CHAPTER 1

REGULAR EXPRESSIONS AND METACHARACTERS

Regular expressions are like opinions: Everyone has them, but not everyone uses them wisely.

CHAPTER OBJECTIVES

In this chapter, you will learn about:

If you have not encountered regular expressions before, this chapter will help you gain an understanding of how to use and become effective in writing regular expressions. If you have encountered and used regular expressions within the UNIX shell, differences exist between regular expressions used within the shell and regular expressions used within grep, awk, and sed. This chapter does not explain those differences. Rather, we discuss and understand regular expressions as used within grep, sed, and awk. We do explain differences among regular expressions used by grep and sed, and regular expressions used by awk and egrep whenever possible. If you already feel comfortable with regular expressions as used in grep, awk, and sed, you still might want to work out the problems and exercises as a review.

LAB 1.1

USING THE PERIOD AND BACKSLASH METACHARACTERS

LAB OBJECTIVES

After this Lab, you will be able to:

- ✓ Identify the Operands and Operators of a Regular Expression
- ✓ Understand the Results of a Regular Expression Consisting of Operands and Operators
- ✓ Identify When a Regular Expression Is Evaluated
- ✓ Understand the Results of Evaluating the Expression as a Pattern
- ✓ Understand the Wildcard Metacharacter

This Lab briefly describes what a regular expression is and lists all the metacharacters used within this book. The best way to learn regular expressions is to learn the functions performed by the metacharacters and the result of using them in a regular expression. Therefore, the best way to

learn regular expressions is by example. The chapters that follow show many examples of regular expressions and the metacharacters that are used within them.

OPERANDS AND OPERATORS

Like an arithmetic expression, a regular expression contains operands and operators. Therefore, you can think of a regular expression as an expression like any other expression that contains operands and operators. In arithmetic expressions, operands are numbers, and operators are the plus sign for addition, the minus sign for subtraction, and so forth. In regular expressions, the operands are strings of characters, and the operators are the various metacharacters.

■ *FOR EXAMPLE:*

Consider the following regular expression:

```
chapter *[0-9]+
```

The operands are the characters "`chapter`," `0`, and `9`. The operators are the `*`, `[]`, `-`, and `+` metacharacters. Like arithmetic expressions, the operators use the operands to perform some kind of evaluation that produces some result.

Unlike an arithmetic expression, which usually results in a single value, the result of a regular expression is a list of substrings. The list of substrings form a subset of all substrings that are possibly formed from combining all characters that are printable from length one (single character) to the largest string that can be stored on a computer. Therefore, after evaluating the preceding regular expression, the result would be the list of strings:

```
{chapter 0, chapter 1, chapter 2, chapter 3, chapter
  4, …, chapter9, chapter0, …, chapter9}
```

Here, each value between commas is a substring that results from evaluating the expression. Therefore, you can think of a regular expression as a notation for specifying a subset of strings, or equivalently, a list of substrings. In other words, in our example, the notation `chapter *[0-9]+`

specified a list of substrings that begin with the text `chapter` followed by an optional space, followed by at least one occurrence of the digits 0-9.

Now, consider the following regular expression:

```
yes
```

When evaluated, this expression results in a single value:

```
yes
```

The regular expression forms a single substring of length three with the letters `y` followed by `e` followed by `s`. In these examples, a single element (in the last example, the single element is `yes`—that is, `y` literally followed by `e` followed by `s`) is selected from a subset of all possible substrings that could have been formed as regular expressions.

When a regular expression that contains metacharacter operators, called a *metacharacter expression*, is evaluated, the result is more than one substring. A regular expression that results in a single element is referred to as a *literal* regular expression, and therefore contains no metacharacter operators. In the preceding examples, the regular expression `chapter *[0-9]+` contains several metacharacters, as you saw, and resulted in more than one substring. The regular expression `yes`, on the other hand, contains no metacharacters and resulted in the single substring `yes`. Therefore, the simplest notation within regular expressions is to specify a single element from the set of all possible substrings or a literal regular expression.

However, thinking and describing regular expressions just as a means of simply selecting substrings tells half the story; it does not say what it will be used for and how it will be used. The result of a regular expression will be used for matching a possible set of strings. The set of strings to be matched is usually a line of text.

FOR EXAMPLE:

Consider the following regular expression:

```
her
```

and the following line of text:

```
feathered,
```

The literal regular expression `her` would match the string `feathered` (the string `her` is contained within the string `feathered` starting at the fifth character). Be sure to understand the view of a regular expression that, when evaluated, results in a string or set of strings that is used for matching (I say string(s) because when you use metacharacters, more than one string results and is used in matching). Also be sure to know that `her` exactly matches `her` as well as longer substrings that contain `her`, such as `feathered`.

Now that we know that regular expressions are used to match another substring, it's time to see where they are used and at what time the regular expression gets evaluated. You will see in more detail when we go over each utility (awk, sed, and grep) in what context regular expressions are used, but for now a couple of examples, without getting into details, should help more clearly define what they are. In grep, you usually want to find out within a line of text from an input file whether a particular string occurs. Therefore, in grep, the particular string to find is a regular expression, and the substring to match is each line in the file.

FOR EXAMPLE:

Consider the following:

```
grep her input-file
```

Here, the literal regular expression `her` is determined by grep as a regular expression matching argument to be compared with each line in the file `input-file`. If a match occurs, then the entire line is printed out. Within the grep program the literal regular expression is evaluated as an expression, not at the command line. This fact is important to know, because if the regular expression argument contains any metacharacters that are also interpreted as metacharacters at the command line, then the regular expression argument must be quoted, like so:

```
grep “her*” input-file
```

Here, the asterisk has special meaning at the command line, so to keep its regular expression meaning, it must be quoted.

Don't worry about syntax of grep at this point; that is covered in full detail in later chapters.

MICROCOSM/MACROCOSM

When working with regular expressions, two views are constructed that help determine the results of evaluating regular expressions, understanding what strings the regular expressions match, and finally constructing regular expressions. The two views are the macrocosm and microcosm. You can interpret the views as a partial (microcosm) or a whole (macrocosm) relationship.

In the macrocosm, we are concerned only about the results of evaluating a regular expression as a whole.

■ *FOR EXAMPLE:*

In the previous examples, we are concerned in the macrocosm that the regular expressions `her` and `chapter *[0-9]+` result as a whole in the regular expressions:

```
yes
chapter 1
chapter 2
chapter 3, and so on
```

We are also concerned in the macrocosm with the strings that the regular expressions match. In the macrocosm, the regular expression `chapter 1` matches the following strings:

```
chapter 1
chapter 11.
In chapter 1
```

The previous discussion has centered around the regular expression in the macrocosm. This view centered on a regular expression as an expression in the whole (operands, operators, results as sets).

In the microcosm, we are concerned with the regular expression and the result of each individual operation.

In the previous example, we are concerned with what results when we evaluate `[0-9]`. The expression results in

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

In other words, in the microcosm, we are concerned how the parts build up and result in the whole, the macrocosm. As another example, consider the regular expression `her`. In the microcosm, this is constructed by taking the characters `h` and `e` and concatenating them to form `he`, which is then concatenated to the character `r` to form the regular expression `her`.

In the microcosm, we are also concerned with the matching process itself. In other words, what in the regular expression matching process did a regular expression character match in the matching substring? In this process, we view regular expressions as patterns; like other patterns, the regular expressions specify a sequence and arrangement of characters that have to appear in the matching substring. In the previous regular expression `her`, the regular expression specifies that the arrangement of characters or pattern is such that the first character is an `h`, the second is an `e`, and the next character is an `r`, and that taken together they should be used to find out whether another string contains the same pattern. This suggests that, taken as the view of a pattern, a regular expression performs the following algorithm. We assume that the regular expression is `her` and the matching string is `feathered`.

1. Start at the leftmost character, `f`, in the matching string `feathered`.
2. Ask whether this character, `f`, is equal to the first character in the regular expression, or `h`.
3. Because the answer is no, the next leftmost character is used, `e`.
4. Repeat asking whether the next leftmost character in the matching substring is equal to the first character in the regular expression `h`.

5. We continue to answer no until we reach the h in the matching string `feathered`.
6. Here, the answer is yes, the next leftmost character in the matching substring is equal to the first character of the regular expression (`h` is equal to `h`).
7. We then ask whether the next leftmost character in the regular expression, `e`, is equal to the next leftmost character in the matching string, `feathered`.
8. Because the answer is yes, we continue to match the next leftmost character until all characters in the regular expression are matched in the matching substring, IN ORDER!!
9. Of course, if we reach the end of the matching substring before we reach the end of the regular expression, then the strings did not match.

At any time if the current leftmost character of the matching substring does not match the current character in the regular expression, then the current character in the regular expression is reset to the first character of the regular expression and the matching continues.

When we use regular expressions in examples and discussions of metacharacters in this chapter and in grep, sed, and awk, the macrocosm is used to show what the result of a regular expression is as a whole and the strings the regular expression matched as a whole. We use the microcosm view to describe each metacharacter and what characters or patterns they create and match and to explain the matching process. Maintain this part/whole view to better understand the functionality of each metacharacter expression and to make constructing and writing your own metacharacter regular expressions easier.

WILDCARD

I briefly mentioned that a metacharacter operator is a mechanism whereby *more* than one element may be selected from a list of all possible substrings. The first one discussed here is the wildcard metacharacter, and you will see what types of patterns the wildcard metacharacter matches. The wildcard metacharacter is denoted by the period character (`.`). In the previous discussion on literal regular expressions, you selected a single element or pattern from a list of many possible patterns. Sometimes,

however, the set of substrings you wish to match is too large to list out individually, so you need some notation that can be used to specify a number of substrings, or patterns, without having to individually list them all out. For example, suppose you wanted to specify all two-character substrings or patterns that start with the letter a. Having to type out aa, ab, ac, ..., az every time you wanted to specify two characters that start with the letter a would be tedious. The wildcard regular expression operator allows you to avoid this tedious task and provides a notation to instruct that when the wildcard is encountered, any character will match it. So the regular expression that provides this instruction is:

```
a.
```

In the macrocosm, the set of substrings from all substrings that will result from the evaluation of this regular expression are:

```
aa
ab
ac
az and so on
```

In the microcosm, the previous wildcard regular expression means that any pattern that contains the letter a followed by any other character will match the regular expression. Therefore, the following substrings will all match a:

```
aa
alfred    - a followed by l, l matches the wildcard
anytime   - a followed by n, n matches the wildcard
```

In the following wildcard regular expression, c.i, if the matching substring is chip, then the first character of the regular expression matches the first character of the matching substring. The second character of the regular expression is the wildcard metacharacter, so no comparison needs to be made in the matching substring (it will match), and the comparison moves to the next character position for both the regular expression and matching substring. This is continued until either a match is made or we reach the end of the matching substring. So anytime the wildcard metacharacter is encountered, it acts like a skip function. That is, we can skip the comparison for both the current character in the regular expression and the current character in the matching substring and move on to the

next character(s) in both and continue making comparisons. The only exception to the rule is the newline character. The wildcard metacharacter will not match the newline character. So consider the following wildcard regular expression:

```
chapter.
```

and matching substrings that are contained in a file one line after the other:

```
chapter
chapter one.
```

The regular expression would match only the second line. The reason the first line is not matched is that it contains the string `chapter`, which is followed by the newline. Because the newline does not match the wildcard metacharacter, the matching string does not match the regular expression (a newline occurs at the end of every line in the input file)

THE BACKSLASH METACHARACTER

Earlier in this chapter, I said that a literal regular expression is a regular expression that contains no metacharacters and results in a single string, and a regular expression that contains a metacharacter results in more than one single string. The backslash metacharacter is the only exception to the rule that any regular expression that contains a metacharacter will result in more than one string. The backslash metacharacter, denoted by the backslash character, \, results in a single literal regular expression.

■ *FOR EXAMPLE:*

Consider the following string:

```
"chapter."
```

We would like to use the string as a regular expression and match it with all lines containing the word `chapter` followed by a period. How can we construct this regular expression while at the same time not having the period be evaluated as the wildcard metacharacter, which matches any

character? We need some mechanism so that we may interpret the period literally and not as the period metacharacter.

The backslash metacharacter enables us to instruct that the character following the backslash metacharacter be interpreted literally and not as a metacharacter. Any metacharacter operator may be used as a literal character by preceding it with the backslash metacharacter. So in the previous example, we would write our regular expression as follows:

```
chapter\.
```

This would correctly select any substring with the pattern `chapter.` Again, the backslash character does not result in multiple strings being constructed as a result of the evaluation. Unlike all other metacharacters, it results in a literal or single element and can be interpreted as a literal regular expression. The duality that metacharacters are both operators and literal characters forces the need of the backslash metacharacter.

LAB 1.1 EXERCISES

These exercises will test your understanding of the discussion presented in this chapter. A number of these exercises will not be very difficult. We will challenge and reinforce your knowledge of regular expressions more as we go over more metacharacters. For now, the more important goal is that you understand the basics and answer simple questions before attempting more advanced questions.

1.1.1 IDENTIFY THE OPERANDS AND OPERATORS OF A REGULAR EXPRESSION

What are the operands and operators of the following regular expressions?

a) `Her`

_____

b) `feathered`

_____

c) `Her.`

_____

d) `Feathered\.`

_____

e) `2\*.`

_____

f) `chapter\n`

_____

What type of regular expression is each of the following?

g) `the`

_____

h) `the.`

_____

i) `the\.`

_____

1.1.2 UNDERSTAND THE RESULTS OF A REGULAR EXPRESSION CONSISTING OF OPERANDS AND OPERATORS

What are the results of evaluating the following regular expressions? Think of the expressions as operators and operands, and the results as a set of literal strings.

a) `a.b`

_____

b) `80.86`

_____

c) `her`

_____

d) `\.`

_____

e) `\\t`

_____

1.1.3 IDENTIFY WHEN A REGULAR EXPRESSION IS EVALUATED

a) Identify when the regular expressions in the following code or invocations are evaluated as a regular expression.

```
grep t* input-file

awk { x = 3
    if ($0 ~ x) print $0
    }

grep 't*' input-file
```

_____

_____

LAB 1.1

1.1.4 UNDERSTAND THE RESULTS OF EVALUATING THE EXPRESSION AS A PATTERN

Use the following regular expressions and matching substrings to answer the questions in this Exercise.

```
reg_expr = her
match string = her, hereafter. Heresy

reg_expr = a.c
match string = abc, acdc, a$c, access

reg_expr = try\.
match string = trying, try$, try.
```

a) Which matching strings match the regular expression?

_____

_____

b) At what character position(s) does the first character of the regular expression match in the matching string?

_____

_____

c) At what character position(s) in the matching string does a match not occur?

_____

_____

1.1.5 UNDERSTAND THE WILDCARD CHARACTER

What strings do the following wildcard regular expressions produce?

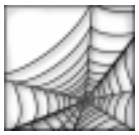
a) `the.`

_____

b) `.ed`

_____

LAB 1.1 EXERCISE ANSWERS



This section gives you some suggested answers to the questions in Lab 1.1, with discussions related to those answers. Your answers may vary, but the most important thing is whether or not your answers work. Use these discussions to analyze differences between your answers and those presented here.

If you have alternative answers to the questions in this Exercise, you are encouraged to post your answers and discuss them at the companion Web site for this book, located at:

`http://www.phptr.com/phptrinteractive/`

I cannot emphasize enough that the best way to learn computer languages and utilities is by practicing. Try out sample queries on your own. In addition, I have found that sometimes the best way to learn is by your own mistakes. Try figuring out where you went wrong, and try entering a query that you suspect might not work. You might be surprised at the result.

LAB 1.1

1.1.1 ANSWERS

What are the operands and operators of the following regular expressions?

a) Her

Answer: Operands: H, e, and r Operator: concatenation

No explicit operator exists, and the implicit operator concatenation is used to concatenate h with e and then r.

b) feathered

Answer: Operand: feathered Operator: concatenation

Same reasoning as the previous example.

c) Her.

Answer: Operand: H, e, and r, and all single characters Operator: concatenation, wildcard

In this particular expression, we encounter the wildcard operator. The operand to this operator are all single characters, by definition, of the wildcard operator. This is concatenated with the literal expression Her, which was constructed by concatenating h, e, and r.

d) Feathered\.

Answer: Operand: Feathered. Operator: concatenation, backslash

The backslash operator is encountered next, whose operand is the period. This is concatenated to the literal string Feathered.

e) 2*.

*Answer: Operand: 2, *, all single characters Operator: concatenation, backslash, wildcard*

Here, the wildcard operator has as its operands all single characters, which are concatenated to the character *, which is the operand to the backslash, which is then concatenated to the literal character 2.

f) chapter\n

Answer: Operand: `chapter`*, newline character Operator:* `concatenation`*,* `backslash`

What type of regular expression is each of the following?

g) the

Answer: Literal regular expression, no metacharacters.

h) the.

Answer: Metacharacter regular expression. In particular, a wildcard metacharacter regular expression.

i) the\.

Answer: Literal regular expression; it has a metacharacter but the backslash still evaluates to a literal.

1.1.2 ANSWERS

What are the results of evaluating the following regular expressions? Think of the expressions as operators and operands, and the results as a set of literal strings.

a) a.b

Answer: `{aab,…,azb,a1b, ..,a9b,a!b, ..,a\fb}`

In other words, all three character strings, including punctuation and non-printable characters like the form feed. Everything except the newline character.

b) 80.86

Answer: Again, because the wildcard matches all characters except the newline, the result is the same as before, except that this one is prefixed by `80` *and suffixed by* `86`*.*

c) her

Answer: `{her}` *—the single element* `her`*.*

d) \.

Answer: `{.}` —the single element period, which was the operand to the backslash. Therefore, the backslash returns the operand following it literally.

e) \\t

Answer: `{\t}` —the single element consisting of two characters, a backslash, and the character `t`. The backslash operator has as its operand a backslash. Therefore, the backslash will be taken literally, and the literal character `t` is concatenated to it.

1.1.3 ANSWERS

a) Identify when the regular expressions in the following code or invocations are evaluated as a regular expression.

```
grep t* input-file
```

Answer: No regular expression exists.

The shell will evaluate the asterisk, because it is not quoted. The shell interprets the expression `t*` as a filename.

```
awk { x = 3
      if ($0 ~ x) print $0
    }
```

The regular expression `x` will be evaluated as a regular expression when `awk` is executing, and `awk` reaches the line of code `if ($0 ~ x) print $0`.

In awk, frequently we would like to know whether a string pattern or substring is contained within a line of text from a file. In awk, unlike grep, we would also like to know whether a substring is contained within a portion (or field) of the input line. We might also want to know simply whether a substring or pattern is contained within a variable that contains a string. Therefore, because awk allows string variable assignment, it does not limit the string to which the regular expression will match simply to an input line like grep.

This awk snippet searches for the literal regular expression `3` but further searches whether the regular expression is contained within the string variable `$0` and not an entire input line of text. The tilde (~) is an awk regular expression matching operator. When awk encounters the operator, the operator requires that one of its arguments be a regular expression that

needs to be evaluated. Because a regular expression is enclosed in slashes in awk, it recognizes the second argument (rightmost operand) `/her/` as the regular expression that needs to be evaluated and uses `her` as the regular expression.

It then determines whether the regular expression `her` is contained within the substring that is stored in the variable `word`. In awk, a regular expression is recognized not simply as a string, but as an expression, a regular expression, that needs to be evaluated. We could have rewritten the awk snippet of code as:

```
awk '{reg_expr = "her"
      word ~ reg_expr
}'
```

This would have returned equivalent results. However, the string `her` was not evaluated as a regular expression in the assignment statement, but simply as a string. Only when the string is encountered with a regular expression operator is the string recognized as a regular expression to be evaluated. Therefore, the context in which the string is used determines whether it is a regular expression or not.

```
grep 't*' input-file
```

Answer: The regular expression is evaluated when grep is executing.

1.1.4 ANSWERS

Use the given regular expressions and matching substrings to answer the questions in this Exercise.

a) Which matching strings match the regular expression?

Answer: `her` *—matches* `her` *and* `hereafter`*, but not* `Heresy`*; the first character of* `Heresy` *is capitalized.*

`a.c` *—matches,* `abc`*,* `a$c`*, and* `access`*, but does not match* `acdc`*. The pattern of the character* `a` *followed by any character and then a* `c` *never occurs in* `acdc`*.*

`try\.` *—matches only* `try.`*. The others do not have the period character following the string* `try`*.*

b) At what character position(s) does the first character of the regular expression match in the matching string?

Answer: `her` *—the first character,* `h`*, matches in position 1 in* `her` *and* `hereafter`*, and never matches in* `Heresy`*.*

`a.c` *—the first character,* `a`*, matches in position 1 for all matching strings.*

`try\.` *—the first character,* `t`*, matches in all substrings.*

c) At what character position(s) in the matching string does a match not occur?

Answer: `her`*,* `hereafter` *—no positions do not match (positions 1, 2, and 3 are matched and the matching stops).*

`Heresy` *—no position is a match.*

`abc`*,* `a$c`*,* `access` *—no positions match.*

`acdc` *—positions 2, 3, and 4 do not match.*

1.1.5 ANSWERS

What strings do the following wildcard regular expressions produce?

a) the.

Answer: `{thea, …, thez, theA, .., theZ, the1, .., the9, the$, .., the\f, theaa, theaz, …}`

All strings length four and greater that are prefixed by the string `the`. Newline cannot follow the string `the`, but could come after it.

b) .ed

Answer: {aed, .., zed, Aed, …, Zed, 1ed, .., 9ed, $ed, .., \fed, aeda, .. zeda, ..}

All strings that have any character suffixed by `ed`, and followed by zero or more characters after the three characters preceding it.

LAB 1.1 SELF-REVIEW QUESTIONS

In order to test your progress, you should be able to answer the following questions.

In each of the following multiple choice questions, the answer may be one or more of the available choices.

1) Which of the following are a result of evaluating the regular expression `123*[a-c]`?

a) _____ `12`
b) _____ `12a`
c) _____ `123ad`
d) _____ `12cc`
e) _____ `12d`

2) What type of regular expression is `chapter\.` ?

a) _____literal
b) _____metacharacter
c) _____it is not a regular expression

3) What characters does `.` match?

a) _____a letter a-z or A-Z
b) _____a number
c) _____a metacharacter

4) What are the operands of the regular expression `cd*e` ?

a) _____c
b) _____d
c) _____e
d) _____1
e) _____3

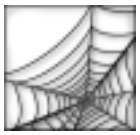
5) Does the string `sentence.` need the backslash metacharacter to be interpreted literally?

a) _____yes
b) _____no

Quiz answers appear in Appendix A, Lab 1.1.

CHAPTER 1

TEST YOUR THINKING



The projects in this section are meant to have you utilize all of the skills that you have acquired throughout this chapter. The answers to these projects can be found at the companion Web site to this book, located at:

`http://www.phptr.com`

Visit the Web site periodically to share and discuss your answers.

In the rest of this section we will explore additional metacharacters including the asterisk, plus, and question mark metacharacters. These three metacharacters are described in the following table:

Table 1.1 ■ Asterisk, Plus, Question Mark, and Positional Metacharacters

Operator	Description	Usage
Asterisk Metacharacter (*)	`A* matches zero or more occurrences of A in a matching sub-string`	Regular expression a* matches substrings a, aa, after, daal, fred. In the first substring a, the regular expression a*; a occurs one time in a. In the substring aa, the regular expression a*; a occurs twice in aa, and daal. In the substring fred, the regular expression a*; a occurs zero times. (remember that the asterisk metacharacter specifies that zero or more occurrences of the character immediately preceding the asterisk metacharacter may occur in the matching substring
Plus Metacharacter (+)	`A+ matches one or more occurrences of A in a matching substring`	Regular expression a+ matches the same substrings a, aa, after, daal but does not match fred. The plus metacharacter specfies that at least one occurrence of the character immediately preceding the plus metacharacter may occur in the matching substring. The matching substring fred has zero occurrences of a in the string fred.

Table 1.1 ■ Asterisk, Plus, Question Mark, and Positional Metacharacters

Question Mark Metacharacter (?)	`A? matches exactly zero or one occurrences of A in a matching susbtring`	Regular expression a? matches the substrings a, after, and fred but does match aa, and daal. In a, and after the regular expression a? matches one occurrence of a. In fred the regular expression a matches zero occurrences of a.
Positional Metacharacters (^,$)	`^A matches A at the very beginning (first character) of a matching substring` `A$ matches A at the very end (last character) of a matching substring`	Regular expression ^a matches the substrings a, after, but does match fred, and daal. In a, and after the regular expression ^a matches the a in the beginning (first character) of the substring a. and after. In fred , and daal, the regular expression ^a does not match an a in the beginning (first character is d and f and not a) of the substrings fred and daal.

Each of the above metacharacters differs from the metacharacters in Lab 1.1 in that the metacharacters in Table 1.1 specify and affect metacharacters and literals not at the current character position but at a position that immediately precedes the metacharacter or immediately follows the metacharacter.

1) For each of the following expressions, identify the subset of values that would result.

a) AB+C

b) AB?C

c) AB*C

d) ^AB*C

e) AB*C$

f) ^AB*C$

2) Explain verbally what the following regular expression produces. Which lines does it match and match?

" book.* "

Use the following input file called `mic.dat` for this section.

```
There are two principal types of transducers used in
  mics: dynamic and
condensor.  Dynamics are often favored for miking
  individual
instruments because they add a favorable color to
  the sound.  Condensor
mics are generally more accurate than dynamic and
  are preferable for
audience recording.
```

1) What is the result of the following regular expressions used within grep?

Try to figure not only what the regular expression matched when you execute it at the terminal, but what other strings in other files it might match.

a) grep 'dynamic ' mic.dat

b) grep 'dynamic[.?!]' mic.dat

c) grep 'condenser' mic.dat

d) grep '\. *' mic.dat

e) grep '\. *D' mic.dat

f) p [TD]* mic.dat

2) Suppose you wanted to find all matching strings that have a file name that starts with AVL and has a file extension of `.h` (the rest of the name you do not care about).

g) Write a regular expression that would perform this search.

3) For the following regular expressions, explain verbally what each does.

h) ^.*$

i) ^$

j) "[.?!:] +[a-zA-Z]"

k) ^[+-]?[0-9]+[.]?[0-9]*$

l) ^[0-9][0-9]$