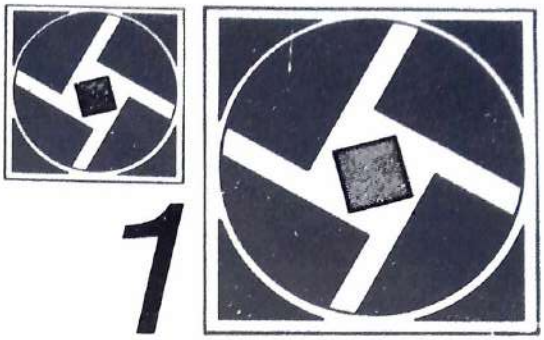
TABLE OF CONTENTS



1

To many computer adventurers, Prolog is the magical door of programming languages. Unlike almost all other computer languages, Prolog supports formal symbolic reasoning, making theoretically possible "intelligent computers" that can understand human languages and diagnose medical illnesses as well as androids that can perform routine tasks without the need for a procedure defined by a human.

As with other languages, however, a computer using Prolog is no more intelligent than its creator. The knowledge stored in the computer must come initially from some human programmer. Prolog gains its advantage by solving some types of problems far more efficiently than other languages. A computer using Prolog is "intelligent" in the sense that it approaches these problems in the same way that a human does.

Before using Turbo Prolog, let us take a quick look at the language and its possibilities.

WHAT IS PROLOG?

The name *Prolog* was taken from the phrase "*pro*gramming in *log*ic." The language was originally developed in 1972 by Alain Colmerauer and P. Roussel at the University of Marseilles in France.

Prolog is unique in its ability to *infer* (derive by formal reasoning) facts and conclusions from other facts. To enlist Prolog's help in solving a problem, the user attempts to describe the problem in a logically coherent and formal manner, presenting facts and knowledge about the problem and specifying a goal. The computer uses this knowledge to achieve the specified goal, defining its own procedure. In Chapters 3 and 4 you will explore some specific examples of the use of formal reasoning in Turbo Prolog programming.

A SHORT HISTORY OF PROLOG

During the 1970s Prolog became popular in Europe for artificial intelligence applications. In the United States, however, Prolog remained a relatively minor computer language. In the United States researchers preferred the LISP language for artificial intelligence applications. LISP was considered a more powerful language for these applications, though it was more difficult to learn and use than Prolog.

During these early years both Prolog and LISP were very slow in execution and consumed large amounts of memory. In addition, users needed considerable programming expertise to use a Prolog or LISP program. Doctors, lawyers, engineers, and others with the most critical needs for expert systems written in these languages found that the languages' complexity limited the use of expert systems to environments such as universities, the federal government, and corporate research departments.

Eventually the microcomputer became popular, and before long versions of Prolog began to appear for the microcomputer. These implementations almost invariably were slow and were of very limited productive use due to Prolog's needs for extensive memory and a fast processor. Compilers eventually emerged to help mitigate the speed problem, but Prolog was still viewed as a limited language, and few developers were willing to spend the money to develop more powerful versions of Prolog.

The picture suddenly changed in 1981 at the First International Conference on Fifth Generation Systems in Tokyo. The Japanese were having great difficulty competing with the rapidly growing U.S. computer market. Forming a new and aggressive national technology plan, the Japanese announced they would leapfrog over the current technology with new hardware and software by the 1990s. The computer language for the new systems would be Prolog.

As a result of this Japanese initiative, developers around the world took a new interest in Prolog. Today's versions of Prolog have features and power that far exceed those of a few years ago. They often are easier to use, cost less

than earlier versions, and are faster. On a mainframe computer or microcomputer, you will now find that Prolog often is the best language for any application that involves formal or symbolic reasoning.

HOW PROLOG IS DIFFERENT

Almost all languages developed for the computer during the last few decades are known generically as *procedural languages.* FORTRAN, COBOL, C, BASIC, dBASE III, and Pascal are all examples of procedural languages. To use a procedural language, an *algorithm,* or procedure, must first be defined to solve the problem at hand. A program is then written using the procedural language to implement the procedure. The program can execute the same procedure hundreds or thousands of times with different input data. The program will be far faster and more reliable than a human being in solving any problem requiring the use of that procedure, but the program will be limited to problems involving that single procedure.

For example, suppose a bank is frequently approached by people who wish to borrow money to expand their businesses. The bank looks at the history of each company, makes many calculations using this historical data, and then decides whether to loan money to the company. The calculation portion of the decision process is an objective procedure and is always the same. A programmer could work with the bank to define this procedure and then write a program to perform the calculation process and make it more efficient and reliable.

Procedural languages distinguish between a program and the data it uses. The procedure and control structures a program uses are defined by a programmer. The program manipulates the data according to the defined procedure.

Developing programs with Prolog is dramatically different. In fact, if you have programmed in conventional languages and try to learn Prolog, you will find you must go through an unlearning process before you can become proficient in Prolog. Turbo Prolog, like other implementations of Prolog, is an *object-oriented language.* It uses no procedures and essentially no program. (However, it is convenient to refer to the systems written in Prolog as *programs,* and we shall do so in this book.) Prolog uses only data about objects and their relationships. Prolog also emphasizes *symbolic processing.* LISP and Prolog are the best-known object-oriented languages.

With Prolog, the user defines a *goal* (a problem or objective), and the computer must find both the procedure and the solution. A Prolog program is a collection of data or facts and the relationships among these facts. In other words, the program is a *database.*

Prolog's database characteristics make the language ideal for a problem that is unstructured and for which the procedures to solve it are unknown. What is known about the problem is stored as data. The user defines a goal, or *hypothesis,* and the program, using formal reasoning, attempts to prove or disprove the goal based on the known data.

Because Prolog uses no real program, Prolog uses no programmers. *Knowledge engineers* design and build the systems that use Prolog. The knowledge engineer listens to experts in a particular domain, abstracting the subjective methods used by the human mind and defining an objective system of formal reasoning that can be supported by a Prolog structure.

One of the exciting aspects of Prolog is that it permits you to do things with your computer that could only be done inefficiently or not at all with other languages. However, you will find Prolog very inefficient if you try to use it for numerical or string processing involving known procedures.

The features of procedural and object-oriented languages are summarized in Table 1.1.

THE POWER OF PROLOG

Prolog's power is in its ability to infer facts from other facts. This makes it distinctly different from a language that is primarily a numerical processor. A

Procedural languages (BASIC, COBOL, Pascal)	**Object-oriented languages** (Prolog, LISP)
Use previously defined procedures to solve problems	Use heuristics to solve problems
Most efficient at numerical processing	Most efficient at formal reasoning
Systems created and maintained by programmers	Systems developed and maintained by knowledge engineers
Use structured programming	Interactive and cyclic development

Table 1.1: Object-oriented and procedural languages

numerical processor, for example, can calculate the monthly payment for a car loan. This value for the monthly payment is something new; it was not entered as data by the user. The payment value was calculated using a previously defined procedure.

On the other hand, suppose you wanted an intelligent spelling corrector that could guess the word you were trying to spell. This same numerical processor would fail dramatically at this task. Words have a symbolic meaning based on their context; an intelligent spelling corrector has to look at context and make decisions based on the symbolic relationship of words. If the word *for* is used instead of *form,* for example, only a contextual analysis using symbolic processing could distinguish the misspelling; most conventional spelling checkers would accept the word as correct. Prolog makes such symbolic processing possible. *Form* is always a verb, a noun, or an adjective. *For,* in contrast, is a preposition. A Prolog spelling checker could catch the error by examining how the word is used in the sentence.

Procedural languages use *algorithms.* Algorithms are objective procedures that, when followed, guarantee a solution. Object-oriented languages, such as Prolog, use *heuristics.* Heuristics are rules of thumb that are useful in reaching a goal. They do not guarantee a solution, nor do they always point to the most efficient solution. They are useful, however, when no algorithm exists.

In making a chess move, for example, you do not analyze all possible alternatives or use a defined procedure to try to determine your next move. You use certain general rules based on your experience and knowledge of the game. Your decision may be correct, but there may have been a better alternative. You used a heuristic to determine your move.

You can view the process of solving any problem as an attempt to move through a problem space to a specific objective. The human mind, when moving through such a problem space, breaks a problem into smaller units, and the solutions to these units form a series of intermediate goals. Each intermediate goal is a step toward the final solution.

For example, suppose a tape on your stereo plays through only one speaker. To try to solve this problem, you could first switch the stereo from tape to FM input. If the stereo sound now comes from both channels, you know that the tape input is bad. Next, you might try another tape to see if the tape is at fault. Finally, you might try cleaning the tape head. You follow a sequence of steps to try to focus on the cause of the problem. Each step is a heuristic (Figure 1.1).

In the same way, you construct Prolog systems using heuristics instead of algorithms. The problem space consists of nodes and links. Each node is a subgoal, or step, to the final solution. The links are the operations that move you between nodes. In a procedural program, you must define the specific

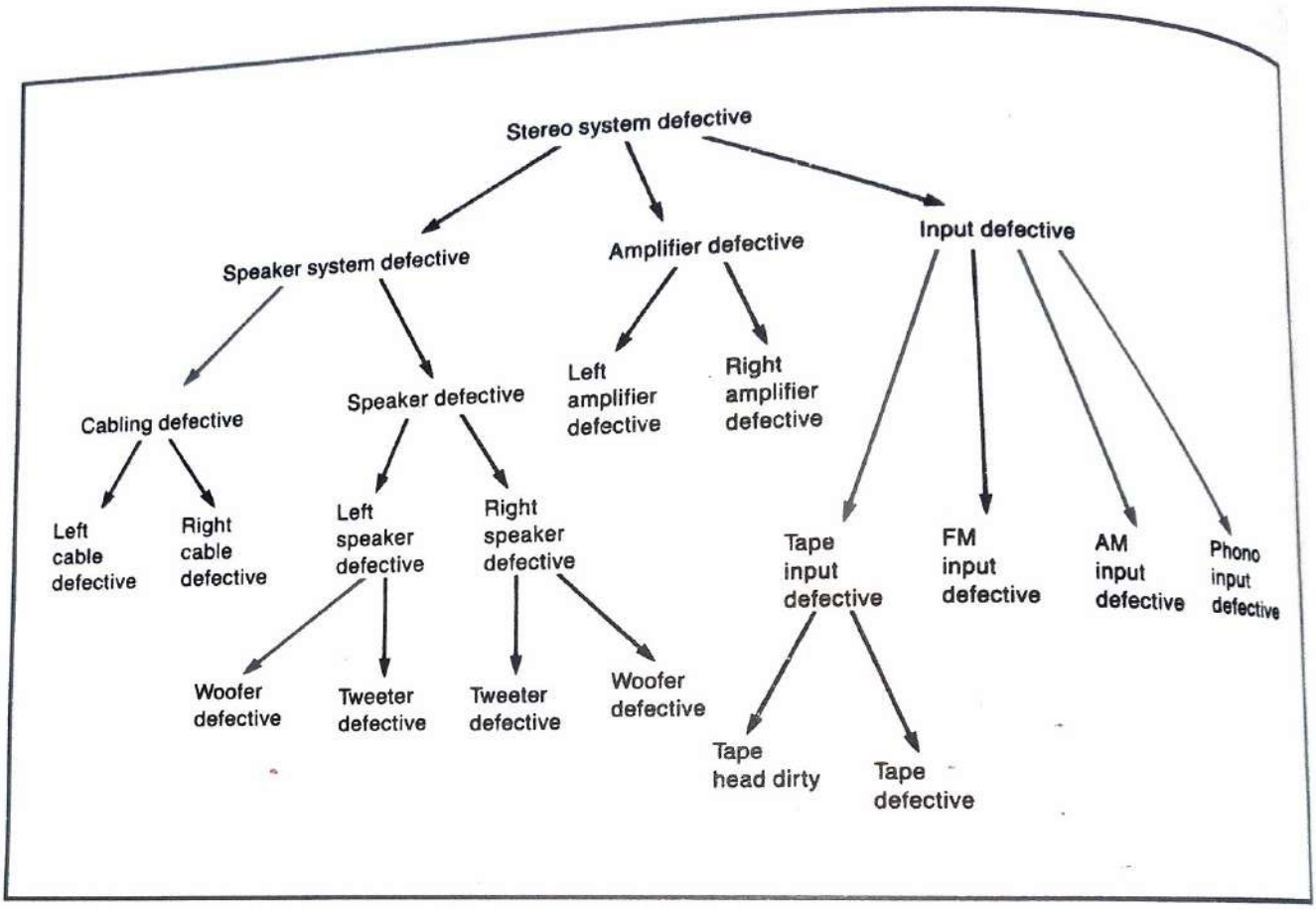


Figure 1.1: Working through a problem space

path through the problem space. With Prolog, you only define the problem space. In defining this space, however, you impose certain heuristics on the real-world problem it represents. The resulting Prolog system should converse with the user and reason in the same way a human reasons in solving the problem. For example, here is a possible dialog between a user and Prolog (the user's responses are italicized):

Does sound come from both speakers in the tape mode?
No.

Does sound come from both speakers in the FM mode?
Yes.

Try another tape, using tape input. Does sound come from both speakers now?
No.

Clean the tape mechanism. Does sound come from both speakers now?
Yes.

The tape head was dirty.

Developing Prolog systems, then, requires a different way of thinking about problems and their solutions than when procedural languages are used. Prolog approaches problems more like humans do, but you still cannot use the full arsenal of human approaches. The human mind can use formal reasoning, numerical processing, analogy, intuition, and subjective reasoning in solving problems. With Prolog, you are limited primarily to objective formal reasoning.

APPLICATIONS FOR PROLOG

Prolog is useful for almost any application that requires formal reasoning. This includes applications in expert systems, natural language processing, robotics, and gaming and simulations.

Expert Systems

Expert systems are programs that use inference techniques that involve formal reasoning normally performed by a human expert to solve problems in a specific area of knowledge. Expert systems can advise, diagnose, analyze, and categorize using a previously defined knowledge base. The knowledge base is a collection of rules and facts, often written in Prolog.

To use an expert system, the user begins by specifying a goal. The system then attempts to solve this goal by asking a series of questions and using its internal knowledge. In some cases the system may use multiple-choice questions.

For example, imagine an expert system for repairing a dishwasher (again, the user's responses are italicized):

Does the dishwater drain properly?
No.

Does the water drain out at the wrong time?
Yes.

Can you examine the output drain hose?
Yes.

Does the drain hose loop back as high as the top of the dishwasher?
Yes.

Do you have an antisuction valve in the drain line?
No.

Natural Language Processing

Anyone who has used a computer soon becomes frustrated by the fact that the computer uses one type of language and humans use another. If computers are so intelligent, why can't we communicate problems to the computer in a human language and let the computer convert this to its own language? Once the computer has solved a problem, why not have the computer convert the answer back to human-language form?

Using Prolog, researchers are beginning to help computers do just that. Knowledge about a human language is expressed in formal rules and facts using Prolog. The computer can then begin a dialog with the user (often called a *consultation*), asking questions about the problem and interpreting the user's answers.

Often such *natural language processors* are part of expert systems, permitting nontechnical users to describe problems and resolve them. Consider the example of an expert system with a natural language interface that is designed to aid in repairing a dishwasher. The dialog with the computer might go like this (user responses are in italics):

> What seems to be the problem with the dishwasher?
> *The water drains out before the wash cycle is completed.*
>
> Can you examine the output drain hose?
> *Yes.*
>
> Does the drain hose loop back as high as the top of the dishwasher?
> *Almost to the top.*
>
> Do you have an antisuction valve in the drain line?
> *I don't know what an antisuction valve is.*

Notice the difference between this and the previous example. The user is not restricted to yes and no. The dialog is more natural and "human."

Although natural language processors require too much memory and processing power for today's microcomputers, the concept is very practical for a faster processor if enough memory is available.

Eventually, as faster processors and more memory becomes available, developers will put the knowledge needed to solve problems of a specific type on a small plastic disk called a *CD-ROM*. The CD-ROM will probably be much cheaper than your last service call from Sears.

Robotics

Robotics is a branch of artificial intelligence concerned with enabling computers to see and manipulate objects in their environment. Robotics

research is primarily involved in studying and developing sensor systems, manipulators, and controls, and heuristics for solving object- and space-oriented environmental problems.

Prolog facilitates the development of robotic control programs that can use input data from sensors to control manipulators. A robot can then apply formal reasoning to decisions, much as a human being does.

Gaming and Simulations

Prolog is ideal for games and simulations (cognitive modeling) involving formal reasoning. Most games and simulations employ a set of logical rules that control the play or action, and these are very adaptable to Prolog programming. Prolog makes it possible to test various heuristics against a particular control strategy. Many classic games, such as ELIZA, Towers of Hanoi, the N Queens Problem, and some versions of Adventure, are written in Prolog.

COMPILER OR INTERPRETER?

Computer programs are written in the source code of a particular language. A *compiler* or *interpreter* must be used to convert the source code to the computer's own language. To use any particular language, then, you need either a compiler or interpreter for that particular language.

If language conversion is performed using an interpreter, the interpreter must remain in memory with the source code. Each time the program is executed, the interpreter reads each line of code, interprets it, and then executes it. Interpreters are notoriously slow. Another disadvantage is that because they remain in memory with the source code, they occupy precious memory space. A third disadvantage is that the user must have a copy of the interpreter to execute the program.

Interpreters do have an advantage, however. The interpreter is always in control of program execution; programs can be stopped, examined, edited, and restarted. Often data can be altered during a program pause, or a program can be restarted from where it was stopped.

A compiler, in contrast, converts source code to a program that will run on its own. Then the compiler is no longer needed, unless the program is changed; the user needs nothing but the compiled program. The resulting program is fast, and memory is used only for the program and for data.

Turbo Prolog is a true compiler; it compiles source code into a true executable form. You can use Turbo Prolog to compile programs that will run without Turbo Prolog, and then you can sell your programs without paying

royalties to Borland International. You also have the execution speed of a true compiler.

FEATURES OF TURBO PROLOG

Turbo Prolog is a Prolog compiler developed and marketed by Borland International for the IBM PC and compatible computers. It contains most of the features of the Clocksin and Mellish Prolog, which is considered the definitive Prolog. It acts and functions as a compiler to a user. The compiled programs execute faster than any other Prolog programs on the market for the IBM PC and compatible computers.

Turbo Prolog offers the following features:

- You can compile stand-alone programs that will execute on a machine that is not running Turbo Prolog. These stand-alone programs can be sold or distributed to users without paying any royalties to Borland International.
- A full complement of standard predicates for many functions such as string operations, random file access, cursor control, graphics, windowing, and sound are available.
- A functional interface to other languages is provided, allowing procedural language support to be added to any Prolog system.
- Declared variables are used to provide more secure development control.
- Both integer and real (floating-point) arithmetic are provided.
- An integrated editor is provided, making program development, compilation, and debugging very easy.

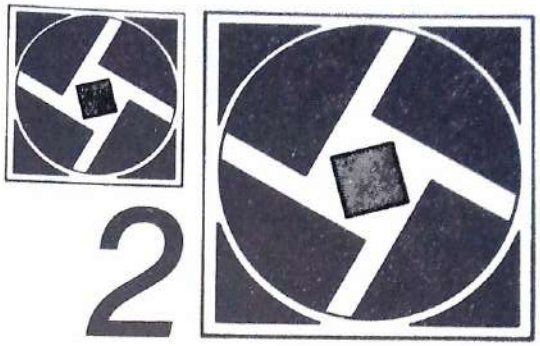
LIMITATIONS OF TURBO PROLOG

Turbo Prolog in some ways reflects the structural aspects of the procedural languages. The variable declaration requirement and the division of the program into sections (see Chapter 3) impose some limitations on Turbo Prolog symbolic processing that do not exist for more traditional Prolog languages.

The current version of Turbo Prolog does not support virtual memory. In a virtual-memory Prolog, program size is limited only by disk space. The size of systems developed with Turbo Prolog, however, is limited by the amount

of memory available. You can use random-file access to overcome this limitation to a degree, but memory-space control is always the responsibility of the user, in contrast to true virtual-memory Prologs, such as Arity Prolog, which manages storage automatically.

Turbo Prolog, like all Prologs, is inefficient for numerical processing. You will find you must implement any procedural programming with crude structures compared with those used by a procedural language.



2

Before looking closely at the Prolog language, let us create, edit, and execute a simple program using Turbo Prolog. This chapter will introduce you to the basic mechanical aspects of using Turbo Prolog and take you through the program creation, editing, and execution processes. Do not worry if you do not understand how Prolog works at this point. What this chapter teaches you is how to use the basic Turbo Prolog commands to create functioning programs.

STARTING TURBO PROLOG

To start Turbo Prolog from a hard disk, switch to the subdirectory of Prolog programs and enter

```
C>PROLOG ↵
```

If you are not using a hard disk, place a working diskette in drive A and enter the same word. (For information about installing Prolog, see Appendix A.)

Turbo Prolog will then load and display a copyright screen with a request to press the space bar. Press the space bar to complete the start-up operation. You will then see a screen with four windows and a menu bar at the top containing seven options (Figure 2.1). Take a few minutes to examine this screen closely.

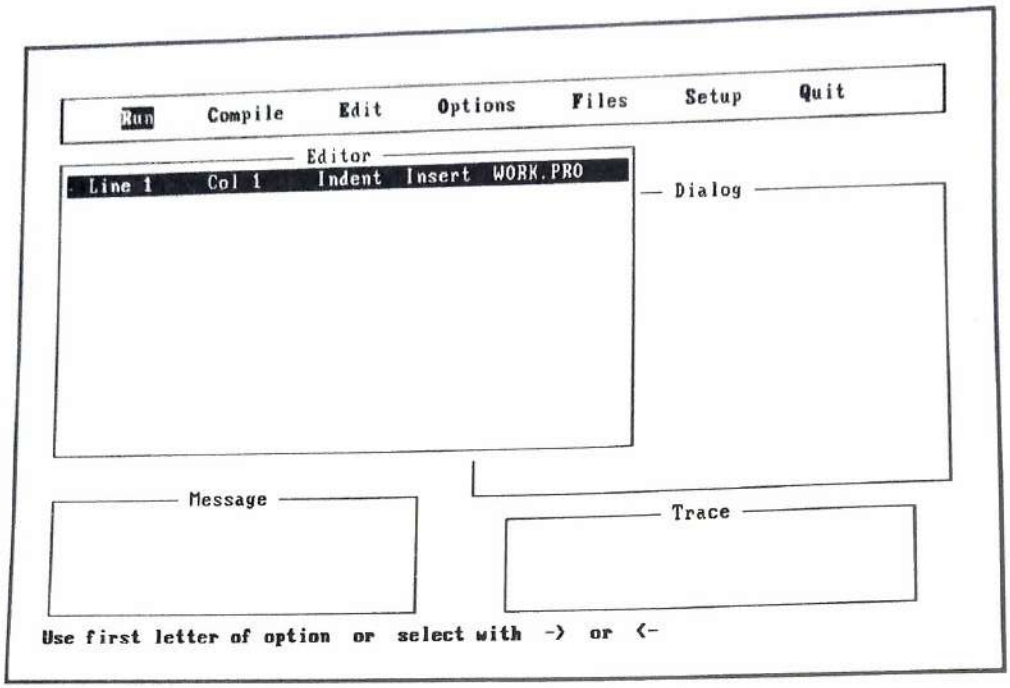


Figure 2.1: Starting the Prolog program

The Four Windows

Four windows are displayed: Editor, Dialog, Message, and Trace. The Editor window is used to create or edit programs. When a Prolog program is executing, output will be in the Dialog window. The Message window keeps you up to date on processing activity. The Trace window is useful for finding problems in the programs you create.

All four windows can be moved to any place on the screen. You can also make any of them smaller or larger and overlay one on top of another. You can change the windows to meet your specific needs and then save the new configuration to use in later sessions. You will see how to do this later in this chapter.

The Seven Options

At the top of the screen a menu bar shows seven options: Run, Compile, Edit, Options, Files, Setup, and Quit. Notice that the default selection is Run. This option will be of little use at the moment, as no program is loaded to run.

To select an option you can either move the highlight to that particular option with the arrow keys and press ↵ or press the first letter of the option. For example, to quit Prolog you could either move the highlight to Quit and press ↵ or just press Q.

Some of the options have additional menus. For example, press *F* (for Files), and you will see a pop-up menu appear (Figure 2.2). Some submenu options also have further menus that pop up when the option is selected.

If you do not wish to make a selection, you can always use the Esc key to back up through the menus you have selected. For example, press the Esc key now, and the Files menu will disappear, backing you up to the original screen and main menu.

Each option on the menu bar has a specific function:

Edit	Creates a new program or changes a program that is currently in memory.
Files	Loads a program from the disk to memory or saves a program currently in memory to the disk.
Compile	Compiles a program in memory for execution.
Run	Compiles *and* runs a compiled program.
Options	Selects the type of compilation to use.
Setup	Changes the setup parameters (such as window sizes or positions) and saves the new values for use in subsequent sessions.
Quit	Aborts Turbo Prolog and returns to the operating system.

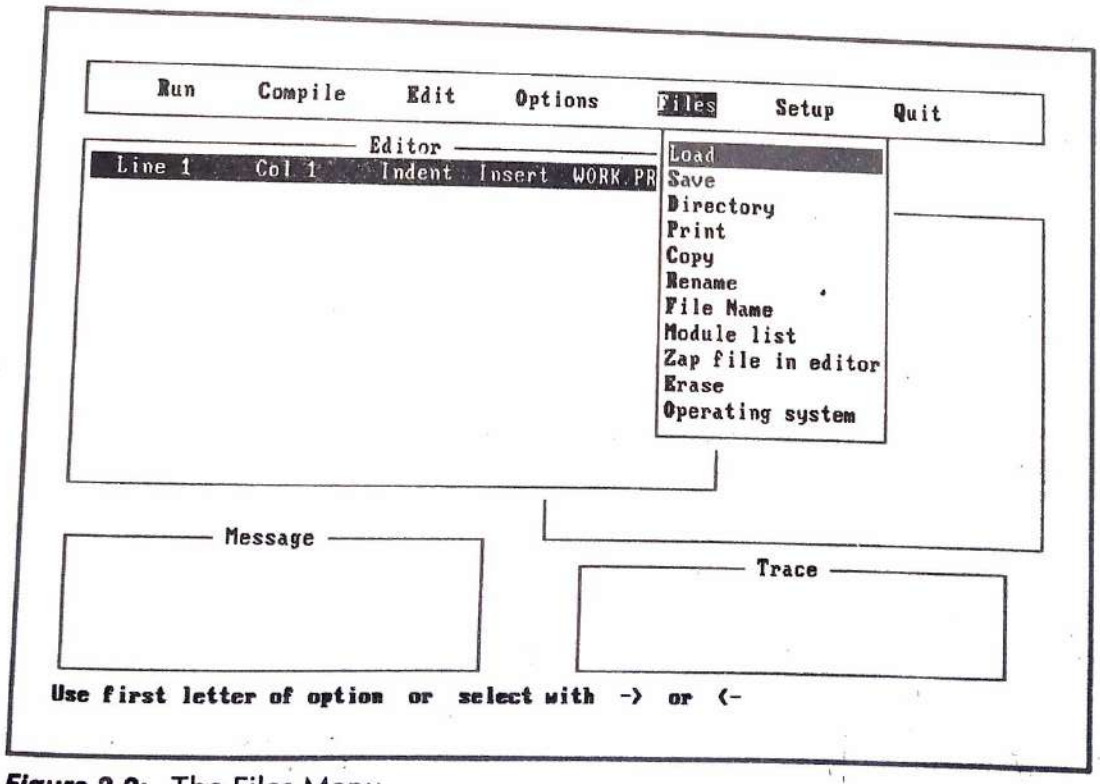


Figure 2.2: The Files Menu

CREATING A SAMPLE PROGRAM

The basic steps for creating and executing a program are

1. Select the Edit mode and enter the program.
2. Enter text or make corrections as necessary.
3. Save the program to disk using the Files Save option.
4. Compile the program.
5. Execute the program.

The compilation process converts the program into a special form that executes very quickly. The original (or *source*) text of the program also remains in memory. You can always return to the Editor window and modify, compile, and execute your program again because Turbo Prolog, your source program, and the compiled version of your program are all in memory at the same time.

Now try to enter, compile, and execute the simple Prolog program shown in Figure 2.3. Select the Edit mode. The cursor will move to the Editor window, and the window will be clear. Enter the program shown, using the ↵ key to place a carriage return at the end of each line. Be sure to include the two periods.

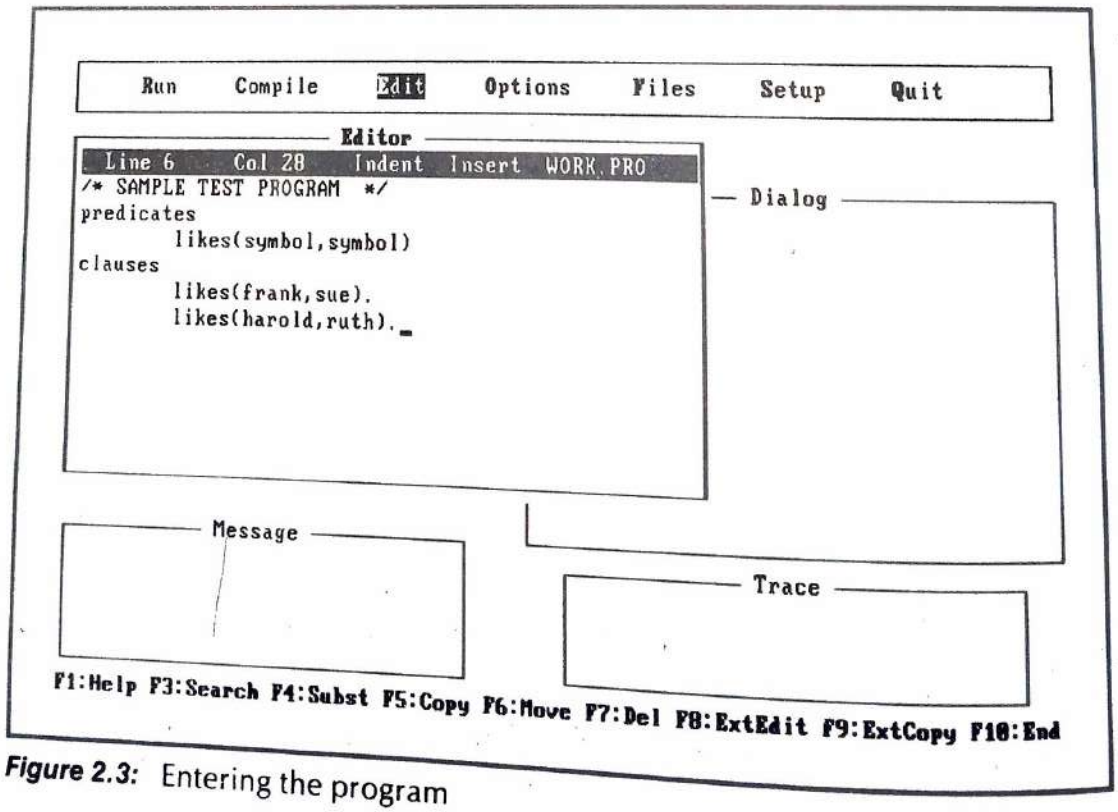


Figure 2.3: Entering the program

Note: As you enter the text, you will notice that indentation is controlled automatically. Each time you press ⟵, the cursor moves to a starting point on the next line that corresponds to the starting point of the previous line. There is no "word-wrap"; a long line causes the screen to scroll or "pan" to the right. Only ⟵ starts a new line. To move the cursor to the left, use the Backspace key. To move the cursor to the right, use the Right Arrow or Tab key. To turn the automatic indentation off, use Ctrl-QI.

If you make a mistake, ignore it for now and continue your entry. You will learn how to correct your mistakes in the next section.

Editing

You can easily edit your program and correct any mistakes. If you have used WordStar, you will see that Prolog's editing commands work almost the same as WordStar's. You can insert characters, delete characters, move and copy blocks, and perform search and replace operations. A summary of the commands appears in Table 2.1. Let us take a look at the basic aspects of editing.

Function	Command
Cursor movement	
Character left	← or Ctrl-S
Character right	→ or Ctrl-D
Word left	Ctrl-← or Ctrl-A
Word right	Ctrl-→ or Ctrl-F
Line up	↑ or Ctrl-E
Line down	↓ or Ctrl-X
Page up	PgUp or Ctrl-R
Page down	PgDn or Ctrl-C
Beginning of line	Home or Ctrl-QS
End of line	End or Ctrl-QD

Table 2.1: Turbo Prolog editing commands

Function	Command
Top of file	Ctrl-PgUp or Ctrl-QR
End of file	Ctrl-PgDn or Ctrl-QC
Beginning of block	Ctrl-QB
End of block	Ctrl-QK
Insert/delete	
Insert mode toggle	Ins or Ctrl-V
Delete left character	Backspace
Delete character under cursor	Del
Delete line	Ctrl-Y
Delete to end of line	Ctrl-QY
Block commands	
Mark block begin	Ctrl-KB
Mark block end	Ctrl-KK
Copy block	F5 or Ctrl-KC
Move block	F6 or Ctrl-KV
Delete block	F7 or Ctrl-KY
Read block from disk	F9 or Ctrl-KR
Hide-display block	Ctrl-KH
Miscellaneous	
Call auxiliary editor	F8
Go to line	F2
End edit	Esc or F10
Auto indent toggle	Ctrl-QI
Find	F3 or Ctrl-QF
Repeat last find	Ctrl-L
Find and replace	F4 or Ctrl-QA

Table 2.1: Turbo Prolog editing commands (cont.)

Cursor Movement

You can use the arrow keys to move the cursor anywhere on the screen one character or line at a time to make a correction. The Right Arrow key moves the cursor one character right, and the Left Arrow key moves the cursor one character left. The Up Arrow key moves the cursor one line up, and the Down Arrow key moves the cursor one line down.

You can also use the WordStar cursor-movement commands:

Ctrl-S	Move one character left.
Ctrl-D	Move one character right.
Ctrl-E	Move one line up.
Ctrl-X	Move one line down.

If you locate these letters on the keyboard, you will see they form a small diamond, with the corners of the diamond corresponding to the direction of movement. This makes remembering the codes easy.

You can move a whole word left or right by pressing the Ctrl key along with the Left or Right Arrow key. You can also use Ctrl-A to move a word left and Ctrl-F to move a word right. Locate these keys on the keyboard and see where the letters are with respect to the diamond.

You can also move a page of text at a time. Use the PgUp key or Ctrl-R to move backward a page at a time. Use the PgDn key or Ctrl-C to move forward a page at a time.

To move quickly to the top of the file (to the beginning of the program), use Ctrl-PgUp or Ctrl-QR (press Ctrl and Q simultaneously, release both, and then press *R*). To move quickly to the end of the file (the end of the program), use Ctrl-PgDn or Ctrl-QC.

Home (or Ctrl-QS) will take you to the beginning of the current line. End (or Ctrl-QD) will take you to the end of the current line.

Experiment with the cursor-control keys using the program currently displayed. Moving the cursor does not change any text or alter the program.

Inserting Text

There are two modes for text insertion: Insert mode and Overwrite mode. The Ins key (or Ctri-V) acts as a toggle, switching the user between the two modes. In Insert mode, text is inserted by positioning the cursor where you wish to add the text and then typing the new text. Any characters entered are placed at the cursor location, and text to the right of the new characters is moved over to make room. In Overwrite mode, characters entered overwrite, or replace, characters under the cursor, and no text is moved. The Insert mode is the default mode for Turbo Prolog.

Deleting Text

Two keys are available for text deletion: Backspace and Del. The Backspace key deletes the character to the left of the current cursor location. The Backspace key is most useful for making corrections as you enter text to delete characters you have just typed.

The Del key deletes the character above the cursor. It is most useful for deletions during editing. You position the cursor under the character you want to delete and then press the Del key.

You can use Ctrl-Y to delete the entire current line. Ctrl-QY deletes the portion of the line to the right of the cursor.

If you wish to delete the entire program from the work area, first exit the editor by pressing F10 or Esc. If a program is important, be sure to save it on disk before you delete it from the computer's memory (see "File Operations: Saving and Loading Programs" in this chapter). Then select the Files option. Finally, from the Files menu select Zap file in editor. The program then prompts for a confirmation. Enter *y* if you really want to delete the program from memory. The program is then cleared from memory, but Turbo Prolog queries to see if the program file on the disk should be deleted as well. You will be prompted with the current file name for the program on the disk. To avoid zapping the disk file, press Esc. You will then be back in the Edit mode with a blank Editor window.

> ***Note:*** Exercise caution in using the Zap file in editor command. It normally is used to delete both the program in memory and the corresponding file on the disk. If you want to delete only the program in memory, terminate the operation after the memory is cleared by pressing the Esc key when the file name is displayed.

Block Operations

Sometimes you may need to perform an operation with a block of text. Typical block operations include erase, move, and copy. Block operations require two steps: defining the block and initiating the operation.

Blocks are defined the same way for any operation. Place the cursor at the beginning of the block and press Ctrl-KB (press Ctrl and *K* simultaneously, release both, and then press *B*). Then move the cursor to the end of the block and press Ctrl-KK. The easiest way to remember this is to remember that Ctrl-K always initiates the marking process (and most block operations). Then, for the second letter of the command, the word *block* is a mnemonic, with *B* marking the beginning of the block and *K* marking the end. If you performed this operation correctly, the entire block will be reduced in intensity

compared to the rest of the text on the screen. (For simplicity, this book will refer to the marked block as *highlighted*.)

Once the block is marked, you can erase it by pressing Ctrl-KY. The block will be deleted, and the text below the block will be moved up into the vacated space.

If you wish to create a copy of the block at a new location and keep the original at the old location as well, mark the block and then move the cursor to the new location. Press Ctrl-KC. The block will then be at both locations. To make multiple copies of a marked block, press Ctrl-KC repeatedly.

If you wish to move a block, mark the block and then move the cursor to the new location. Press Ctrl-KV. The block will disappear at the old location and appear at the new location.

As you can see from Table 2.1, there are function key equivalents of these block commands. If you use these, you don't mark the block first; you just press the function key and follow the prompts.

You can move the cursor quickly to the beginning of the current marked block using Ctrl-QB. You can move the cursor quickly to the end of the marked block using Ctrl-QK. The current block can be toggled from highlighted to not highlighted mode using the Ctrl-KH keys. When the block is not highlighted, the other block commands will not work on it.

To write a copy of a block to a disk as a separate file, use Ctrl-KW. The program will prompt you for a file name.

To read blocks from the disk to your program, use the F9 key or Ctrl-KR. With Turbo Prolog, you can move a portion of a stored file to the current cursor position in the Editor window. First, position the cursor where you wish to add the text. Press the F9 key, and the program will prompt you for a file name. Enter the file name, and the file will be loaded in a new, auxiliary Editor window. You will then see a prompt for each step. Mark the beginning and end of the block you wish to move from the auxiliary window using the F9 key. The auxiliary window will disappear, and the block will appear at the cursor location in the Editor window.

You can clear the work area by marking the entire program as a block and deleting it, but it is easier to exit the Editor window and use the Files menu (see the previous section).

Search and Replace

If you wish to search for a particular character string in the program text, initiate the request with the F3 key or Ctrl-QF. When you see the prompt, enter the character or characters for which you wish to search. Then press F3 or Ctrl-QF again. The cursor will move forward to the first occurrence of the text string.

The search is always made forward from the current cursor location. Thus, to search for the first occurrence of a text string in the program, the cursor should first be positioned at the beginning of the program before F3 is pressed. You can repeat any search using the same search string and Shift-F3. Because the ↵ key is not used to terminate the text string, you are free to make the carriage return a part of the search string as well as any other character, such as a tab.

The replace operation works in the same way except that F4 or Ctrl-QA is used. The editor then requests both a search string and a replacement string. You are also asked if the replace operation should be global (all occurrences of the string replaced) or local (only the first occurrence replaced). The replace operation can be repeated using Shift-F4.

Miscellaneous Functions

While in the Editor window, you can open another Editor window using the F8 key. You can then load or create a block in this window. You can use the F10 key to close this auxiliary window.

You can go to any specific line in the program by pressing F2 and then entering the line number. Press F2 again. This will move the cursor to the specified line.

Getting Help

If you get stuck at any point but you know the corresponding WordStar command, try the WordStar command. You may need an extra keystroke to accomplish your objective (compared with using a function key), but this approach often works.

You can also use the F1 key in the Editor window as a help key. This displays the help file, PROLOG.HLP. If you wish, you can edit this file to meet your specific needs.

Exiting the Editor

There are two basic ways to exit the Editor window: press Esc or press F10. Most of the time both ways work the same. Either exits the Editor to the main menu, permitting you to select another option from the menu. The only exception occurs when Turbo Prolog finds an error during the compilation process. In this case you are automatically returned to the Editor. After the error is corrected, pressing F10 forces a new compilation and execution of the program. Using Esc would simply return you to the main menu.

File Operations: Saving and Loading

As a Prolog program is created, it is created in the computer memory. If the computer should fail for any reason or be rebooted, the entire program would be lost, and you would not be able to recover it. For this reason, you should always periodically save a program as a file on a diskette or hard disk.

You cannot save a program as a file from within the Editor. You must always exit the Editor using Esc before saving the program. Because of the vulnerability of computer memory, you should make a practice of pausing every 15 or 20 minutes, exiting the Editor, saving the program, and then returning to the Editor. Make this a discipline until it becomes a habit. Eventually you will be very, very grateful you have done so.

Here are the general steps for saving a program:

1. Exit the Editor using Esc. The program will still be displayed in the Editor window.
2. Select the Files option from the main menu by pressing *F* or by moving the highlight to the Files option and pressing ↵.
3. Select the Save option from the Files menu by pressing *S* or by moving the highlight to the Save option and pressing ↵ (Figure 2.4).

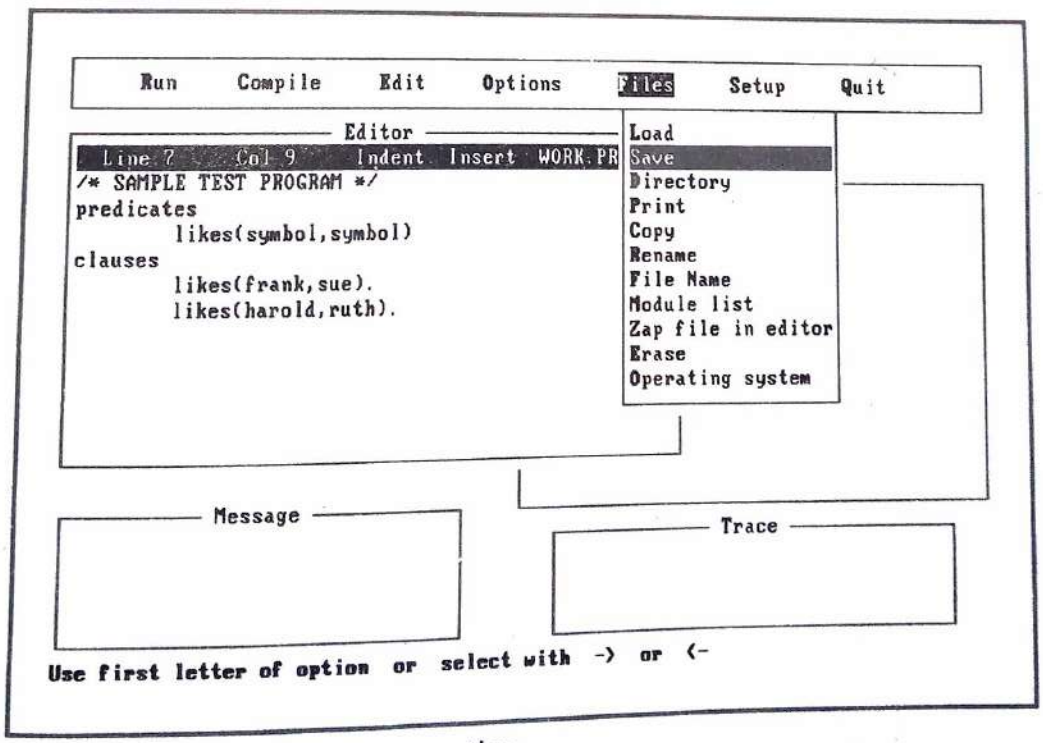


Figure 2.4: Starting the save operation

4. Enter the file name when requested to do so. You can use from one to eight characters. It is not necessary to enter an extension name. Press ↵ to complete the file name entry (Figure 2.5).
5. The file will be saved with the name you entered and the .PRO extension appended.
6. Press Esc to return to the main menu.
7. If you wish to continue editing, select the Edit option again.

Once a program is saved as a file, you can always load it again to the Editor area using the File Load command. Select Files from the main menu and then select Load from the Files menu. When the prompt appears, enter the file name (without the extension) to load it. The file will then be loaded to the work area.

Note: If a file is loaded to the work area, it will overwrite and destroy the previous contents of the work area. If you wish to keep what is in the work area, be sure to save it to the disk before loading a new file.

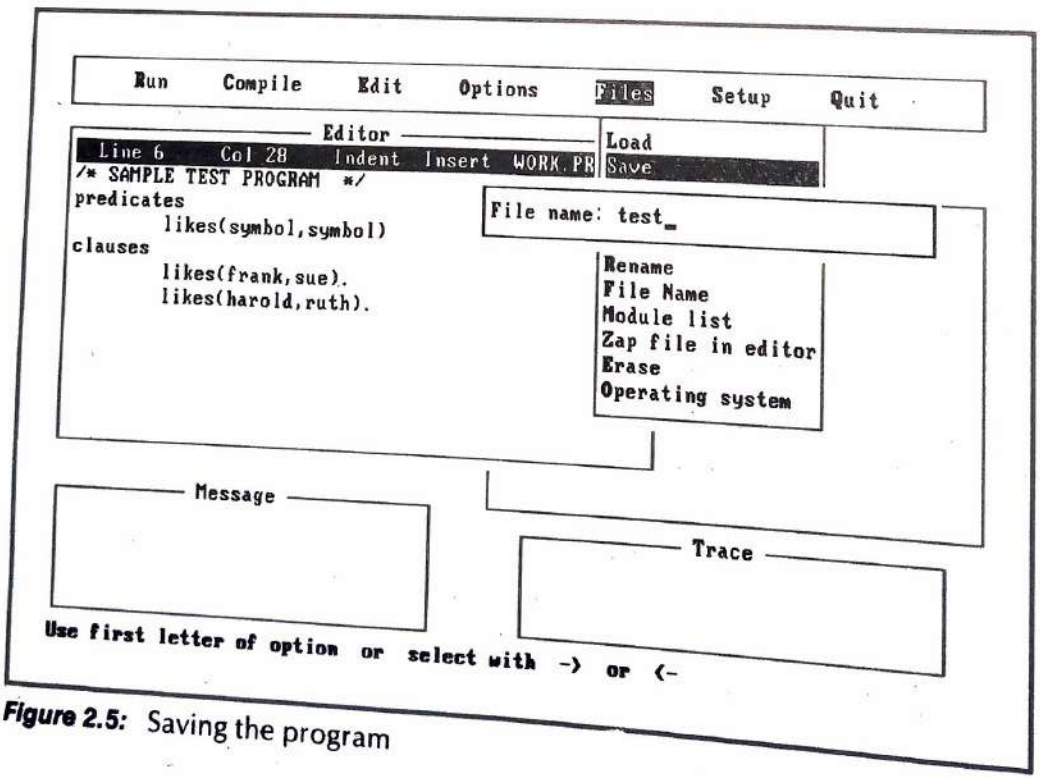


Figure 2.5: Saving the program

Here are several other options on the Files menu you will find useful:

Directory	Displays the contents of the current directory without leaving Turbo Prolog.
Print	Prints a copy of the program that is in the Editor window.
Copy	Copies a file on the disk.
Rename	Changes the name of any disk file.
File name	Changes the name of the file in the Editor window.
Module list	Names a module file that lists all modules in a given system of programs; useful for working with large programs.
Zap file in editor	Deletes the entire program in the Editor window (see previous section about deleting).
Erase	Deletes a file from the disk.
Operating system	Permits you to exit to DOS and execute a command with Turbo Prolog and your program remaining in memory; to return to Turbo Prolog, enter *exit*.

Now It Is Your Turn

If you have not yet used the Turbo Prolog Editor, take some time to do so now. First, make corrections in your simple program, exit the Editor, and save your program. Then return to the Editor and try commands you did not use the first time, working through a few block operations and search and replace operations. Continue until you feel comfortable using the Editor.

When you feel comfortable with the Editor, exit it and reload the corrected copy of your program to prepare for the next section.

COMPILING AND EXECUTING

To compile and execute your program, exit the Editor using F10 or Esc. Then select the Run option from the main menu. The program will compile automatically. You will see a few messages in the message screen. If there are no errors, you will see a prompt in the Dialog window (Figure 2.6).

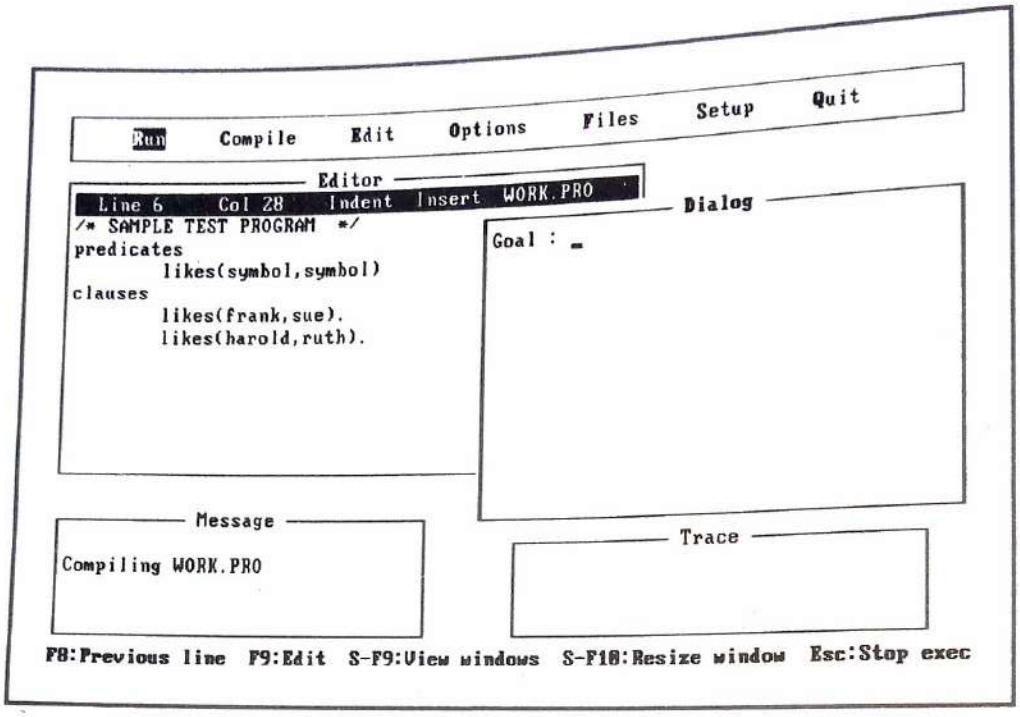


Figure 2.6: Starting to execute the program

If the program contains errors, Turbo Prolog will return you to the Editor. You can then make changes and compile the program again by once more exiting with F10.

When you see the goal prompt, enter the desired goal. As an example, try

```
Goal : likes(harold,ruth) ↵
```

Turbo Prolog should respond with

```
True
Goal :
```

This indicates Prolog was able to prove the goal you entered as true using the facts it had available.

You can stop program execution at any time by pressing the Esc key. This will return you to the main menu. Save the program if you have made changes since you last saved it.

CHANGING THE DEFAULT WINDOWS

You have probably already discovered that the default window sizes are not adequate for your work. For most examples in this book, you will want

an Editor window that is as wide as the screen. You will also want a Dialog window that is as wide as the screen. No problem—with Turbo Prolog you can have both. If you do this, Turbo Prolog will always put the active window on top. When you are editing, the Editor window will cover the Dialog window and extend across the entire screen. Once the program is executing, the Dialog window will cover the Editor window.

To change the window sizes, select the Setup option from the main menu. Choose the Window option. Select the Window Size option from the Window menu and press ←┘. You will then see a menu from which you can select the window to resize and a menu at the bottom of the screen that is useful for moving and sizing windows (see Figure 2.7). Notice that the resizing is done with the arrow keys.

Select the window to be changed and then move or change the size of the window. Notice that the arrow keys change the size of the window; using Shift with the arrow keys changes the position of the window. Sometimes you must move a window before you can change its size. For example, the Dialog window must be moved left before it can be made wider, because windows can only grow to the right. Change each window as necessary. You can return to the main menu using multiple strokes of the Esc key.

Once you have finished resizing the windows, you can save the new configuration (if you wish) to use automatically in subsequent sessions. To save the configuration, select the Setup menu again and then the Save option.

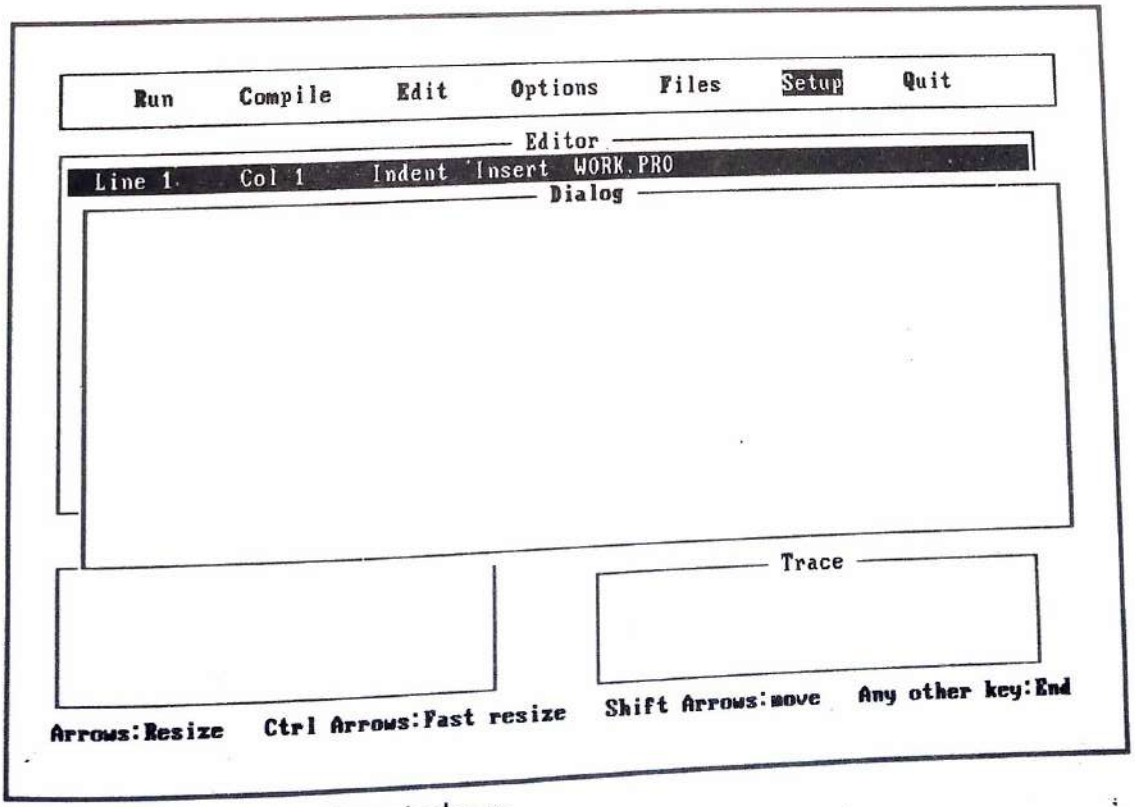


Figure 2.7: Changing the windows

This will save the configuration in the PROLOG.SYS file, and the same configuration will then be used each time Turbo Prolog is started.

CHANGING THE DEFAULT DISK DIRECTORY

Turbo Prolog normally defaults to the current directory for the configuration file, program files, and other types of files. You can change the default directories for any of five types of files using the Setup option from the main menu. On the Setup menu, select Directories. From the Directories menu select the default directory you want to change.

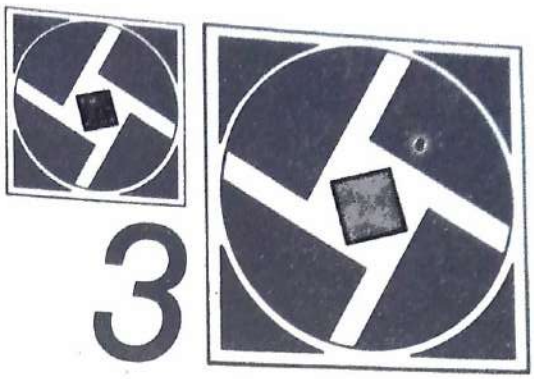
The change will apply only to the current session. If you wish to save the new default directories for subsequent Turbo Prolog sessions, select the Setup option from the main menu and then select the Save option.

CHANGING THE BACKGROUND

On some IBM PC compatibles, Turbo Prolog defaults to an undesirable background color or monochrome shade in some of the windows. If this is true of your system, change the background color or shade using the Setup menu. The Colors option changes the background and foreground colors on a color monitor, and it can also control the contrast on a monochrome monitor. The Miscellaneous Settings option permits you to turn the CGA option on or off, which can also affect the contrast on some systems.

IN CASE OF TROUBLE

If you get lost with Turbo Prolog and need to return to the main menu, use Ctrl-Break. This will either return you to the main menu or to a prompt asking you to press the space bar, which then returns you to the main menu.



3

The main part of a Prolog program consists of a collection of knowledge about a specific subject. This collection is called a *database*, and this database is expressed in facts and rules. Let us look at how facts are expressed.

EXPRESSING FACTS

Turbo Prolog permits you to describe facts as symbolic relationships. For example, if the right speaker in your stereo system is not emitting sound (is dead), you can express this in English as

The right speaker is dead.

This same fact can be expressed in Turbo Prolog as

```
is(right_speaker,dead).
```

This factual expression in Prolog is called a *clause*. Notice the period at the end of the clause.

In this example *right_speaker* and *dead* are *objects*. An object is the name of an element of a certain type. It represents an entity or a property of an entity in the real world.

Although Turbo Prolog permits the user to use any of six different object types, for the moment we will look at only one of these types: the symbol object. The name of a symbol type object in Prolog always begins with a lowercase letter. The remaining characters in the name can be uppercase or lowercase letters, numbers, or an underscore. From 1 to 250 characters can be used in a name. Here are some examples of valid symbol object names:

```
right_speaker
dead
jack
automobile
red
age12
```

The word *is* is the *relation* in the example. A relation is a name that defines the way in which a collection of objects (or objects and variables referring to objects) belong together. Here are a few more examples:

Fact	Relation
has_a(bill,computer).	has_a
is_a(collie,dog).	is_a
likes(sue,chocolate).	likes

The entire expression before the period in each case is called a *predicate*. A predicate is a function with a value of true or false. Predicates express a property or a relationship. The word before the parentheses is the name of the relation. The elements within the parentheses are the *arguments* of the predicate, which may be objects or variables. Here are a few examples of predicates:

```
employee(bill)
eligible(mary)
marital_status(joyce,married)
```

If you add a period to a predicate, you have a complete clause.

You can also express facts using compound predicates. For example, you can express the fact that Mary is married and eligible for employment in this clause:

```
eligible(mary) and marital_status(mary,married).
```

So far all the clauses we have looked at have been facts. There is another category of Prolog clauses called *rules*, which will be discussed later. But

based on what we already know, the relationship of clauses, predicates, relations, and objects is summarized in Figure 3.1.

TURBO PROLOG OBJECTS

Before continuing, let us look at a few more examples of facts expressed as Prolog clauses:

English: Bill is an employee.
Prolog: **employee(bill).**

English: Bob is married to Mary.
Prolog: **married_to(bob,mary).**

English: The speaker is defective.
Prolog: **diagnosis(speaker,defective).**

English: The speaker is defective.
Prolog: **defective_speaker.**

English: Tom is a student.
Prolog: **student(tom).**

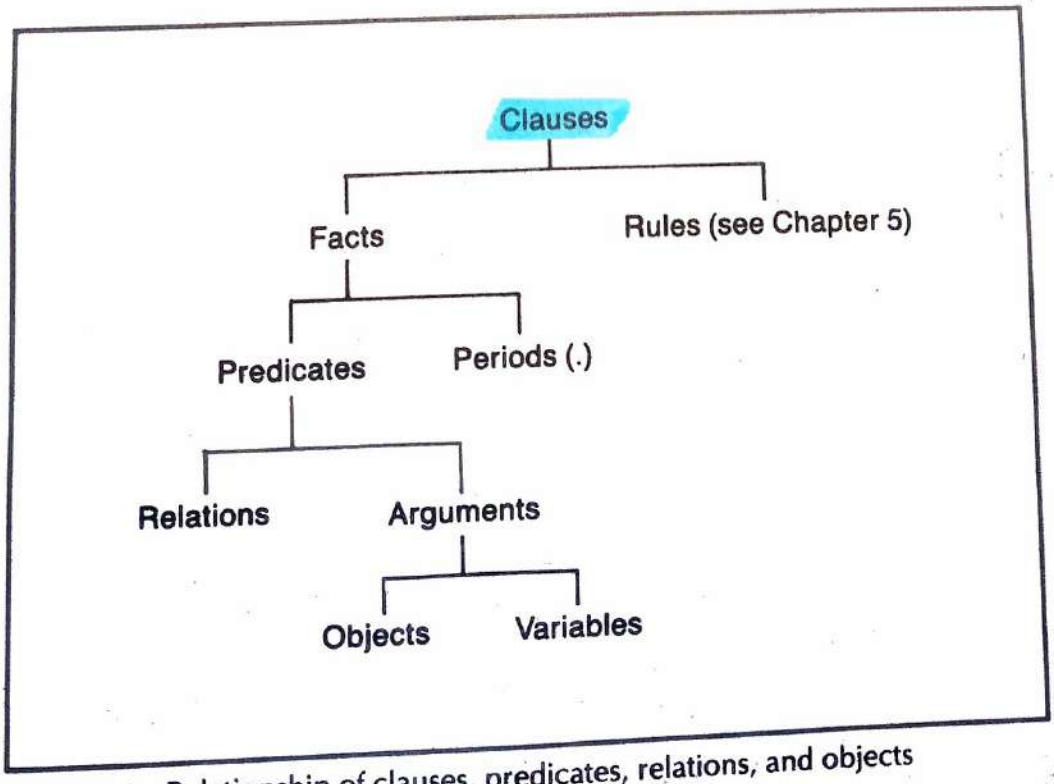


Figure 3.1: Relationship of clauses, predicates, relations, and objects

English: Joseph is ten years old.
Prolog: **age(joseph,10).**

English: Bob likes automobiles.
Prolog: **likes(bob,automobiles).**

English: Bob likes automobiles and computers.
Prolog: **likes(bob,automobiles) and likes(bob,computers).**

Notice several things about these clauses:

- A given relation can have any number of objects. A predicate can have any number of arguments, including zero. The number of arguments is called the *arity* of the predicate.
- An object name can represent a physical entity (Bob, automobiles) or an abstract concept (defective). It can be a noun, adverb, or adjective. Indeed, it can be any sequence of characters that abides by the rules for the object type.
- Objects are always singular. In the last example above, *automobiles* is considered a singular object.
- Certain names are reserved in Turbo Prolog and should not be used for object names. These are listed in Table 3.1.
- The Prolog expression does not have to contain all the words of the English expression.

A Prolog expression is a symbolic extension of an English expression. You can define a relation in any way you wish, but you should follow a consistent pattern of word order in a given program. For example, you could express the relation "Tom likes Janet" as

```
likes(tom,janet).
```

or

```
likes(janet,tom).
```

The latter expression does not necessarily mean "Janet likes Tom"; it shows how a relation can be defined by the user in whatever way the user wishes, so long as the user follows a consistent pattern in a given database.

Objects in Knowledge Representation Theory

If you have read books on expert systems using Prolog, you may have come across another type of object that you should be careful not to confuse

abs	domains	nl
and	dott	nobreak
arctan	edit	not
asserta	editmsg	nowarnings
assertz	eof	openappend
attribute	existfile	openmodify
back	exit	openread
beep	exp	openwrite
bios	fail	or
bitand	field_attr	pencolor
bitleft	filemode	pendown
bitnot	file_str	penpos
bitor	filepos	penup
bitright	findall	port_byte
bitxor	flush	predicates
bound	forward	project
char_int	free	ptr_dword
check_determ	frontchar	random
clauses	frontstr	readchar
clearwindow	fronttoken	readdevice
closefile	global	readint
code	goal	readln
comline	gotowindow	readreal
concat	if	reference
config	include	removewindow
consult	inkey	renamefile
cos	isname	retract
cursor	keypressed	right
cursorform	left	round
database	length	save
date	line	scr_attr
deletefile	ln	scr_char
diagnostics	log	scroll
dir	makewindow	shiftwindow
disk	membyte	shorttrace
display	memword	sin
div	mod	sound

Table 3.1: Turbo Prolog reserved names

sqrt	system	upper_lower
storage	tan	window_attr
str_char	text	window_str
str_int	time	write
str_len	trace	write_device
str_real	trail	writef

Table 3.1: Turbo Prolog reserved names (cont.)

with the Turbo Prolog language object. In designing an expert system, the expert abstracts reality to create a mental picture that has both objective and subjective aspects. The knowledge engineer takes the expert's mental abstraction and builds a model called a *knowledge representation*. This knowledge representation is then converted to a program in the Prolog language that behaves, at least in theory, as the real world does (Figure 3.2).

In knowledge representation theory, real-world entities are represented as objects. The objects have attributes and properties, and the attributes and properties have values. For example, a stereo has a tape head (object), the tape head has a property (the status of the tape head), and the property has a value (clean or dirty). So objects are distinguished from properties and their values.

In the Turbo Prolog *language*, an object is something completely different. In Turbo Prolog, an object is the name of *any* constant. The Turbo Prolog object could be an attribute, a property, or a value. It is unfortunate that the Turbo Prolog manual uses the same word as the knowledge engineer, but so long as the user distinguishes between the two, there will be no problem. Study the following example:

Knowledge Representation Theory	**Turbo Prolog**
car -> color -> blue	is(car,blue).
object:car	objects:car, blue

Objects in Procedural Programming

If you have had some experience with procedural languages, you can compare Prolog (an object-oriented language) with these languages. Objects

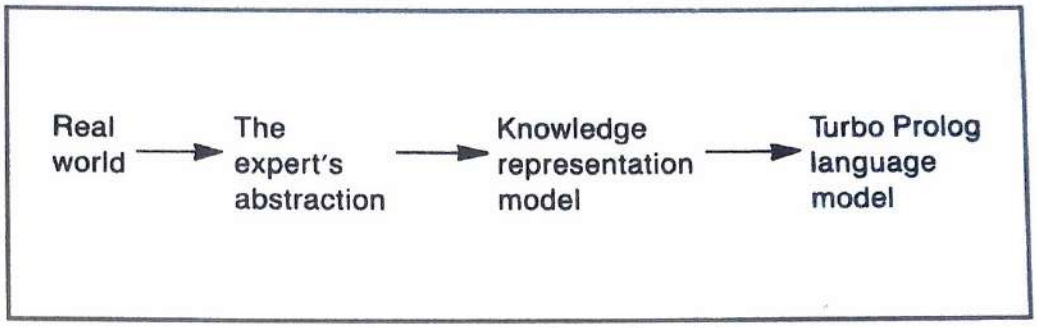


Figure 3.2: Using Turbo Prolog to model the real world

are roughly equivalent to the constants used in procedural programming but are a much more powerful concept. In procedural programming, a constant is a character string or numerical value that remains fixed. In Turbo Prolog, an object is a symbolic extension of a real-world entity.

The predicate in Prolog is comparable to the function in other languages. Predicates, as mentioned, are functions with a value of true or false. The relation is the operator, and the objects are operands for the operator or function arguments.

THE TURBO PROLOG PROGRAM

A Turbo Prolog program consists of two or more sections. The main body of the program, the *clauses section*, contains the clauses and consists of facts and rules. You are already familiar with clauses that express facts. In the next chapter you will learn about clauses expressed as rules.

The relations used in the clauses of the clauses section are defined in the *predicates section*. Each relation in each clause must have a corresponding predicate definition in the predicates section. The only exceptions are built-in predicates that are an integral part of Turbo Prolog. You will learn about these in Chapter 5. Turbo Prolog also requires that each relation must head at least one clause in the clauses section. A predicate definition in the predicates section does not end with a period. For example, in the clauses section you may have the clause

```
is(right_speaker,defective).
```

In the predicate section, you would need a predicate definition such as

```
is(component,status)
```

Any other names could be used inside the parentheses, so long as the predicate definition contains the same number of names as the predicate does when it appears in the clauses section. The critical part of a predicate definition is specifying the number of arguments the relation will take; the words are merely placeholders.

The *domains section* also is a part of most Turbo Prolog programs. It defines the type of each object. In almost all effective languages, variables and constants (objects) are typed, or *declared,* before they are used. Declaration permits the user to set up the proper storage conditions to control the way the constants or variables are used. Strings (such as *"Box 43"*) are stored differently than integers (such as *4*), and both are stored differently than floating-point numbers (such as *4.3*). In some languages (such as C or Pascal) typing is done with a separate declarations section. In other languages the name itself (such as BASIC names that use an **%** or **$** suffix) defines the type of variable.

> ***Note:*** Most other Prologs do not permit any control of variable or constant typing. Omitting typing control gives the user more flexibility in symbolic operations, but it also poses a great danger. If the user uses an untyped object in the wrong way, Prolog will not know this, and the computer may crash, erase a portion of a hard disk, or create some other problem. By forcing the user to define the type, Turbo Prolog can better control execution. Turbo Prolog can check objects as they are used to be sure they match the type that is specified for that object. If you have used a language (such as other Prologs or FORTH) that does not use typing, you will appreciate Borland International's design.

The domains section of Turbo Prolog controls the typing of objects. Six basic object types are available to the user: char, integer, real, string, symbol, and file (see Table 3.2).

For example, in the clauses section you may have

```
is(right_speaker,defective).
```

In the predicates section, the corresponding predicate would be

```
is(component,status)
```

In the domains section, you would have

```
component,status = symbol
```

char	Single character (enclosed between single quotation marks)
integer	Integer from −32,768 to 32,767
real	Floating-point number ($1e^{-307}$ to $1e^{308}$)
string	Character sequence (enclosed between double quotation marks)
symbol	Character sequence of letters, numbers, and underscores, with the first character a lowercase letter
file	Symbolic file name

Table 3.2: Turbo Prolog domain types

It is technically possible to omit the domains section in many programs by typing the object in the predicates section itself. In this example the predicate would become

```
is(symbol,symbol)
```

If this were done, the domains section would not be needed. For most applications, however, this is not the best way to proceed. It saves the programmer a few keystrokes, but it makes the program harder to read, particularly if it contains many objects.

How an object is used in a program determines how you should type it. If you plan to use the object in arithmetic operations, it should be typed as integer (a whole number) or real (which includes decimal fractions and very large whole numbers). The real type can accommodate any integer, but using the more restrictive type for integers saves storage space and speeds execution. The symbol type is the most general type, but a symbol type name cannot contain spaces or commas and must begin with a lowercase letter. This limits its usefulness for object names, such as addresses that will be printed or displayed. If you plan to print or display the object name, you should use a string type and enclose the object name in quotation marks, as follows:

```
address("John Smith","43 Sater Lane, Suite 2",
"Portland","OR","97212")
```

In a Turbo Prolog program, the sections should always be in the following order:

1. Domains
2. Predicates
3. Clauses

In later chapters you will see that additional sections may be added in certain applications.

USING A SIMPLE TURBO PROLOG PROGRAM

Now let us see how to write and use a simple Turbo Prolog program.

> ***Note:*** This program and other medical diagnostic programs in this book are meant as examples only. Their purpose is to illustrate Turbo Prolog programming principles. They are not endorsed by the medical profession or by any doctor.

Enter and compile the program in Figure 3.3. Review Chapter 2 if necessary. Here is a brief summary of how to proceed:

1. Start Turbo Prolog.
2. Select the Edit mode.
3. Enter the program.
4. Exit the Editor using Esc.
5. Save the program as TEST using the Files Save option.
6. Select the Compile option to compile the program. If you find an error, correct the error and repeat from step 4.
7. Select the Run option to run the program.

The clauses in this example are a simple collection of facts. Start program execution, and you will see the following prompt in the Dialog window:

```
Goal :
```

Turbo Prolog is asking for a goal. A goal is essentially a question. Specify the

```
/* EXAMPLE ONLY          */
/* NOT FOR MEDICAL USE */

domains
        disease,indication = symbol

predicates
        symptom(disease,indication)

clauses
        symptom(chicken_pox,high_fever).
        symptom(chicken_pox,chills).
        symptom(flu,chills).
        symptom(cold,mild_body_ache).
        symptom(flu,severe_body_ache).
        symptom(cold,runny_nose).
        symptom(flu,runny_nose).
        symptom(flu,moderate_cough).
```

Figure 3.3: A simple Turbo Prolog program

following goal:

```
Goal : symptom(cold,runny_nose) ↵
```

Turbo Prolog will then respond with *True* and prompt for another goal:

```
Goal : symptom(cold,runny_nose) ↵
True
Goal :
```

If Turbo Prolog can match the goal with a fact in the database, it "succeeds" and responds with *True*. Each time you specify a goal, one of three possible results will occur:

1. The goal will succeed; that is, it will be proven true.
2. The goal will fail; that is, Turbo Prolog will not be able to match the goal with any facts in the program.
3. The execution will fail because of an error in the program.

Execution is a simple matching process. Turbo starts at the first clause with a predicate that matches the predicate of the goal. Turbo Prolog then scans the clauses from that point seeking a match on the arguments. If a complete match is achieved, the goal succeeds, and *True* is displayed.

Enter the following goal:

```
Goal : symptoms(cold,headache)
```

Turbo Prolog responds with

```
Goal : symptom(cold,headache)
False
Goal:
```

This response does not mean that a headache is not a cold symptom, but simply that Prolog could not find a match in the clauses for the specified goal. In Prolog, *False* indicates a failure to find a match using the current information; it does not necessarily mean that the fact is not true.

Again, Prolog execution is a simple matching process. There is no procedure to follow and no sequence of instructions to provide. At execution time the program simply tries to prove the specified goal through matching.

EXERCISES

1. Identify the relation and arguments in each of the following:

   a. address(bob,portland).

   b. age(tim,12).

   c. computer(ibm_pc).

2. Which of the following are valid symbol object names?

   a. john_smith

   b. John Smith

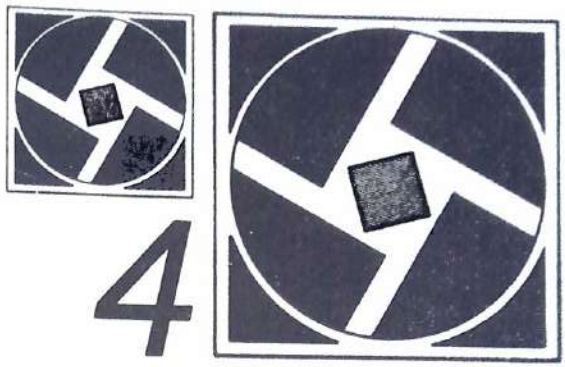
   c. 12

   d. bill*algers

   e. computers

3. Which of the following are valid object names, and what type is each?

   a. "Jane Smith"
   b. jane smith
   c. jane_smith
   d. Age
   e. BillMerchant
   f. billMerchant
   g. 23.45
   h. $43.25



Up to now you have used Turbo Prolog for nothing more than verifying individual facts that are already in a database. Let us see what happens when we add variable capability to this.

THE PROLOG VARIABLE

You can use a variable in a Turbo Prolog clause or goal to specify an unknown quantity. A variable name must begin with a capital letter and may be from 1 to 250 characters long. Except for the first character in the name, you may use uppercase or lowercase letters, digits, or the underline character. Names should be meaningful. Here are some examples of variable names:

```
Age
Disease
Patient
John_Smith
```

Turbo Prolog data types relate as shown in Figure 4.1. (What Turbo Prolog calls *objects* are often called *atomics* in other versions of Prolog, and the symbol object type is often called an *atom*.) The list and compound structure types will be introduced in later chapters. The other data types are discussed here

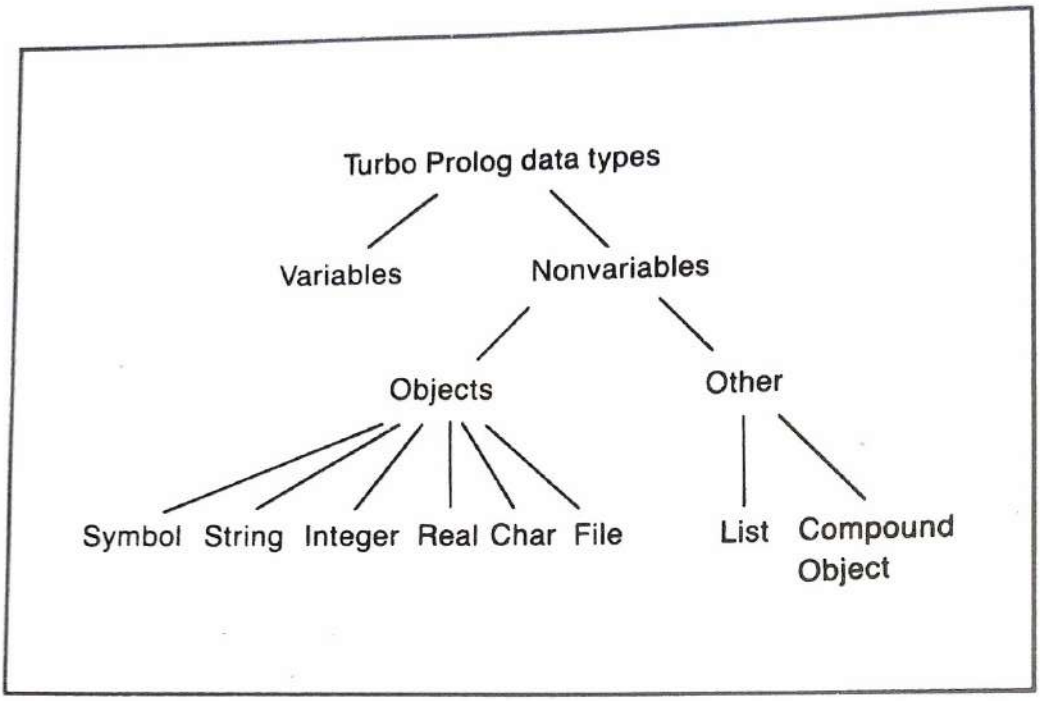


Figure 4.1: Turbo Prolog data types

USING VARIABLES

Recall the program you entered in Chapter 3 (see Figure 3.3). Suppose your patient has a runny nose and you wish to get a list of all the diseases in your data base for which this is a symptom. To do this, you could use the same program and specify the following goal:

```
Goal: symptom(Disease,runny_nose)
```

Notice that this goal includes the variable *Disease*.
Turbo Prolog will respond

```
Goal : symptom(Disease,runny_nose)
Disease = cold
Disease = flu
2 Solutions
Goal :
```

Prolog found two matches for the specified symptom.

Now examine what happens after you enter this goal. As illustrated in Figure 4.2, Turbo Prolog starts at the first clause that matches the specified predicate, which in this case is the first clause in the program. It then tries to

match the object argument in the specified goal (runny_nose) with the corresponding argument in the clause predicate. This match will fail. Prolog then tries the next clause, continuing until it finds this match:

```
symptom(cold,runny_nose).
```

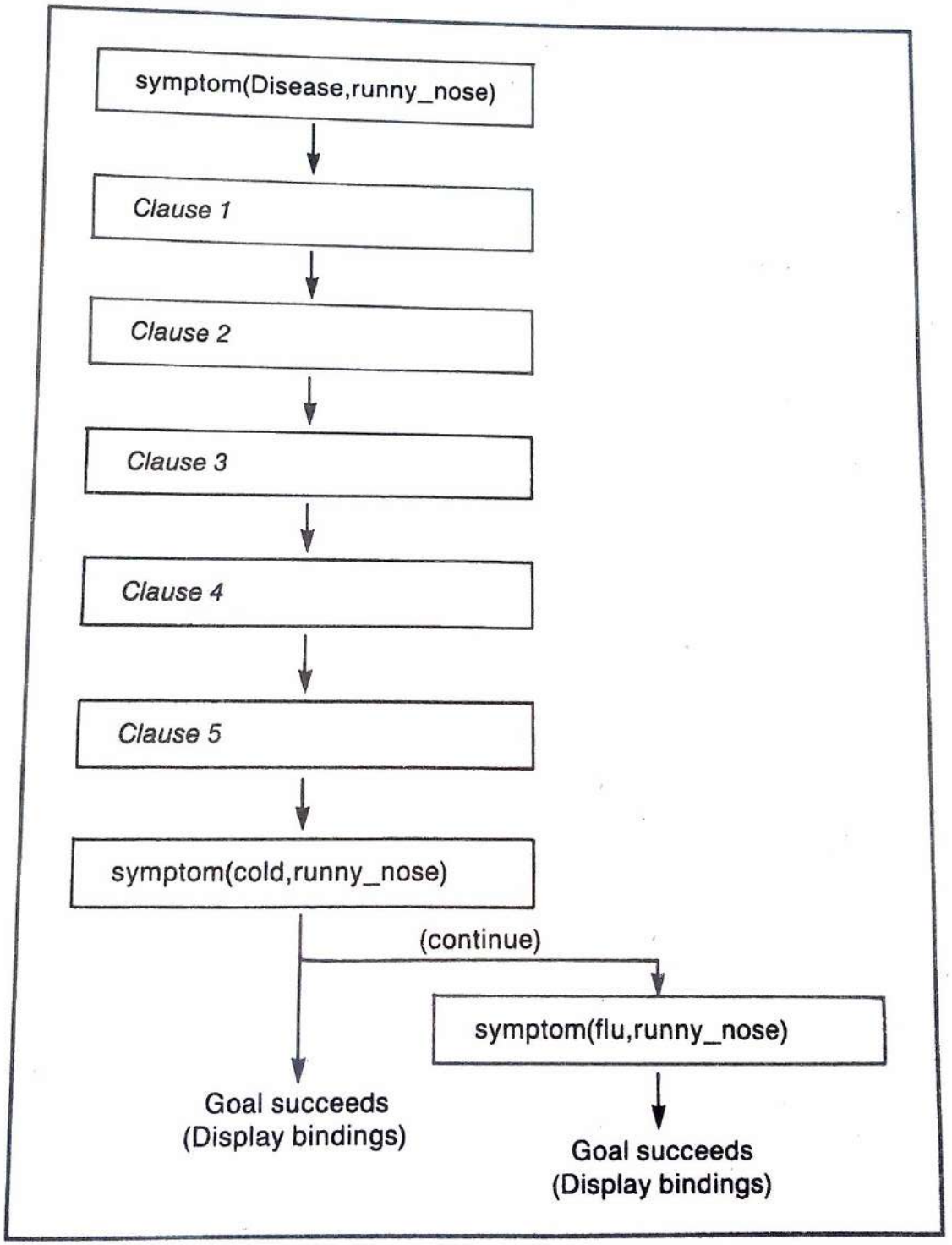


Figure 4.2: Execution flow of Prolog program

The match succeeds, and Prolog displays the values of the variables for the successful match. Execution does not stop, however. Turbo Prolog continues its search until it has tested all predicates for a match with the specified goal. In this case the goal can be solved in two ways by successively matching the variable *Disease* to *cold* and *flu*.

You could also use the same database to find all the symptoms for a particular disease. To do this, you specify the symptom as the variable

```
Goal : symptom(cold,Symptom)
Symptom = mild_body_ache
Symptom = runny_nose
Symptom = chills
3 Solutions
Goal :
```

You can also use variables in clauses, which will be very important later when you add rules to clauses.

BOUND AND FREE VARIABLES

If a variable has a value at a particular time, it is said to be *bound* or *instantiated*. If a variable does not have a value at a particular time, it is said to be *free* or *uninstantiated*. In the previous example, the program starts with a free variable, *Disease*. The goal succeeds with *Disease* bound to *cold*. Turbo Prolog, however, does not quit after the goal succeeds once. Once the goal succeeds and the variable bindings are displayed, Turbo Prolog frees all variables again and backtracks, trying to make the goal succeed again. In this example, Turbo Prolog finds another solution with *Disease* bound to *flu*. For a goal to succeed, all variables in the goal must become bound.

ANONYMOUS VARIABLES

Sometimes you may wish Prolog to ignore the value of one or more arguments when determining a goal's failure or success. To accomplish this, express the argument as an underline. Using the previous example, you could express this goal:

```
Goal : symptom(_,chills)
```

Prolog will respond as follows, indicating that the goal succeeds:

```
Goal : symptom(_,chills)
True
Goal :
```

Because there is no variable, there is no binding; if Prolog can match the relation name and the last argument, the goal succeeds. Prolog doesn't tell you which disease had the symptom *chills,* as it would have if you had used an ordinary variable. Your goal asked if there was any disease with that symptom, and Prolog answered yes.

You can also use an underscore in a clause to express the fact that the clause is true for all argument values. For example, consider this clause:

```
likes(jane,_).
```

This clause states that Jane likes everything. If the goal *likes(jane,computers)* were specified, it would succeed with this clause.

Note that in a clause, the anonymous variable stands for *all* values, while in a goal it is satisfied if at least one value corresponds to it.

COMPOUND GOALS

Suppose a patient comes into your imaginary doctor's office and complains of several symptoms. You want to express all of these symptoms in a compound goal to find the single disease that all of them relate to. For example, suppose the patient has a runny nose and a mild body ache. This could be expressed as this Prolog goal:

```
Goal : symptom(Disease,mild_body_ache) and
symptom(Disease,runny_nose)
```

Note: You can enter this goal even if it is too long for your Dialog window. Just keep typing without pressing ⟵, and the words will wrap to the next line as necessary.

Turbo Prolog responds

```
Goal : symptom(Disease,runny_nose) and
symptom(Disease,mild_body_ache)
```

```
Disease = cold
1 Solution
Goal :
```

This goal is compound; that is, all of the specified conditions must succeed for the goal to succeed. Prolog works from left to right in proving the compound goal, just as you read it. Once the first part of the goal is satisfied, the variable *Disease* is bound. If other parts of the goal contain this same variable, they must be satisfied with the same binding.

Now examine what happened this time in solving the goal. The search starts with *Disease* as a free variable. When the clause *symptom(cold, runny_nose)* is found, *Disease* becomes bound to *cold*. The first part of the goal is satisfied, and Prolog now tries to prove the second part of the goal. *Disease* is now bound, and Prolog marks this place in the list of clauses because it must return to it later.

As illustrated in Figure 4.3, Prolog next tries to prove the second part of the goal, searching again from the first clause that matches the second part of the goal. Since Disease is bound to *cold*, the second part of the goal must be satisfied with the same binding. A match is found with the clause *symptom cold,mild_body_ache)*. Since both parts of the compound goal succeed, the goal succeeds, and the solution is displayed.

The variable *Disease* becomes free again, and Turbo Prolog returns to the marked place, the clause where *Disease* was bound to *cold*, and goes on from there to try to find another match. This succeeds with the clause *symptom(flu,runny_nose)*. *Disease* becomes bound to *flu*, and Prolog returns to the second part of the compound goal and tries to solve it, searching for a match with *flu* as the value of *Disease*, which would be *symptom(flu,mild_body_ache)*. This will fail. Again *Disease* becomes a free variable, and the search continues from the last marked place, the clause where *Disease* was last bound (to *flu*). There are no more matches, and execution terminates.

BACKTRACKING

The solution in the previous example illustrates one of the most important principles of Prolog execution: *backtracking*. The solution of the compound goal proceeds from left to right. If any condition in the chain fails, Prolog backtracks to the previous condition, tries to prove it again with another variable binding, and then moves forward again to see if the failed condition will succeed with the new binding. Prolog moves relentlessly forward and backward through the conditions, trying every available binding in an attempt to get the goal to succeed in as many ways as possible.

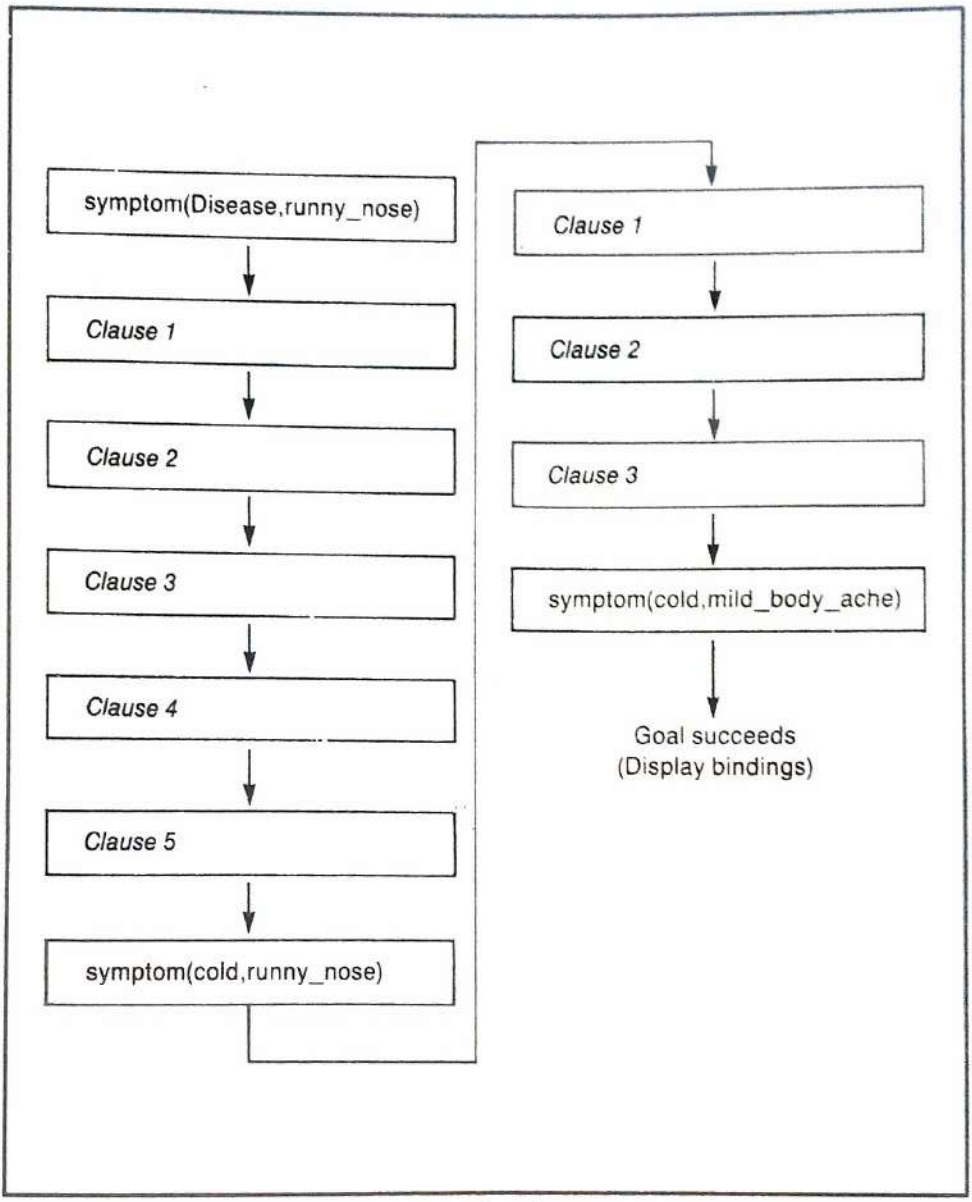


Figure 4.3: Execution with a compound goal

In other words, even though Prolog is an object-oriented language, the path Prolog takes in an attempt to prove a goal is very predictable. Prolog is considered object oriented in that the database (facts and rules about objects and their relationships) determines the path (procedure) used to prove the goal. You don't tell it how to proceed: it finds its own way.

Before continuing, experiment with the previous example using various compound goals (see Exercise 2). Be sure you understand backtracking as it applies to this simple example before trying the more complex examples beginning in the next chapter.

VARIABLE RULES

Four basic rules apply to Turbo Prolog variables:

1. All variable names must begin with an uppercase letter. The name may be from 1 to 250 characters and, except for the first character, contain lowercase or uppercase letters, digits, or underscores.
2. A variable is either bound or free at any given time.
3. Variable binding applies only to a specific clause. A variable of the same name in another clause is considered a different variable. In goals, including compound goals, the same binding applies wherever the same variable name is used.
4. Once a variable is bound to an object, it is typed the same as the object. For example, if the argument age is typed as an integer in the predicates or domains section and the variable *Employee_age* is bound to this argument in one of the clauses, *Employee_age* is then treated as an integer type. It will not match with an argument of a different type in another clause.

ADDING COMMENTS

You can add a comment at any point in a Prolog program. It is good practice to add comments liberally to help the user understand your program. You should always use comments to title and date your programs.

When adding a comment, begin the comment with /* and terminate it with */. You can add comments to lines of code or set them off on separate comment lines.

```
    /* MEDICAL DIAGNOSTIC SYSTEM      */
    /* January 4, 1987                */
    /* EXAMPLE ONLY                   */
    /* NOT FOR MEDICAL USE            */
domains                 /* Domains section */
        person1,person2 = symbol
predicates              /* Predicates section */
        likes(person1,person2)
clauses                 /* Clauses section */
    likes(mary,john).   /* sample clause */
```

EXERCISES

1. Suppose you have created this database:

```
location(portland,or)
location(washington,dc)
location(jackson,ms)
location(meridian,ms)
location(salem,or)
```

   a. Specify a goal to list all Oregon (or) cities.

   b. Specify a goal to find the state in which Jackson is located.

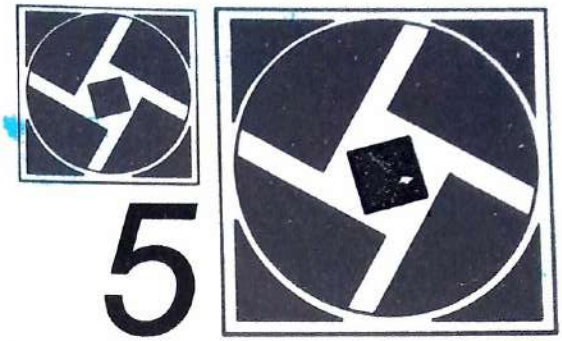
2. What is the result in each of the following cases using the example database in Figure 3.3?

   a. Goal : symptom(X,runny_nose)
      and symptom(X,mild_body_ache)

   b. Goal : symptom(X,runny_nose)
      and symptom(Y,mild_body_ache)

   c. Goal : symptom(X,runny_nose) and symptom(X,Y)

   d. Goal : symptom(cold,X) and symptom(flu,X)



5

You have already learned how you can use Prolog to simulate elementary medical diagnoses. With nothing more than a database of facts, you created a productive and useful medical diagnostic system. Now you will take your programming one step further by adding rules to the facts in the database. Rules permit Prolog to infer new facts from existing facts.

RULES

A *rule* is an expression that indicates that the truth of a particular fact depends upon one or more other facts. Consider this example:

IF there is a body stiffness or pain in the joints

AND there is a sensitivity to infections,

THEN there is probably a vitamin C deficiency.

This rule could be expressed as the following Turbo Prolog clause:

```
hypothesis(vitc_deficiency) if
    symptom(arthritis) and
    symptom(infection_sensitivity).
```

Notice the general form of the Prolog rule. The conclusion is stated first and is followed by the word *if*. The conditions upon which the conclusion

depends are stated next, each connected by the word *and*. The rule, like a fact, is terminated with a period. The conclusion can be viewed as a Prolog goal. This goal is true if all the conditions specified for the goal are true.

Every rule has a *conclusion* (or head) and an *antecedent* (or body). The antecedent consists of one or more *premises*. The premises in the antecedent form a conjunction of goals that must be satisfied for the conclusion to be true. If all the premises are true, the conclusion is true; if any premise fails, the conclusion fails.

A rule expresses a relationship between facts. Any given Prolog program is simply a database (collection of clauses) of facts and rules.

Notice that the conclusion and each condition are written on separate lines. This is done simply to make the rule more readable; it is not necessary for proper execution. The hypothesis and each condition are predicates and have a value of true or false.

Since rules are such an important part of any Prolog program, a shorthand has been developed for expressing rules. In this abbreviated form the previous rule becomes

```
hypothesis(vitc_deficiency) :-
    symptom(arthritis),
    symptom(infection_sensitivity).
```

The **:-** operator is called a *break*. A comma expresses an *and* relationship, and semicolon expresses an *or* relationship. However, if you wish to express an *or* relationship, it is generally clearer to use two rules:

```
hypothesis(vitc_deficiency) :-
    symptom(arthritis),
    symptom(infection_sensitivity).

hypothesis(vitc_deficiency) :-
    symptom(colitis),
    symptom(infection_sensitivity).
```

In English, these expressions state there is evidence of a vitamin C deficiency if there is either arthritis *and* infection sensitivity *or* colitis *and* infection sensitivity. If one or the other pair of premises is true, the conclusion is true.

USING RULES TO SOLVE PROBLEMS

The process of using rules and facts to solve a problem is called *formal reasoning*. Each of us uses formal reasoning each day to solve a variety of

problems: how to get a promotion at work, what is the best way to drive to a friend's house across town, how to start the car when turning the key only results in a dull, grinding sound on a cold morning.

A classic example of formal reasoning is found in the murder mystery, in which the detective must work through a maze of facts using formal reasoning to solve the mystery. The reader is challenged to try to use the same clues as the detective to solve the mystery first. Suspects with solid alibis are eliminated. Motives are examined to see which suspects have reasons to have committed the crime. How the person was killed may provide clues. Laboratory research often adds new factual information. Eventually the detective is able to use the facts to arrive at a conclusion. At each step, a rule expressing the relationship of facts leads the detective to a new goal that is closer to the final goal. For example,

IF the suspect was at Rosie's at 10:15 p.m.

AND the victim was killed at home at 10:30 p.m.

AND it takes 30 minutes to get from Rosie's to the victim's home,

THEN the suspect could not have been the killer.

You could use a Prolog program to sift through clues and solve a mystery.

VARIABLES IN RULES

You can use variables in rules to make a general statement about a relationship. For example, examine the following rule:

```
hypothesis(Patient,measles) :-
    symptom(Patient,fever),
    symptom(Patient,cough),
    symptom(Patient,conjunctivitis),
    symptom(Patient,runny_nose),
    symptom(Patient,rash).
```

This rule states that if *Patient* has a fever, cough, conjunctivitis, a runny nose, and a rash, then there is evidence that *Patient* has measles.

Now you will see how all of this fits together. Enter the short program in Figure 5.1 and compile it. This program can be used to simulate the diagnosis of childhood diseases.

A quick look at our database tells us that Charlie is sick. Charlie's symptoms (fever, rash, headache, and a runny nose) have been entered as facts in

```
/* MEDICAL DIAGNOSTIC SYSTEM  */
/* CHILDHOOD DISEASES         */
/* 1/4/87                     */
/* EXAMPLE ONLY               */
/* NOT FOR MEDICAL USE        */
domains
     disease,indication,name = symbol

predicates
     hypothesis(name,disease)
     symptom(name,indication)

clauses
     symptom(charlie,fever).

     symptom(charlie,rash).

     symptom(charlie,headache).

     symptom(charlie,runny_nose).

     hypothesis(Patient,measles) :-
          symptom(Patient,fever),
          symptom(Patient,cough),
          symptom(Patient,conjunctivitis),
          symptom(Patient,runny_nose),
          symptom(Patient,rash).

     hypothesis(Patient,german_measles) :-
          symptom(Patient,fever),
          symptom(Patient,headache),
          symptom(Patient,runny_nose),
          symptom(Patient,rash).

     hypothesis(Patient,flu) :-
          symptom(Patient,fever),
          symptom(Patient,headache),
          symptom(Patient,body ache),
```

Figure 5.1: A simulated medical diagnostic system for childhood diseases

```
        symptom(Patient,conjunctivitis),
        symptom(Patient,chills),
        symptom(Patient,sore_throat),
        symptom(Patient,cough),
        symptom(Patient,runny_nose).

hypothesis(Patient,common_cold) :-
        symptom(Patient,headache),
        symptom(Patient,sneezing),
        symptom(Patient,sore_throat),
        symptom(Patient,chills),
        symptom(Patient,runny_nose).

hypothesis(Patient,mumps) :-
        symptom(Patient,fever),
        symptom(Patient,swollen_glands).

hypothesis(Patient,chicken_pox) :-
        symptom(Patient,fever),
        symptom(Patient,rash),
        symptom(Patient,body_ache),
        symptom(Patient,chills).

hypothesis(Patient,whooping_cough) :-
        symptom(Patient,cough),
        symptom(Patient,sneezing),
        symptom(Patient,runny_nose).
```

Figure 5.1: A simulated medical diagnostic system for childhood diseases (cont.)

the database. There are also seven rules in the database that can be used to make inferences from these known facts. When the program prompts for a goal, enter the following:

```
Goal : hypothesis(Patient,Disease)
```

This asks Prolog to list all patients and their respective diseases. The program will then display

```
Goal : hypothesis(Patient,Disease)
Patient = charlie, Disease = german_measles
```

```
1 Solution
Goal :
```

Notice two things here:

- The variable name in the goal does not need to match the one in the clause; it is the position in the predicate that is important. Variable values are passed by the process of *unification*, discussed in detail later in this chapter.
- The argument names in the predicate section are simply placeholders. They are independent of any names in the clauses or goals, but they must by typed in the domains section.

Now look more closely at what happened when you specified this goal. Prolog moves immediately to the first rule whose head matches the goal or the first matching fact, whichever it encounters first:

```
hypothesis(Patient,measles)
```

Prolog marks this place and takes this conclusion as a goal and tries to prove it. Moving to the first premise, Prolog then tries to prove

```
symptom(Patient,fever)
```

Prolog starts from the head of the first rule or fact matching this predicate. This is the first clause in the program:

```
symptom(charlie,fever)
```

The match occurs, and *Patient* is bound to *charlie*. This value of the variable is passed to the original rule.

Prolog then moves to the next premise in the rule, holding the binding to *charlie* that has already occurred:

```
symptom(charlie,cough)
```

This match will fail, forcing the conclusion *hypothesis(charlie,measles)* to fail. Prolog then backtracks and tries to find another match on the original goal. The goal *hypothesis(Patient,Disease)* will unify next with

```
hypothesis(Patient,german_measles)
```

The variable *Patient* here is not the same as the variable of the first clause. There is no "variable value" passed to this clause from the first. In each clause, the variables in the clause can only by bound by the process of matching.

Again the first premise of the conclusion is tested, and *Patient* is again bound to *charlie*. The four premises of the German measles conclusion will all succeed with this binding, causing the original specified goal with *hypothesis(charlie,german_measles)* to succeed. The variables *Patient* and *Disease* in the original goal are now bound and displayed.

Prolog continues, trying to prove the specified goal again with different bindings. In this program, the original goal will match the head of each rule, so each rule will be tested before the program stops. In each case, at least one premise will fail. The goal will not succeed again.

PROLOG EXECUTION RULES

We can draw some conclusions from our discussion thus far:

- Prolog executes using a matching process.
- When the original goal is specified, Prolog tries to find a fact or the head (conclusion) of a rule that matches this goal.
- If a fact is found, the goal succeeds immediately.
- If a rule is found, Prolog then tries to prove the head of the rule by using the antecedent (the body of premises) as a new compound goal and proving each premise of the antecedent. If any premise of the antecedent fails, Prolog backtracks and tries to solve the preceding premises with other bindings. If Prolog is not successful, it tries to find another fact or rule that matches the original goal. If all premises succeed, the original goal succeeds.
- Turbo Prolog continues to execute until all possible solutions for the goal are tested.

Notice how variables are used. There are no "global" variables; that is, variables that maintain a value throughout the entire program's execution. All variables are local to the clause of which they are a part. Even if the same variable name is used in another clause, it is not the same variable. If a particular clause fails, Prolog backtracks and tries to solve the same goal another way using clauses.

In this example, the variable *Patient* is a unique variable in each rule. Values are passed between facts and rules based on the rules of unification (see the section "Unification" later in this chapter).

USING THE TRACE

You can use a Trace mode with your program to watch it execute a step at a time. Begin your program by adding the word *trace* anywhere before the domains section.

```
trace
domains
```

Now compile your program and specify as your goal

```
Goal : hypothesis(Patient,Disease)
```

The program will begin to execute and then pause with a menu at the bottom of the screen. You can then use the F10 key to execute the program one step at a time. Step through the entire execution, noticing how the program flow follows the description given in this chapter. Press Esc to terminate execution.

UNIFICATION

In Prolog, a *term* is a simple object, variable, or structure, such as a list or a compound structure (these will be introduced in later chapters). The process by which Prolog tries to match a term against the facts or the heads of other rules in an effort to prove a goal is called *unification*. Unification is a pattern-matching process.

A term is said to *unify* with another term if

- Both terms appear in predicates that have the same number of arguments (the same arity), and both terms appear in the same position in their predicates.
- Both terms appear as arguments of the same type—a symbol type can only unify with a symbol type and so on.
- All subterms unify with each other. (As you shall see later, a term can be a compound object or list. When it is, all subterms must unify.)

Here are the basic rules for unification:

- A variable that is free will unify with any term that satisfies the preceding conditions. After unification, the variable is bound to the value of the term.
- A constant can unify with itself or any free variable. If the constant is unified with a variable, the variable will be bound to the value of the constant.
- A free variable will unify with any other free variable. After unifying, the two variables will act as one. If one of the variables becomes bound, the other will be bound to the same value.

Predicates unify with each other if

- They have the same relation name.
- They have the same number of arguments.
- All argument pairs unify with each other.

For example, in our program, *hypothesis(Patient,Disease)* unified with *hypothesis(charlie,german_measles)*. In the process *Patient* was bound to *charlie*, and *Disease* was bound to *german_measles*.

Unification is similar to parameter passing in procedural programming. Values for one term are passed to another term, binding any variables in that term.

EXECUTION CONTROL

Although Prolog is not a procedural language, you can provide some control over its execution. Prolog follows these general rules:

- All clauses for the same predicate must be grouped together in the program. In the example here, all rules with the head *hypothesis(Person,Disease)* were grouped together. Any facts with the same predicate also have to appear in this grouping. If you fail to group clauses, Turbo Prolog will give you an error message. Predicate groups can be entered in any order.
- Within a specific predicate group, Prolog begins testing for a match (unification) at the first fact or rule head that matches the specified goal. Subsequent clauses of the same predicate that also unify are tested in the order in which they appear in the program.

- If the head of a rule unifies with the goal, the antecedent becomes a new subgoal and must be proven next. Subgoals must be satisfied from left to right.

Execution, then, is from top to bottom and from left to right. Prolog uses depth-first testing; that is, the testing of any particular rule will proceed as far as it can until all subgoals fail, and then the next rule is tested to see if it unifies with the specified goal.

If you reorder the clauses in the medical diagnostic system, you will find that the program still works so long as you keep all the *symptom(Person,Indication)* predicates together and all the *hypothesis(Person,Disease)* predicates together. The efficiency of the search, however, is affected by the order of the clauses. If your program contained tests for 300 diseases, you would want the rules for the least likely diseases at the end of the program. If the rule for the common cold was at the end of the program, then each time you tested for the common cold, the program would first have to test for all the other (and perhaps rare) diseases. In long and complex programs, this can take some time. In most applications the facts about the patient are not entered as facts (as in this example), but through a dialog session. If the clauses are ordered inefficiently, the user will have to respond to many questions that have little to do with the problem at hand.

The definition of the predicates and the order of the clauses impose a heuristic on the program. This heuristic defines the way Prolog works through the problem space to the eventual solution. Most Prolog program development time is spent trying to define the best heuristic possible for a given problem space.

Beginning in Chapter 7, you will find examples in which techniques for controlling execution are used, and as a result, the order of the clauses controls more than the efficiency of the program. In many cases, reordering the clauses will cause the program to fail. The order of the clauses often controls more than the heuristic; it can define the procedure that is used.

BUILT-IN PREDICATES

Most versions of Prolog contain a variety of *built-in*, or *standard*, predicates that can support a variety of functions, such as control and data input and output. Turbo Prolog is no exception. It offers dozens of built-in predicates to support not only input and output operations, but graphics, file operations, string handling, and type conversion. These predicates can be used in rules or facts, just like any other predicate. They do not need to be defined in the predicates section, as they are integral to Turbo Prolog. The

Turbo Prolog built-in predicates are classified into nine groups:

- Control predicates—For controlling program execution, forcing or preventing backtracking.
- Reading predicates—For reading data from the keyboard or file to a variable.
- Writing predicates—For writing data to the screen, printer, or a file.
- File system predicates—For managing disk files from a Turbo Prolog program.
- Screen-handling predicates—For graphic control of the display (windows, turtle graphics, and so on) and sound control.
- String-handling predicates—For various operations on string data.
- Type-conversion predicates—For converting data types from one form to another.
- System-level predicates—For access to DOS functions from within a Turbo Prolog program.

These various groups will be discussed in later chapters.

THE not PREDICATE

You may sometimes want to express explicitly in the data base that a particular fact is not true. To do this, you must use built-in *not* predicate. The *not* predicate cannot be used to express a fact or appear in the head of a rule. It can only be used in a premise, as in

```
replace(right_speaker):-
    not(is(right_speaker,functional)).
```

In this case, if

```
is(right_speaker,functional).
```

is in the database, the rule will fail.

Be careful in your program design to distinguish between the use of the *not* predicate and the omission of facts from the database. If a fact is not in the database, Prolog considers it false. However, this falsity based on absence from the database is not sufficient to prove a *not* premise. The absence of the *is* fact above would not prove the *not* premise in the rule.

EXERCISES

1. What is the head and what is the body of each rule?

   a.
   ```
   hypothesis(meningitis) :-
       symptom(fever),
       symptom(headache),
       symptom(skin_spots).
   ```

   b.
   ```
   hypothesis(rain) :-
       barometer(falling),
       wind(east).
   ```

2. The following database exists:

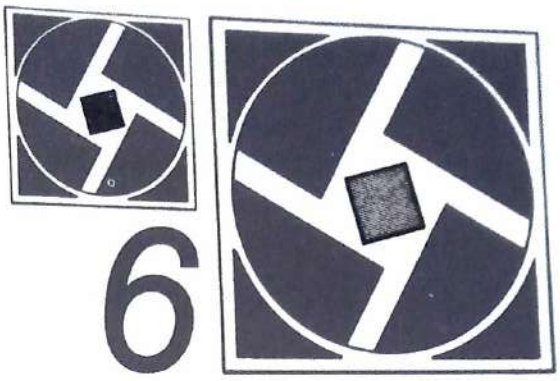
```
chkstate("NC").
chkstate(_).
chkstate(State) :-
    location(State).
```

Which clause will each of the following goals unify with first?

a. chkstate(State)

b. chkstate("OR")

c. chkstate(_)



A nontechnical user would have difficulty using the medical diagnostic system in Chapter 5. You could add more rules and symptoms, but a larger problem exists. The user interface needs to be changed so that

- The user can easily define the goal. For example, the user should have to enter only a single word, such as *go* or *run*, to start the program. From that point on program execution proceeds automatically.
- Specific symptoms are entered by the user in response to prompts at the time of program execution; they are not written into the program. For example, the program could ask whether Charlie has a fever and then set *symptom(charlie,fever)* to true or false based on the response.
- Results are communicated to the user in English-like sentences, such as

  Charlie probably has German measles.

- Underlines between words are omitted, true spaces are used, and names are capitalized. (These changes require nothing more than using string variables when character string values are output or entered.)

This chapter will show you how to add this kind of friendlier user interface to your medical diagnostic program.

THE CONSULTATION PARADIGM

Often, you will want a Prolog program to enter into a dialog with the user—to ask questions, issue responses, and perhaps tell the user how it reached a specific conclusion. Some examples of this were given in Chapter 1.

Your aim might be to allow the following type of dialog:

```
What is the patient's name?
Charlie
What is the patient's age?
10
Does Charlie have a fever?
yes
Does Charlie have a cough?
no
Does Charlie have a headache?
yes
Does Charlie have a runny nose?
yes
Does Charlie have a rash?
yes
Charlie probably has German measles. Is the rash rose colored?
yes
Is the fever only slight?
yes
Did the rash start on the face and is it now on the body?
yes
Are the joints stiff?
yes
Are the lymph nodes behind the ear enlarged?
yes
Charlie has German measles.
```

In expert systems design, this is called a *consultation paradigm*. The general structure works with almost any type of diagnostic system. Notice two elements of this structure:

- Most of the questions require yes or no answers, although some assign values, such as the patient's age, to input variables

(bind the variables) that will be used later in the program. You can also create multichoice questions in Prolog (see Chapter 12).

- The program does not ask questions about some of the symptoms (such as chills or conjunctivitis) that are included in some of the rules. Some type of heuristic quickly narrowed Prolog's search so Prolog could skip irrelevant questions.

To add input and output features to any Prolog program, you use the built-in predicates that support these features.

OUTPUT PREDICATES

Turbo Prolog has three output predicates: write, writedevice, and writef. Let us see how these predicates work. (You can also write your own output predicates using the built-in *write* and *writef* predicates for special types of output, such as the items in a list. User-defined output predicates are illustrated in Chapter 13.)

The write Predicate

As a starting point, modify the program you developed in Chapter 5 by adding a single new clause to the beginning using the *write* built-in predicate.

```
go :-
    hypothesis(Patient,Disease),
    write(Patient," probably has ",Disease,"."),nl.
```

Compile your new program and execute it.

Now when you start Turbo Prolog and it prompts you for a goal, you only need enter the word *go*. *Go* will unify with the head (conclusion) of your new rule, and Prolog will begin trying to prove *hypothesis(Patient,Disease)*. This goal will succeed, with *Patient* bound to the value of *Charlie* and *Disease* to the value *german_measles*. The write predicate displays the value of these variables in an English-like sentence:

```
charlie probably has german_measles.
```

You can improve the appearance of this sentence by defining the *name* and *disease* objects as string objects. This will enable Prolog to output

```
Charlie probably has German Measles.
```

The new listing (also showing some other changes you will make shortly) is shown in Figure 6.1 at the end of this chapter. Don't enter the new version yet. We will build up to it in the course of the chapter.

For the moment ignore the *nl* after the write predicate. You will see its role in the next section. Now look at the write predicate itself.

The general form of the write predicate is

```
write(E1,E2,E3,...,En)
```

where *E1, E2, E3,..., En* represent Prolog variables or objects of the standard domain types. Any variable in this list *must* be bound before the write predicate is invoked. The write predicate executes immediately and always evaluates as true. You can mix objects and variables in the arguments.

Execution proceeds from left to right and from top to bottom. For example, the clause.

```
test :-
    write("This is an example "),
    write("of multiple write statements.").
```

displays

```
This is an example of multiple write statements.
```

Control

Now return to the *nl* after the write predicate. This is a built-in predicate indicating the *newline* function. This function writes a carriage return and a line feed in the output. For example,

```
write("Patient," probably has ",Disease,".").nl,
write("Is the rash rose colored (y/n) ?").
```

displays

```
Charlie probably has german_measles.
Is the rash rose colored (y/n) ?
```

Another method of indicating a new line is to use the write predicate with a *backslash command*—in this case, \n:

```
write(      "Patient," probably has ",Disease,".\n"),
write("Is the rash rose colored (y/n) ?").
```

Turbo Prolog provides three backslash commands:

\n	Carriage return/line feed
\t	Tab
\b	Backspace (overtype)

Backslash commands must appear inside of either single or double quotation marks. They also can be part of another character string that is being output, as shown in the preceding example.

To output a backslash as a character on a display or printer, enter a double backslash:

```
\\
```

You now have three methods of creating a carriage return/line feed combination:

```
nl
write("\n")
write('\n')
```

All of these work equally well.

Using the Printer

Any output to the display can be directed instead to a printer or a file. To redirect output, use the *writedevice* built-in predicate. The following code directs the output of the write predicate to the printer and then redirects further output back to the screen.

```
writedevice(printer),
write("This will print on the printer."),
writedevice(screen).
```

The writef Predicate

Sometimes you may wish to format output. For example, you may want the decimal points in a table to be aligned. With Turbo Prolog, you can use the *writef* built-in predicate to force alignment of numbers or text. The standard form of the writef predicate is

```
writef(format,E1,E2,E3,...,En)
```

The variable *format* is a control code of the type

```
%-m.p
```

where elements of the code are as follows:

-	Optional hyphen forcing left justification; default is right justification.
m	Optional parameter defining minimum field width.
p	Optional parameter defining the precision of a decimal floating-point number or the maximum string length.

Note that a period is used to separate the *m* and *p* parameters.

The *p* parameter can also contain an optional suffix to indicate the numeric format type:

f	Fixed point (default)
e	Exponential
g	General (whatever format is the shortest)

For example,

```
writef("%-10 #%5.0 $%3.2 \n",fan,23,3.1)
```

displays

```
fan # 23 $3.10
```

In this example, the first argument is not an output string, but a format control string. It can be read as

1. Start the line with a left-justified field of 10 characters. This is used to display the symbolic object *fan*.
2. Skip one space and then display #.
3. Display the integer *23* right-justified in a field of five characters with no decimal places.
4. Skip one space and then display a dollar sign.
5. Display the real-number value *3* right-justified in a field of three characters with one decimal place.
6. Issue a carriage return and line feed.

INPUT PREDICATES

The Turbo Prolog manual uses the word *domain* to refer to the type of an object. In the science of expert systems, however, the word has another meaning. In expert system design, a domain is a definable body of knowledge about a specified subject matter. In the example system, the dc ain is diagnoses of eight childhood diseases. You have limited the age of the patients and the number of diseases you will try to diagnose.

You can see from the previous chapter that you use two kinds of knowledge in creating a Prolog program for a particular domain. The first is general knowledge that can be applied to any problem in the domain. An example of this is the rules in the medical diagnostic system. This is a *static* database; it remains virtually unchanged each time the program is executed. The second is a *dynamic* database. This consists of knowledge about a particular problem. In the medical diagnostic system, the dynamic database consists of the facts that apply to Charlie's specific symptoms.

In your program's current design, the program would have to be rewritten and recompiled each time the problem (the dynamic database) changes—a new program for each patient. This is inconvenient and time consuming, particularly if the user needs answers quickly. It would be much better if just the static database (rules) were written into the program and knowledge for the dynamic database (patient and symptoms) could be obtained from consultation with the user each time the program runs.

You have already seen how you can output questions and results. Now you will see how to get input from the user.

Turbo Prolog provides several built-in input predicates for this purpose: readln, readchar, readint, readreal, inkey, and keypressed. You will find that these input predicates will work for most input operations. The first four function in basically the same way. Let us look at these four first.

The choice of which of the four read predicates to use is determined by the type of variable used for the input, as follows:

readln	String or symbol
readchar	Character
readint	Integer
readreal	Real

The readln Predicate

The *readln* predicate permits a user to read any string or symbol into a variable. For example, you can redefine the *symptom(charlie,fever)*

predicate clause in the Figure 5.1 example as follows:

```
symptom(Patient,fever) :-
    write("Does the  ", Patient,"  have a fever (yes/no)? "),
    readln(Reply),
    Reply = "yes".
```

When this rule is invoked, it displays the question and then pauses for an answer. The answer must be terminated with a carriage return. The rule will succeed if the user enters

```
yes ↵
```

and fail if the user enters anything else. Characters entered on the keyboard are displayed on the screen as they are entered.

Before continuing, edit the first symptom predicate clause of the Figure 5.1 medical diagnostic program using readln. Add the two *go* clauses as shown in Figure 6.1 for the entry of the patient's name. Then compile and execute your new program.

The readchar Predicate

You can use the *readchar* built-in predicate in the same way as readln:

```
symptom(Patient,fever) :-
    write("Does  ",Patient,"  have a fever (y/n)? "),
    readchar(Reply),
    write(Reply),nl,
    Reply = 'y'.
```

In this case, *Reply* is a character-typed variable. (Recall that character type arguments are enclosed in single rather than double quotation marks.) With readchar, there is no echo on the screen during input; instead, the write predicate and newline character must be used to display what has been entered. With readchar, the user enters only a single character and does not need to enter a carriage return with the input.

When you use the readchar predicate, you may wish to define your own input predicate to create the echo and reduce the programming in other clauses. For example,

```
symptom(Patient,fever) :-
    write("Does  ",Patient,"  have a fever (y/n)? "),
```

```
    response(Reply),
    Reply = 'y'.
```

If you do this for each symptom, Prolog will try to prove the *response(Reply)* premise, which is not a built-in predicate. In testing the premise, it will find a rule that you add at the end of the program:

```
response(Reply) :-
    readchar(Reply),
    write(Reply),nl.
```

This rule reads the user response, writes it to the screen, and unifies it with *Reply*. Then Prolog goes back to the *symptom* predicate rule and tests the last premise to see if the value bound to *Reply* is equal to *'y'*. If it is, the conclusion is proved: Fever is one of the symptoms.

The readint Predicate

The *readint* predicate can be used to read an integer value to a variable. For example,

```
chkage(Patient) :-
    write("What is ",Patient," 's age?"),
    readint(Age),
    Age > = 12,
    write(Patient," cannot be evaluated with"),nl,
    write("this system. The system is designed"),
    write("for childhood diseases only.").
```

In this case the system asks a question, "What is Patient's age?" and then pauses for an answer, the entry of the age. The next premise compares this answer to a specified value. In this example, the age entered is compared to 12; the test fails if the age is less than 12 (this type of arithmetic operation is discussed further in Chapter 10). If the test succeeds, Prolog goes on to the rest of the premises so that a message is displayed saying that the system is for childhood diseases only.

The readreal Predicate

The *readreal* predicate can be used to read floating-point numbers into a variable. For example, an inventory control system might include the

following code:

```
askprice(Item,Price) :-
    write("What is the price of ",Item," ?"),
    readreal(Price).
```

This program displays a question asking for entry of an item's price. The user then enters the price, and the value is bound to *Price*.

The inkey and keypressed Predicates

The *inkey* predicate reads a single character from the input. The predicate form is

```
inkey(Char)
```

If there is no input, the predicate fails. If there is input, the character is returned bound to *Char*.

You also can use the *keypressed* predicate to determine whether a key has been pressed without a character being returned. This predicate form is

```
keypressed
```

This predicate fails if no key has been pressed and succeeds if the user has pressed any key.

USING WINDOWS

Turbo Prolog permits you to create and use windows while executing. This allows you to develop excellent user interfaces with minimal programming. Windows will be discussed more fully later in this book, but one built-in predicate is easy to use and could be a part of your program at this point. To clear the dialog window, use the *clearwindow* predicate. The modified version of the *go* clause is

```
go :-
    clearwindow,
    write("What is the patient's name? "),
    readln(Patient),
    hypothesis(Patient,Disease),
    write(Patient," probably has ",Disease),nl.
```

If you have made all of the changes to the medical diagnostic program discussed in this chapter, your program should now look like Figure 6.1.

```
/* MEDICAL DIAGNOSTIC SYSTEM    */
/* CHILDHOOD DISEASES           */
/* EXAMPLE ONLY                 */
/* NOT FOR MEDICAL USE          */
/* Modified for Input & Output */
/* 1/4/87                       */

domains
     disease, indication = symbol
     patient = string

predicates
     hypothesis(patient,disease)
     symptom(name,indication)
     response(char)
     go

clauses
     go :-
          write("What is the patient's name? "),
          readln(Patient),
          hypothesis(Patient,Disease),
          write(Patient," probably has ",Disease,"."),nl.

     go :-
          write("Sorry, I don't seem to be able to"),nl,
          write("diagnose the disease."),nl.

     symptom(Patient,fever) :-
          write("Does ",Patient," have a fever (y/n) ?"),
          response(Reply),
          Reply='y'.

     symptom(Patient,rash) :-
          write("Does ",Patient," have a rash (y/n) ?"),
          response(Reply),
          Reply='y'.

     symptom(Patient,headache) :-
          write("Does ",Patient," have a headache (y/n) ?"),
          response(Reply),
          Reply='y'.

     symptom(Patient,runny_nose) :-
          write("Does ",Patient," have a runny nose (y/n) ?"),
          response(Reply),
          Reply='y'.

     symptom(Patient,conjunctivitis) :-
          write("Does ",Patient," have conjunctivitis (y/n) ?"),
          response(Reply),
          Reply='y'.
```

Figure 6.1: The revised medical diagnosis program

```
symptom(Patient,cough) :-
     write("Does ",Patient," have a cough (y/n) ?"),
     response(Reply),
     Reply='y'.

symptom(Patient,body_ache) :-
     write("Does ",Patient," have a body ache (y/n) ?"),
     response(Reply),
     Reply='y'.

symptom(Patient,chills) :-
     write("Does ",Patient," have chills (y/n) ?"),
     response(Reply),
     Reply='y'.

symptom(Patient,sore_throat) :-
     write("Does ",Patient," have a sore throat (y/n) ?"),
     response(Reply),
     Reply='y'.

symptom(Patient,sneezing) :-
     write("Is ",Patient," sneezing (y/n) ?"),
     response(Reply),
     Reply='y'.

symptom(Patient,swollen_glands) :-
     write("Does ",Patient," have swollen glands (y/n) ?"),
     response(Reply),
     Reply='y'.

hypothesis(Patient,measles) :-
     symptom(Patient,fever),
     symptom(Patient,cough),
     symptom(Patient,conjunctivitis),
     symptom(Patient,runny_nose),
     symptom(Patient,rash).

hypothesis(Patient,german_measles) :-
     symptom(Patient,fever),
     symptom(Patient,headache),
     symptom(Patient,runny_nose),
     symptom(Patient,rash).

hypothesis(Patient,flu) :-
     symptom(Patient,fever),
     symptom(Patient,headache),
     symptom(Patient,body_ache),
     symptom(Patient,conjunctivitis),
     symptom(Patient,chills),
     symptom(Patient,sore_throat),
     symptom(Patient,cough),
     symptom(Patient,runny_nose).

hypothesis(Patient,common_cold) :-
     symptom(Patient,headache),
     symptom(Patient,sneezing),
```

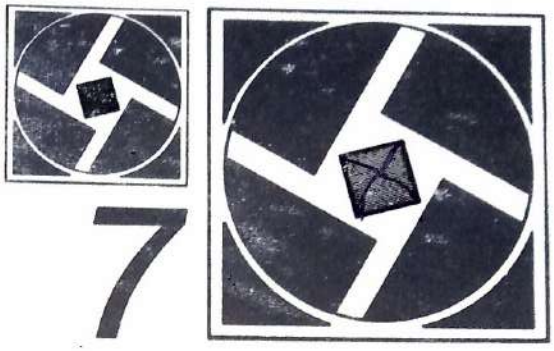
Figure 6.1: The revised medical diagnosis program (cont.)

```
        symptom(Patient,sore_throat),
        symptom(Patient,chills),
        symptom(Patient,runny_nose).

hypothesis(Patient,mumps) :-
        symptom(Patient,fever),
        symptom(Patient,swollen_glands).

hypothesis(Patient,chicken_pox) :-
        symptom(Patient,fever),
        symptom(Patient,rash),
        symptom(Patient,body_ache),
        symptom(Patient,chills).

hypothesis(Patient,whooping_cough) :-
        symptom(Patient,cough),
        symptom(Patient,sneezing),
        symptom(Patient,runny_nose).

response(Reply) :-
        readchar(Reply),
        write(Reply),nl.
```

Figure 6.1: The revised medical diagnosis program (cont.)

EXERCISES

1. Enter the program in Figure 6.1 and experiment with it. This program has a serious problem. What is it and what causes it?



7

This chapter is the first of three chapters on techniques for controlling the execution of a Turbo Prolog program. This chapter describes the use of the fail predicate, Chapter 8 introduces the concept of recursion, and Chapter 9 introduces the cut.

THE fail PREDICATE

In Prolog, forcing a rule to fail under certain conditions is a type of control and is essential to good programming.

Failure can be forced in any rule by using the built-in *fail* predicate. The fail forces backtracking in an attempt to unify with another clause. Whenever this predicate is invoked, the goal being proved immediately fails, and backtracking is initiated. The predicate has no arguments, so failing at the fail predicate is not dependent on variable binding; the predicate always fails.

Why is this type of predicate useful? A few examples will illustrate its value.

Here is a very simple example illustrating the basic principle of the fail predicate:

```
go :-
    test,
    write("You will never get here.").
```

```
test :-
    fail.
```

If you specify the goal as *go*, Prolog will unify with the head of the first rule and then try to prove its premises. The *test* premise will unify with the head of the second rule, whose premise is *fail*, and the goal will fail. Prolog will backtrack to the first rule, and the *go* goal will fail again. The write predicate will never be executed. The following will be displayed:

```
Goal : go
False
Goal :
```

Try the same program again, this time reversing the two antecedents (premises) and changing the text string slightly:

```
go :-
    write("You will get here."),
    test.

test :-
    fail.
```

Again the goal will fail, but the text string will be displayed this time:

```
Goal : go
    You will get here.
    False
    Goal :
```

USING THE fail PREDICATE

Figure 7.1 shows a sample program for printing a list of names. Notice that it contains no domains section. The domains of the two arguments are declared as string types in the predicates section. Notice where the fail predicate is placed in the program:

```
go :-
    location(City,State),
    writef("%-10 %2 \n",City,State),
    fail
```

```
/* EXAMPLE WITH THE FAIL PREDICATE */

predicates
     location(string,string)
     go

clauses
     go :-
          location(City,State),
          writef("%-10  %2  \n",City,State),
          fail.

     go.

     location("Jackson","MS").
     location("Washington","DC").
     location("Raleigh","NC").
```

Figure 7.1: The list program with the fail predicate

Enter this program and compile it. Execute the program and you should see the following output:

```
Goal : go ↵
Jackson      MS
Washington   DC
Raleigh      NC
```

The entire list is displayed, and the goal succeeds.

When the goal *go* is specified, it unifies with the head of the first rule, and the testing begins for that rule. The *location(City,State)* predicate forces *City* to bind with *"Jackson"* and *State* to bind with *"MS"*. Prolog marks the place in the clauses in case it needs to backtrack to it.

The variable names are then displayed. Because of the formatting specified by the writef predicate, the city is left-justified in a field of 10 characters. Notice that a newline character also is executed, forcing a carriage return and linefeed.

The testing of the rule then fails because of the fail predicate, forcing Prolog to backtrack. The writef predicate is always true, so Prolog backtracks further.

At the *location(City,State)* predicate, the variables are free again. Prolog returns to its marked place and tries to find another match, succeeding with *City* bound to *"Washington"* and *State* bound to *"DC"*. Prolog reverses, moving forward again in an attempt to prove the goal *go*. The writef

predicate succeeds, outputting the new variable values, carriage return, and line feed. Again the goal fails at the fail predicate.

Backtracking the second time, Prolog then finds the third match and displays the values for this last match. Again the fail predicate forces backtracking, but Prolog finds no more *location(City,State)* predicates. The clause fails, and Prolog backtracks to the second *go* clause. This succeeds, and the goal succeeds.

Notice the general program structure when the fail predicate is used. The rule with the fail predicate fails every time. A terminating clause that always succeeds must be added to make the whole definition function properly. The extra clause is called the *terminating condition*. The first rule (or rules) in the definition does all the work and includes a *fail* predicate to force backtracking. The final fact or rule in the definition always succeeds and terminates the backtracking.

The list program in this example illustrates several key points about the use of the fail predicate:

- The order of the *go* clauses is very important. Their order defines an algorithm or procedure. If the order of the two clauses is reversed, the goal succeeds, but no list is printed.
- The second *go* clause (the simple fact) is an important part of the definition. It terminates the backtracking and is called the *terminating condition*.
- The variables in the clause lose their binding every time the rule fails. The backtracking forces a new binding.
- Prolog remembers the bindings it has already tried. It marks the place of each binding, so it can return and unify with the next clause in sequence—in this case, the sequence of *location* clauses.
- Prolog proceeds from left to right, going as far as it can on one path before backtracking and trying another. So, in the example, it exhausts all possibilities of proving the first *go* clause, before moving on to the second one.

In some cases, two or more rules, all ending with a fail predicate, may have the same head. The predicate grouping will conclude with a final fact or a rule without a fail predicate. After the first rule exhausts all possible solutions, the next rule with the same head begins execution and continues until it exhausts all possible solutions. This process continues until Prolog reaches the final clause, a fact, or a rule without a fail predicate. You can use this technique to add a header to the city/state table listing of the previous example (see Figure 7.2).

```
/* EXAMPLE WITH THE FAIL PREDICATE USED TWICE */

predicates
     location(string,string)
     go

clauses
     go :-
          writef("%-10  %5 \n","CITY","STATE"),
          fail.

     go :-
          location(City,State),
          writef("%-10  %2 \n",City,State),
          fail.

     go.

     location("Jackson","MS").
     location("Washington","DC").
     location("Raleigh","NC").
```

Figure 7.2: Adding a header to the list program

EXCLUSION USING THE fail PREDICATE

Sometimes you may wish to exclude from a database all objects that meet a specified criteria. For example, suppose you want to list all city/state combinations in the database except those from DC. One way to do this is to use the fail predicate to skip these particular items.

An attempt to use the fail predicate for this purpose is shown in Figure 7.3. In this example a new premise that tests the state is added to the first *go* rule. When *location(City,State)* has *State* bound to *DC*, *chkstate(State)* will fail, forcing backtracking. To enable tests for other states to succeed, a second clause must be added. However, when *chkstate("DC")* fails, Prolog backtracks to *chkstate(State)*, which tests the second *chkstate* clause, which succeeds. Prolog then writes *Washington, D.C.*, which is not what you want it to do. To make this structure work, you need to use the *cut* predicate, discussed in Chapter 9. The cut and fail combination is a very powerful control structure in Prolog programs, as you will see.

```
/* EXCLUSION EXAMPLE */

predicates
     location(string,string)
     go
     chkstate(string)

clauses
     go :-
          writef("%-10  %5 \n","CITY","STATE"),
          fail.

     go :-
          location(City,State),
          chkstate(State),
          writef("%-10  %2 \n",City,State),
          fail.

     go.

     location("Jackson","MS").
     location("Washington","DC").
     location("Raleigh","NC").

     chkstate("DC") :-
          fail.

     chkstate(_).
```

Figure 7.3: Exclusion using fail (program does not work)

EXERCISES

1. Using the fail predicate, write a short program that lists four addresses in a label form. Each address should list a name, one-line address, city, state, and ZIP code.

2. Rewrite the example in Figure 7.3 so that it will work correctly using the not predicate instead of the fail predicate.

3. Why does the program in Figure 7.1 fail if the clauses are reversed?