller and smaller as the activation of the unit gets greater and greater. For some values of the unit's activation, Δa_i will actually be negative. In particular, suppose that the unit's activation is equal to the resting level. Then Δa_i is simply $(max-rest)net_i$. Now suppose that the unit's activation is equal to *max*, its maximum activation level. Then Δa_i is simply $(-decay)(max-rest)$. Between these extremes there is an equilibrium value of a_i, at which Δa_i is 0. We can find what the equilibrium value is by setting Δa_i to 0 and solving for a_i:

$$\begin{aligned} 0 &= (max - a_i)net_i - decay\,(a_i - rest) \\ &= (max)(net_i) + (rest)(decay) - a_i(net_i + decay) \end{aligned}$$

$$a_i = \frac{(max)(net_i) + (rest)(decay)}{net_i + decay}. \tag{2}$$

Using $max = 1$ and $rest = 0$, this simplifies to

$$a_i = \frac{net_i}{net_i + decay}. \tag{3}$$

What the equation indicates, then, is that the activation of the unit will reach equilibrium when its value becomes equal to the ratio of the net input divided by the net input plus the decay. Note that in a system where the activations of other units—and thus of the net input to any particular unit—are also continually changing, there is no guarantee that activations will ever completely stabilize—although in practice, as we shall see, they often seem to.

Equation 3 indicates that the equilibrium activation of a unit will always increase as the net input increases; however, it can never exceed 1 (or, in

the general case, *max*) as the net input grows very large. Thus, *max* is indeed the upper bound on the activation of the unit. For small values of the net input, the equation is approximately linear since $x/(x+c)$ is approximately equal to x/c for x small enough.

We can see the decay term in Equation 3 as acting as a kind of restoring force that tends to bring the activation of the unit back to 0 (or to *rest*, in the general case). The larger the value of the decay term, the stronger this force is, and therefore the lower the activation level will be at which the activation of the unit will reach equilibrium. Indeed, we can see the decay term as scaling the net input if we rewrite the equation as

$$a_i = \frac{net_i/decay}{(net_i/decay)+1}. \tag{4}$$

When the net input is equal to the decay, the activation of the unit is 0.5 (in the general case, the value is $(max+rest)/2$). Because of this, we generally scale the net inputs to the units by a strength constant that is equal to the decay. Increasing the value of this strength parameter or decreasing the value of the decay increases the equilibrium activation of the unit.

In the case where the net input is negative, we get entirely analogous results:

$$a_i = \frac{(min)(net_i) - (decay)(rest)}{net_i - decay}. \tag{5}$$

Using $rest = 0$, this simplifies to

$$a_i = \frac{(min)(net_i)}{net_i - decay} \tag{6}$$

This equation is a bit confusing because net_i and *min* are both negative quantities. It becomes somewhat clearer if we use *amin* (the absolute value of *min*) and $anet_i$ (the absolute value of net_i). Then we have

$$a_i = -\frac{(amin)(anet_i)}{anet_i + decay}. \tag{7}$$

What this last equation brings out is that the equilibrium activation value obtained for a negative net input is scaled by the magnitude of the minimum (*amin*). Inhibition both acts more quickly and drives activation to a lower final level when *min* is farther below 0.

How Competition Works

So far we have been considering situations in which the net input to a unit is fixed and activation evolves to a fixed or stable point. The

interactive activation and competition process, however, is more complicated than this because the net input to a unit changes as the unit and other units in the same pool simultaneously respond to their net inputs. One effect of this is to amplify differences in the net inputs of units. Consider two units a and b that are in competition, and imagine that both are receiving some excitatory input from outside but that the excitatory input to a (e_a) is stronger than the excitatory input to b (e_b). Let γ represent the strength of the inhibition each unit exerts on the other. Then the net input to a is

$$net_a = e_a - \gamma(output_b) \tag{8}$$

and the net input to b is

$$net_b = e_b - \gamma(output_a). \tag{9}$$

As long as the activations stay positive, $output_i = a_i$, so we get

$$net_a = e_a - \gamma a_b \tag{10}$$

and

$$net_b = e_b - \gamma a_a. \tag{11}$$

From these equations we can easily see that b will tend to be at a disadvantage since the stronger excitation to a will tend to give a a larger initial activation, thereby allowing it to inhibit b more than b inhibits a. The end result is a phenomenon that Grossberg (1976) has called "the rich get richer" effect: Units with slight initial advantages, in terms of their external inputs, amplify this advantage over their competitors.

Resonance

Another effect of the interactive activation process has been called "resonance" by Grossberg (1978). If unit a and unit b have mutually excitatory connections, then once one of the units becomes active, they will tend to keep each other active. Activations of units that enter into such mutually excitatory interactions are therefore sustained by the network, or "resonate" within it, just as certain frequencies resonate in a sound chamber. In a network model, depending on parameters, the resonance can sometimes be strong enough to overcome the effects of decay. For example, suppose that two units, a and b, have bidirectional, excitatory connections with strengths of $2 \times decay$. Suppose that we set each unit's activation at 0.5 and then

remove all external input and see what happens. The activations will stay at 0.5 indefinitely because

$$\begin{aligned}\Delta a_a &= (1 - a_a)net_a - (decay)a_a \\ &= (1 - 0.5)(2)(decay)(0.5) - (decay)(0.5) \\ &= (0.5)(2)(decay)(0.5) - (decay)(0.5) \\ &= 0.\end{aligned}$$

Thus, IAC networks can use the mutually excitatory connections between units in different pools to sustain certain input patterns that would otherwise decay away rapidly in the absence of continuing input. The interactive activation process can also activate units that were not activated directly by external input. We will explore these effects more fully in the exercises that are given later.

Hysteresis and Blocking

Before we finish this consideration of the mathematical background of interactive activation and competition systems, it is worth pointing out that the rate of evolution towards the eventual equilibrium reached by an IAC network, and even the state that is reached, is affected by initial conditions. Thus if at time 0 we force a particular unit to be on, this can have the effect of slowing the activation of other units. In extreme cases, forcing a unit to be on can totally block others from becoming activated at all. For example, suppose we have two units, a and b, that are mutually inhibitory, with inhibition parameter γ equal to 2 times the strength of the decay, and suppose we set the activation of one of these units—unit a—to 0.5. Then the net input to the other—unit b—at this point will be $(-0.5)(2)(decay) = -decay$. If we then supply external excitatory input to the two units with strength equal to the decay, this will maintain the activation of unit a at 0.5 and will fail to excite b since its net input will be 0. The external input to b is thereby blocked from having its normal effect. If external input is withdrawn from a, its activation will gradually decay (in the absence of any strong resonances involving a) so that b will gradually become activated. The first effect, in which the activation of b is completely blocked, is an extreme form of a kind of network behavior known as *hysteresis* (which means "delay"); prior states of networks tend to put them into states that can delay or even block the effects of new inputs.

Because of hysteresis effects in networks, various investigators have suggested that new inputs may need to begin by generating a "clear signal," often implemented as a wave of inhibition. Such ideas have been proposed by various investigators as an explanation of visual masking effects (see,

e.g., Weisstein, Ozog, & Szoc, 1975) and play a prominent role in Grossberg's theory of learning in neural networks (see Grossberg, 1980).

Grossberg's Analysis of Interactive Activation and Competition Processes

Throughout this section we have been referring to Grossberg's studies of what we are calling interactive activation and competition mechanisms. In fact, he uses a slightly different activation equation than the one we have presented here (taken from our earlier work with the interactive activation model of word recognition). In Grossberg's formulation, the excitatory and inhibitory inputs to a unit are treated separately. The excitatory input (e) drives the activation of the unit up toward the maximum, whereas the inhibitory input (i) drives the activation back down toward the minimum. As in our formulation, the decay tends to restore the activation of the unit to its resting level.

$$\Delta a = (max - a)e - (a - min)i - decay(a - rest). \tag{12}$$

Grossberg's formulation has the advantage of allowing a single equation to govern the evolution of processing instead of requiring an *if* statement to intervene to determine which of two equations holds. It also has the characteristic that the direction the input tends to drive the activation of the unit is affected by the current activation. In our formulation, net positive input tends always to excite the unit and net negative input tends always to inhibit it. In Grossberg's formulation, the input is not lumped together in this way. As a result, the effect of a given input (particular values of e and i) can be excitatory when the unit's activation is low and inhibitory when the unit's activation is high. Furthermore, at least when *min* has a relatively small absolute value compared to *max*, a given amount of inhibition will tend to exert a weaker effect on a unit starting at rest. To see this, we will simplify and set $max = 1.0$ and $rest = 0.0$. By assumption, the unit is at rest so the above equation reduces to

$$\Delta a = (1)(e) - (amin)(i) \tag{13}$$

where *amin* is the absolute value of *min* as above. This is in balance only if $i = e/amin$.

Our use of the net input rule was based primarily on the fact that we found it easier to follow the course of simulation events when the balance of excitatory and inhibitory influences was independent of the activation of the receiving unit. However, this by no means indicates that our formulation is superior computationally. Therefore we have made Grossberg's update rule available as an option in the **iac** program.

THE IAC MODEL

The IAC model provides a discrete approximation to the continuous interactive activation and competition processes that we have been considering up to now. We will consider two variants of the model: one that follows the interactive activation dynamics from our earlier work and one that follows the formulation offered by Grossberg.

Architecture

The IAC model consists of several units, divided into *pools*. In each pool, all the units are mutually inhibitory. Between pools, units may have excitatory connections. The model assumes that these connections are bidirectional, so that whenever there is an excitatory connection from unit *i* to unit *j*, there is also an excitatory connection from unit *j* back to unit *i*.

Visible and Hidden Units

In an IAC network, there are generally two classes of units: those that can receive direct input from outside the network and those that cannot. The first kind of units are called *visible* units; the latter are called *hidden* units. Thus in the IAC model the user may specify a pattern of inputs to the visible units, but by assumption the user is not allowed to specify external input to the hidden units; their net input is based only on the outputs from other units to which they are connected.

Activation Dynamics

Time is not continuous in the IAC model (or any of our other simulation models), but is divided into a sequence of discrete steps, or *cycles*. Each cycle begins with all units having an activation value that was determined at the end of the preceding cycle. First, the inputs to each unit are computed. Then the activations of the units are updated. The two-phase procedure ensures that the updating of the activations of the units is effectively synchronous; that is, nothing is done with the new activation of any of the units until all have been updated.

The discrete time approximation can introduce instabilities if activation steps on each cycle are large. This problem is eliminated, and the approximation to the continuous case is generally closer, when activation steps are kept small on each cycle.

Parameters

In the IAC model there are several parameters under the user's control. Most of these have already been introduced. They are

max
: The maximum activation parameter.

min
: The minimum activation parameter.

rest
: The resting activation level to which activations tend to settle in the absence of external input.

decay
: The decay rate parameter, which determines the strength of the tendency to return to resting level.

estr
: This parameter stands for the strength of external input (i.e., input to units from outside the network). It scales the influence of external signals relative to internally generated inputs to units.

alpha
: This parameter scales the strength of the excitatory input to units from other units in the network.

gamma
: This parameter scales the strength of the inhibitory input to units from other units in the network.

In general, it would be possible to specify separate values for each of these parameters for each unit. The IAC model does not allow this, as we have found it tends to introduce far too many degrees of freedom into the modeling process. However, the model does allow the user to specify strengths for the individual connection strengths in the network.

IMPLEMENTATION

The IAC model is implemented by the **iac** program. This program, like all of our simulation programs, is written in C. The program consists of several parts: the command interpreter, the display package, the network configuration package, the patterns package, and the core routines of the model. In describing the implementation of this and other models, we will focus our attention on the core routines, but here we will briefly describe the rest of the package so that the reader has some pointers to understanding what is going on. More detailed implementation information is provided in Appendix F, which serves as a guide for readers who wish to

actually explore and possibly alter the source code itself. Here follow brief descriptions of the various noncore parts of the program.

The Command Interpreter

The command interpreter is a set of subroutines that reads commands, either from a start-up file when the program is first called or from the keyboard while the program is running. There is also a facility that allows the user to direct the command interpreter to read and execute a sequence of commands found in a file. The command interpreter works by looking up commands it encounters in a large table of commands and executing the subroutine that is found in the table associated with the command. We will explain how to use the command interpreter in the section "Running the Program" later in this chapter.

The Display Package

The display package is a set of routines that manages the 24×80 character display screen. One set of routines is used to read a file called the *template* file when the program is first called. The information in this file is used to set up a set of *templates*, or display objects, and to indicate what each template contains and where on the screen it should be displayed. Another set of routines is used to do the actual displaying; these are commands that can be issued either by the user directly or from other parts of the program.

The Network Configuration Package

This package consists of a set of routines that is used in configuring the program for a particular application. The routines read commands from a file called the *network configuration* file, or the *network* file for short, and use these commands to set up arrays for the units and the weights, to specify initial values for the connections, and to indicate whether connections are modifiable or not.

The Patterns Package

This package consists of a set of routines that is used to read in a set of patterns for use as inputs to the model. These routines read a file called

the *pattern* file. Some of the programs—among them, the **iac** program—can be run without reading in a file full of patterns, but the package is available for use if desired.

The Core Routines

Beyond the routines just mentioned is a set of *core* routines that implements the activation and competition processes described earlier. The routines are simple and make up a rather small part of the program. Here we explain the basic structure of the core routines used in the **iac** program.

getinput. This routine is used to specify which of the units in the network will receive external input. The routine prompts the user for names or numbers of units and for corresponding external input values, after first allowing the external inputs to be cleared to all zeros if desired. Note that this does not actually start the process of sending inputs to the units; it simply says which units should receive inputs and how strong they should be when the process actually starts.

reset. This routine is used to reset the activations of units to their resting levels and to reset the time—the current cycle number—back to 0. All other relevant variables are cleared, and the display is updated to show the initial state of the network before processing begins.

cycle. This routine is the basic routine that is used in running the model. It carries out a number of processing cycles, as determined by the program control variable *ncycles*. On each cycle, two routines are called: *getnet* and *update*. At the end of each cycle, the program checks to see whether the display is to be updated and whether to pause so the user can examine the new state (and possibly terminate processing). At the end of *ncycles* of processing, the display is updated if it has not been updated on every cycle. The routine looks like this:

```
cycle() {

  for (cy = 0; cy < ncycles; cy++) {
    cycleno++;
    getnet();
    update();

/* what follows is concerned with
   pausing and updating the display */
    if (step_size == CYCLE) {
      update_display();
```

```
      if (single_step) {
        if (contin_test() == BREAK) break;
      }
    }
  }
  if (step_size > CYCLE) {
    update_display();
  }
}
```

The *getnet* and *update* routines are somewhat different for the standard version and Grossberg version of the program. We first describe the standard versions of each, then turn to the Grossberg versions.

Standard getnet. The standard *getnet* routine computes the net input to each unit. The net input consists of three things: the external input, scaled by *estr*; the excitatory input from other units, scaled by *alpha*; and the inhibitory input from other units, scaled by *gamma*. For each unit, the *getnet* routine first accumulates the excitatory and inhibitory inputs from other units, then scales the inputs and adds them to the scaled external input to obtain the net input.

Whether a connection is excitatory or inhibitory is determined by its sign. Thus if w_{ij} is positive, $w_{ij}a_j$ is added into the excitation term of unit i. If w_{ij} is negative, $w_{ij}a_j$ is added into the inhibition term of unit i. These operations are only performed if the activation of the sending unit is greater than 0. The code that implements these calculations is as follows:

```
getnet() {

  for (i = 0; i < nunits; i++) {
    excitation[i] = inhibition[i] = 0;

    for (j = 0; j < nunits; j++) {
      if (activation[j] > 0) {
        if (w[i][j] > 0) {
          excitation[i] += weight[i][j]*activation[j];
        }
        else if (w[i][j] < 0) {
          inhibition[i] += weight[i][j]*activation[j];
        }
      }
    }
    netinput[i] = estr*extinput[i] + alpha*excitation[i]
                  + gamma*inhibition[i];
  }
}
```

Standard update. The *update* routine increments the activation of each unit, based on the net input and the existing activation value. Here is what it looks like:

```
update() {

  for (i = 0; i < nunits; i++) {
    if (netinput[i] > 0) {
      activation[i] += (max - activation[i])*netinput[i]
                       - decay*(activation[i] - rest);
    }
    else {
      activation[i] += (activation[i]-min)*netinput[i]
                       - decay*(activation[i] - rest);
    }
    if (activation[i] > max) activation[i] = max;
    if (activation[i] < min) activation[i] = min;
  }
}
```

The last two conditional statements are included to guard against the anomalous behavior that would result if the user had set the *estr*, *istr*, and *decay* parameters to values that allow activations to change so rapidly that the approximation to continuity is seriously violated and activations have a chance to escape the bounds set by the values of *max* and *min*.

Grossberg versions. The Grossberg versions of these two routines are structured like the standard versions. In the *getnet* routine, the only difference is that the net input to each unit is not computed; instead, the excitation and inhibition are scaled by *alpha* and *gamma*, respectively, and scaled external input is added to the excitation if it is positive or is added to the inhibition if it is negative:

```
excitation[i]*= alpha*excitation[i];
inhibition[i]*= gamma*inhibition[i];
if (extinput[i] > 0) excitation[i] += estr*extinput[i];
else if (extinput[i] < 0)
  inhibition[i] += estr*extinput[i];
```

In the *update* routine the two different versions of the standard activation rule are replaced by a single expression. The routine then becomes

```
update() {

  for (i = 0; i < nunits; i++) {
    activation[i] += (max - activation[i])*excitation[i]
                   + (activation[i] - min)*inhibition[i]
                     - decay*(activation[i] - rest);
```

```
    if (activation[i] > max) activation[i] = max;
    if (activation[i] < min) activation[i] = min;
  }
}
```

The reader may have noticed that the main computational loops of the program make no explicit mention of the IAC network architecture, in which the units are organized into competitive (inhibitory) pools and in which excitatory connections are assumed to be bidirectional. These architectural constraints are imposed in the *network* file. In fact, the **iac** program can implement any of a large variety of network architectures, including many that violate the architectural assumptions of the IAC framework.

As these examples illustrate, the core routines of this model—indeed, of all of our models—are extremely simple. Actually, some complexity has been suppressed, but not much. What makes the programs rather complex is all of the auxiliary routines.

RUNNING THE PROGRAM

Starting Up

To run the **iac** program, it is first necessary to set up a working directory containing the relevant files. An explanation of how this is done is given in Appendix A. Here we assume that you have created a working directory for **iac** and that you have positioned yourself in that directory. To execute the program, you would enter the following:

iac <*templatefile*> <*startupfile*>

Note that any commands entered either inside or outside of our program must be terminated by pressing the *return* or *enter* key. We adopt the convention of giving variables that must be replaced by specific values inside of angle brackets. Thus <*templatefile*> must be replaced by the name of a specific template file, and <*startupfile*> must be replaced by the name of a specific start-up file. By convention, the names of template files end with the extension *.tem* and the names of start-up files end with the extension *.str*. Henceforth we will refer to the template file as the *.tem* file, and the start-up file as the *.str* file.

The program will run without a template file or a start-up file being given, but the template file is necessary to tell the program what to display and where to display it; without one, there will be no display on the screen. The first argument to the program is always interpreted as a template file

name, so the program will misinterpret the *.str* file if the *.tem* file is left out. To prevent this, the program may be run with a single "–" in place of the template file name:

iac – <startupfile>

The *.str* file can be omitted without any ill effects. In general this file contains commands that initialize the network configuration and set the values of various parameters of the model. These can all be entered directly by the user once the program has started to run. The two things that are special about the *.str* file is that the commands in it are executed without printing anything to the screen and that errors encountered in the *.str* file cause the program to terminate immediately, with an error message printed to the screen. The *.str* file can contain any commands the user wishes to put in it, including commands to run the program, save output, and quit. This allows programs to be run in background mode on UNIX systems, using a script of commands from the *.str* file.

Assuming the *.str* file is processed without error and without encountering a *quit* command, the program will present a display containing an **iac**: prompt on line 0, a menu listing commands that may be entered on lines 1 through 4, and a display of the current state of the network. From this point on, the user may enter commands to the program via the command interface.

Entering Commands via the Command Interface

It is useful to think of commands as being entered one per line, with spaces separating the command from its various arguments. For example, the **iac** program provides a command that allows the user to display any of the various display chunks or *templates* that have been specified in the *.tem* file. This command is given by entering

iac: *disp <template>*

where *<template>* is the name of any template specified in the *.tem* file. Note that in this and subsequent examples, we display the prompt typed by the computer in **bold**, with the response from the user in *italic*. Also note that the user interface is case sensitive, and command names are in lowercase throughout. In the exercises we capitalize the first letter of some of the unit names; otherwise everything is lowercase.

A nice feature of the command interface is that it will generally prompt you with possible options should you wish to see them. At the top level, the program always provides a list of the commands that can be entered.

To see lists of options that are specifiable within a given command, enter the command name by itself. Thus if you enter

iac: *disp*

the program comes back with the list of possible continuations of the display command and a revised prompt,

iac: disp /

indicating that you may now enter the continuation you want. If you just want to look at the list of possible continuations, you can type *return* (press the return or enter key), and the program will return to the top level. Alternatively, you may enter one of the available continuations. If further input is required, you will be prompted for it; you may type *return* at almost any time, and the program will revert to the top-level prompt, awaiting a new command input.

One may think of the commands available in the program as consisting of a command name, followed by one or more *specifiers*, followed finally by one or more *arguments*. The specifiers indicate which particular one of several specific commands you wish to execute, and the arguments are the parameters of the command itself. Commands or specifiers that must be followed by further specifiers are terminated in the menus by a "/" character. The other commands or specifiers do not require further specifiers, though the user will generally be prompted for additional arguments. The *display* command is an example of the former type of command: It requires a further specifier indicating which template to display. The *log* command is an example of the latter type. It is used to control the storing of a log of the activity of the network in a file. This command requires a file name argument.

In all cases, the entire command, including the command name, the specifiers, and the arguments, may be entered as a single line. Alternatively, the user may enter any part of a command and be prompted for the possible continuations of it.

A further feature of the command interface is that commands and command specifiers do not have to be entered in their entirety; instead, it is only necessary to type enough of the beginning of the command or specifier to uniquely distinguish it from all other available alternatives. Thus,

iac: *lo foo.log*

is sufficient to open a log file called *foo.log*, given that there are no other commands accessible at the top level that begin with the characters *lo*.

The command interpreter prints an error message if the command string entered is not consistent with any of the available commands. The message is available for a short period (a few seconds), then the command