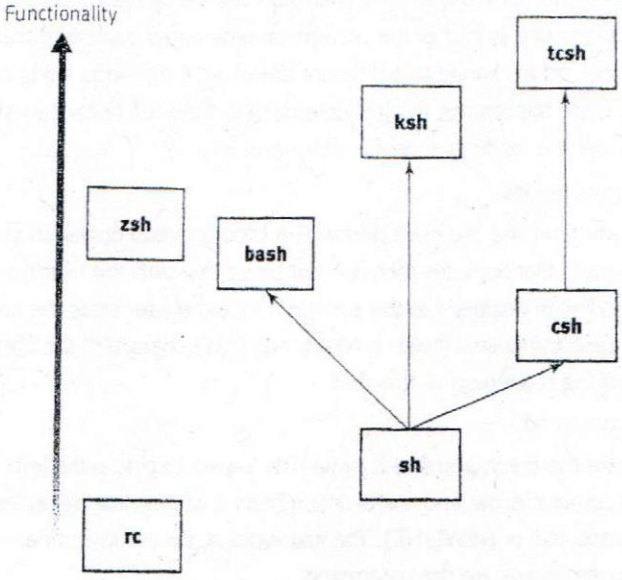

Figure 4.1 Shell families and their relative functionalities

the top of the hierarchy is the Korn shell (ksh), which includes all the functionality of the Bourne shell and much more. The rc and zsh shells are outliers that cannot be readily associated with any of the primary shell families.

4.2.2 Which Shell Suits Your Needs?

Most shells perform some similar functions, and knowing the details of how they do so is important in deciding which shell to use for a particular task. Also, using more than one shell during a session is a common practice, especially among shell programmers. For example, you might use the Bourne or Korn shell for their programming capabilities and use the C shell to execute individual commands. We discuss this example further in Section 4.2.3. The similarities of major shell functions are summarized in Table 4.2.

TABLE 4.1 Shell Locations and Program Names

Shell	Location on System	Program (Command) Name
rc	/usr/bin/rc	rc
Bourne shell	/usr/bin/sh	sh
C shell	/usr/bin/csh	csh
Bourne Again shell	/bin/bash	bash
Z shell	/usr/local/bin/zsh	zsh
Korn shell	/usr/bin/ksh	ksh
TC shell	/usr/bin/tcsh	tcsh

Consider the following steps that a shell must take to accomplish its job.

1. Print a prompt.

   A default prompt string is available, sometimes hardcoded into the shell, for example the single character string %, #, or >. When the shell is started, it can look up the name of the machine on which it is running and prepend this string to the standard prompt character, for example a prompt string such as kiowa>. The shell also can be designed to print the current directory as part of the prompt, meaning that each time that the user types cd to change to a different directory, the prompt string is redefined. Once the prompt string is determined, the shell prints it to stdout whenever it is ready to accept a command line.

2. Get the command line.

   To get a command line, the shell performs a blocking read operation so that the process that executes the shell will be asleep until the user types a command line in response to the prompt. Once the user types the command line (and terminates it with a NEWLINE ('\n') character), the command line string is returned to the shell.

3. Parse the command.

   The syntax for the command line is trivial. The parser begins at the left side of the command line and scans until it sees a whitespace character (such as space, tab, or NEWLINE). The first word is the command name, and subsequent words are the parameters.

4. Find the file.

   The shell provides a set of *environment variables* for each user. These variables are first defined in the user's **.login** file, though they can be modified at any time by using the **set** command. The PATH environment variable is an ordered list of absolute pathnames specifying where the shell should search for command files. If the **.login** file has a line such as

   ```
   set path=(. /bin /usr/bin)
   ```

   then the shell will first look in the current directory (since the first full pathname is "." for the current directory), then in **/bin**, and finally in **/usr/bin**. If no file with the same name as the command can be found (from the command line) in any of the specified directories, then the shell notifies the user that it is unable to find the command.

5. Prepare the parameters.

   The shell simply passes the parameters to the command as the argv array of pointers to strings.

6. Execute the command.