Lab 3: Regular expressions. Shell Script.

1 Introduction

- **Topics:**
  - Regular expressions
    - basic rules (see also course notes and UNIX in a nutshell, ch 6)
    - {} in more details
    - examples
  - Use of REs in various UNIX applications:
    - grep, egrep
    - vi,
    - sed,
    - emacs
  - short shell script
    - variables
    - return value of each UNIX command
    - if
    - for
    - case

2 Regular Expressions (RE)

RE is a way of describing a set of strings without having to list all the strings in your set. RE contains instructions on how to match the pattern. This is accomplished by using plain text and special tokens. A sample RE may look like:

hello

This matches any bit of text which contain *hello*. It will not match *"Hello"* in *"Hello, world"* since REs are case sensitive. REs may also contain tokens. These tokens are used in place of other text. For example *hel*o* is a RE, where *** is a token. As we will see later, *** means 0 or more repetitions of the previous character. Our RE will match *heo*, as well as *helo*, *hello* or *hellllllllo*.

REs are used by many UNIX programs (*grep, egrep, sed, perl, vi, emacs* etc.). Tokens may vary from program to program and you should refer to manual for the program to see, which tokens are supported. However the following table gives the basic rules used by most programs:

2.1 Rules for forming (basic) RE:

1. A non-special character is a RE matching that character.
Example:

a
6
%

Special characters are:

"."	period
""*	asterisk
"["	left square bracket
"\\"	backslash
"^"	caret
"$"	dollar sign

Note: A special character used as "normal" must be preceded by a backslash.
Example:

Main\\.m3
first\\.java

2. If token *A* is RE and token *B* is RE, then token *AB* is also RE
Token AB is called a concatenation of REs *A* and *B*. RE first matches *A* then *B*.
Example: RE *cs* matches the string "cs" because it first matches *c* and then *s*.

3. *(A)* matches anything that matches token *A* (precedence relations or grouping)

Notes:

- "(" is not listed as a special character. As a result, in many UNIX applications, which support "(" we have to use "\(" to preserve its special meaning.

- "()" has also another meaning : it will store the pattern for later replay. (see UNIX in a Nutshell)

4. $A|B$ matches anything that matches token A or token B.
Example :

a) $a|b$
b) $(ab)|(cd)$.

Similarly as before, since | is not listed as a special character, \ is used to preserve its special meaning. And so,

a) will be written as:$a\backslash|b$
b) will be written as em \(ab\)\|\(cd\)

5. *A , where token *A* is a RE, matches zero or more repetition of *A***
Example:

*a** will match:
"" - an empty string
a - one repetition of a
aaaaaaa - more then one repetition of a etc.

*w(ab)** will match:
w
wab
wababab etc.

6. '.' (single period) matches any character except newline \n.
Example:

Main.m3 will match:
MainAm3
Main7m3 etc.

Recall, that if *Main.m3* should be matched exactly, the appropriate RE has to be written as *Main\\.m3*.

***[ab]* - character classes**

$[ab]$ means (a|b)
$[abcdefg]$ means$(a|b|c|d|e|f|g)$

Note, that a shortcut like $[a-g]$ is admissible. In case, the character '−' is really one of the character in [], '−' has to be putted as first or last character in the list. For example: $[a-g-]$ will match $a|b|c|d|e|f|g|-$.

Examples:

$[A-Za-z][A-Za-z0-9\_]*$ matches any modula3 identifier or keyword
$[A-Z][A-Z]*$ matches any sequence of at least one capital letter

7. [ˆ*ab*] inverted character classes

Examples: $[\hat{}ab]$ matches anything except $[ab]$

$[\hat{}a-q]$ will match t or u or z but not a or d.

8. $\hat{}A$ - where A is a RE, matches anything that matches A, but only if it occurs at the beginning of the line.

Example:

$\hat{}[A-Z]$ matches any capital letter at the beginning of the line.

9. $A\$$ matches anything that matches A but only if it occurs at the end of a line

Example: The following RE finds all lines that contain only one word END, perhaps with spaces before and after it:

$"\hat{}\ \ *END\ \ *\$"$

Some additional rules not supported by all UNIX applications:

10. $A+$ matches one or more repetitions of A.
Note: $A+$ is the same as $AA*$ and many UNIX applications do not support '+' at all. Consult man pages of the particular application for what is or is not supported.

11. $A?$ matches zero or one repetition of A
Note: ? is the same as $(A|\epsilon)$

Examples:

$a1p[1-4]\backslash.txt$ test cases for the assignment 1

$\hat{}[.]$ any character but '.' at the beginning of the line.

$H[aeiou]llo$ second letter is a vowel

$bugs*$ bug, bugs, bugssss

Multiple matches:
$\{n,m\}$ must occur at least n times, but no more than m times.
$\{n,\ \}$ at least n times
$\{\ \ ,m\}$ no more than m times
$\{n\}$ exactly n times
$*$ 0 or more times (same as $\{0,\ \ \}$)
$+$ 1 or more times (same as $\{1,\ \ \}$)
$?$ 0 or 1 time (same as $\{0,1\}$)
Examples:

$quu*x = qu+x = qu\{1,\ \ \}x$

$qu|qux = qu?x = qu\{0,1\}x$

etc.

2.2 Programs Using Regular Expressions

Many UNIX programs and tools use Regular Expressions. In this sections we mention *grep*, *sed* and *vi*. For a complete description refer to the UNIX in a nutshell book.

2.2.1 Grep and Egrep

Grep and *egrep* are tools that search for patterns in files and find all lines containing a particular string.
Format:

grep RE filename

(see UNIX in a nutschell and man grep for more detailed description)
Examples:

grep "youruserid" marks - prints all lines which contain 'youruserid' in the file 'marks'

grep "" classlist* - prints all lines which contain a character '*'.

grep "[Pp]roced." *.java* - prints all lines which contain Proced or proced in all files with extension java.

Note the significant difference between '*' as used in RE and '*' used in shell. Since shell may be easily confused by '*' or any other special character in regular expression, it is always recommended to quote (by using " or "") RE.

2.2.2 RE in vi:

Another application which uses REs is the *vi* editor. For purpose of an exercise we will create a file 'temp' in the current directory. Open a file by typing:

vi temp

and enter the following two lines:

Tooting and froing
until tomorrow morning

Now suppose we want to search for a string starting with 't' and ending with 'ing'. The correct regular expression is: *t.*ing*
Switch into the command mode by pressing

ESC

and type

*/t.*ing* (*ENTER*). The cursor will point to the first character of the longest match '*ting and froing*'.
Press

/ (*ENTER*) and the cursor will move to the next match: '*til tomorrow morning*'.

For more options refer to the UNIX in a nutshell book.

2.2.3 RE in sed:

SED means a *S*tream *ED*itor. SED does not change the file it edits. It takes data from standard input or a file, transforms them and passes them into a standard output. The UNIX command is *sed* and we will show only a few examples of its application for finding or replacing a string in the given text. More detailed description can be found in UNIX in a nutshell.

Assume that we want to replace all occurrences of *t.*ing* in temp with *HELLO*. We can do so using:

```
cat temp | sed 's/t.*ing/HELLO/g'
```

or

```
sed 's/t.*ing/HELLO/g' temp
```

Notice that the longest match is replaced.
Output goes to stdin.

3 The Shell Script

As mentioned in lab 2, scripting languages are similar to batch files in DOS but are more powerful. They include variables, looping, conditional execution etc. We already saw how to write a simple shell script, here we introduce variables, conditional statements and loops. For a full description refer to the UNIX in a nutshell book. There are also several courses available on the WEB. For example a very nice introductory course in bourne shell can be found here:

http://www.emerson.emory.edu/services/unixhelp1.3/Pages/scrpt/

3.1 The Basics

A script should start with a line

```
#!/bin/sh
```

Normally, '#' introduces a comment line in shell. However, *#!* is a special comment. It tells the shell that it is a script and should be run using the program *sh* (bourne shell) in the bin directory. A script in korn shell will start with the line

```
#!/bin/ksh
```

for example.

3.2 Variables

Shell variables work on the principle of 'variable substitution'. This is different from nonscripting languages such as c++. If you want to use the name of a variable, type the name itself. If you want to substitute its value, type the dollar sign *$* followed by name. In case the value of the variable contains spaces, we have to use pair of double quotes (see the example below). To delimit name of the variable one has to use curly braces {}, as shown in the example.

Example:

```
#! /bin/sh
class=cs241
longname="I like cs241"
echo class                 #prints class
echo $class                #prints cs241
echo $classroom            #prints nothing
echo ${class}room          #prints cs241room
echo $longname             #prints I like cs241
```

After execution the output will be:

```
class
cs241
cs241room
I like cs241
```

3.3 Command Line Parameters

When shell script is started, variables $0 to $9 are set to values of parameters passed from the command line.

Example: Create a script file *test.sh* by using *vi* editor:

```
vi test.sh
    #!/bin/sh
    echo Parameter 1: $1
    echo Parameter 2: $2
```

Save the file, allow execution by changing the permission and execute it with two input parameters *hello* and *world*:

```
./test.sh hello world
```

The following output will be displayed:

```
Parameter 1: hello
Parameter 2: world
```

3.4 Conditional Statements

3.4.1 The if statement

A typical example of the *if* statement is:

```
if bin/testprg
then
    echo return from the testprg is true
```

```
else
    echo return from the testprg is false
fi
```

Every UNIX command returns on exit a value, which the shell can use. This value is held in the read-only shell variable $?. The value 0 (zero) signifies success; anything other than 0 (zero) signifies failure. In our example, if testprg executes successfully, the return value is 0 and the line:

```
return from the testprg is true
```

is printed. In case the execution was not successful, the line

```
return from the testprg is false
```

is printed.

The *if* statement is often used with the test program stored in */usr/bin* directory. The test program has many parameters enabling it to check whether strings are equal or different, if the file exists, and much more. The complete description is in UNIX in a nutshell book. Sometimes the shorthand [ ] for the *test* program is used.

Example: Test if the file test.sh exists:

```
if [ -f test.sh ]
then
    echo file exists
else
    echo file not found
fi
```

Note the syntax of the *test* command: [−*f file*]. The spaces after '[' and before ']' are necessary.

The general syntax of the *if* statement is:

```
if test
then
    commands            (if condition is true)
else
    commands            (if condition is false)
fi
```

then, *else* and *fi* are shell reserved words and as such are only recognized after a newline or ; (semicolon). The *if* construct must end with a *fi* statement. *if* statements may be nested:

```
if ...
then ...
else if ...
    ...
fi
```

```
fi
```

The *elif* statement can be used as shorthand for an *else if* statement. For example:

```
if ...
then ...
elif ...
    ...
fi
fi
```

3.4.2 The case statement

The *case* statement starts with the keyword '*case*' followed by the value to be tested and the keyword '*in*'. This is followed by series of options. Each value to be tested is on a separate line with a closing parenthesis ')' after it. The code which should be executed comes after the value. After that there should be a line with double semi-colon ;;

The general syntax of the case statement is:

```
case value in
pattern1) command(s)
;;
pattern2) command(s)
;;
patternN) command(s)
;;
esac
```

After all the commands are executed, the control is passed to the first statement after the *esac*.

Several values can be matched in the same line. They must be separated from each other by a | symbol. For example:

```
case value in
pattern1|pattern2) command
... ;;
```

Patterns are checked for a match in the order in which they appear. A command is always carried out after the first instance of a pattern.

3.4.3 The for statement

The *for* loop notation has the general form:

```
for var in list-of-words
do
```

```
    commands
done
```

commands is a sequence of one or more commands separated by a newline or ; (semicolon).

The reserved words *do* and *done* must be preceded by a newline or ; (semicolon). Small loops can be written on a single line. For example:

```
for var in list; do commands; done
```

This is a simple example of a *for* loop and a 'command substitution':

```
for i in `ls `
do
    echo "the name of the file is $i"
done
```

In this example, the command *ls* is executed first, since it is used inside back-quotes. Then the values are substituted to *i* and proper message is printed. Suppose that the result of the *ls* command is:

```
first
test.sh
```

Then the output from the *for* loop is:

```
the name of the file is first
the name of the file is test.sh
```

For information on *while* loop and other useful commands used refer to the UNIX in a nutshell book.

3.5 Briefly About Linux

Linux is a UNIX -like operating system designed for personal computers. It is freely available on the WEB:

http://www.debian.org/
http://www.redhat.com/

You may also purchase an installation disk with software needed for all cs courses in the Computer Club of UW or visit the book store.

Install Linux on your PC and enjoy!