Expect Basics

Tutorial 3

ECE453 & CS447

Software Testing, Quality Assurance and Maintenance

Tutorial Overview

1. ***Central commands of Expect***
2. ***Interact with other processes***
3. ***Error and timeout handling***
4. ***Glob Pattern Matching in Expect***
5. ***Regular Expression pattern matching in Expect***

Getting Started with Expect

- **Executing Expect**
  - Using Expect interpreter
  - Executing Expect script files from Unix.
  - For simplicity, > is used as prompt for Expect interpreter.

```
$ expect
expect1.1> exit
$
```

```
$ expect script-filename
……
```

- **Three central commands: *send*, *expect* and *spawn*.**
  - *send*: send a string to a connected process.
  - It does not format string in any way.
  - Can save the command to a file and execute it from Unix shell directly. (assume *speak* is that file)

```
> send "Hello world"
Hello world>
```

```
$ expect speak
Hello world$
```

Command: *expect*

expect **command waits for a string from a process, the default is from keyboard.**

Syntax:
expect [pattern-string {action}]

It waits for first occurrence of pattern and stop only when match is found or timeout.

The match pattern string is stored in *expect_out(0, string)* and the entire string read is kept in *expect_out(buffer)*.

The remain string "cal<newline>" is still in queue and available for next expect command.

```
> expect "hi"
philosophic                //input from keyboard
>
```

```
> expect "hi" {
   send "hello there\n"
  }
philosophic                //input from keyboard
hello there                //command output
>
```

```
> expect "hi" {
   send $expect_out(0, string)\n
   send $expect_out(buffer)\n
  }
philosophical              //input from keyboard
hi                         //value of expect_out(0,string)
phi                        //value of expect_out(buffer)
>
```

Pattern-Action Pair for *expect*

expect can search several patterns simultaneously (like *switch-case*)

```
> expect "hi"     { send "hello there\n"
  }          "hello" { send "hello yourself\n"
  }          "bye"   { send "this is unexpected\n" }
}
tmklbyetasd              //input from keyboard
this is unexpected       //command output
>
```

Another format to handle long line command.

```
> expect { "hi"     { send "hello there\n"}
               "hello" { send "hello yourself\n"}
               "bye"   { send "this is unexpected\n"}
   }
>
```

```
> expect "hi" { send "hello there\n"} "hello" { send "hello yourself\n"}
> expect "hi" send "hello there\n"
```

Incorrect command format

Correct command format

Command: *spawn*

spawn command starts a process and treats user as process too.

Syntax:
spawn program-name argument-list

Example for create a ftp connection to a server in Unix shell

```
$ ftp ftp.uu.net
Connected to ftp.uu.net
220 crete FTP server (SunOS 5.6) ready.
Name (ftp.uu.net:jlian): anonymous
331 Guest login ok, email addr as passwd.
Password:
230 Guest login ok, access restricted.
ftp> quit
$
```

Example by using *expect* to do same thing automatically.

These commands needs to be saved in a script file.

```
spawn ftp ftp.uu.net
expect "Name"
send "anonymous\r"
expect "Password:"
send "jlian@swen.uwaterloo.ca\r"
expect "ftp> "
send "quit\r"                    //quit from ftp
```

To simulate user press *return* key, use \r, don't use \n.

ftp Example

This example uses a script file to establish a connection to specified server (ftp.uu.net) and copy a file from specified directory.
This file is saved in a file named *run*. It is executed from Unix shell directly by using expect.

```
$ expect run tmp milk.c
spawn ftp ftp.uu.net
Connected to ftp.uu.net.
…
230 User logged in.
ftp> cd tmp
250 CWD command successful.
ftp> get test
150 Opening … for milk.c (1149 bytes).
226 Transfer complete.
local: milk.c remote: milk.c
1200 bytes received in 0.16 seconds …
$
```

```
# copy file from specified directory
# in a specified server by ftp

if { [llength $argv] < 2} {
  puts "usage: expect run dir file"
  exit 1
}
set timeout -1
spawn ftp ftp.uu.net
expect "Name"
send "anonymous\r"
expect "Password:"
send "jlian@swen.uwaterloo.ca\r"
expect "ftp> "
send "cd [lindex $argv 0]\r"
expect "ftp> "
send "get [lindex $argv 1]\r"
expect "ftp> "
send "quit\r"
```

Command: *interact*

***interact* command gets back control from expect to user. After execution, expect stops reading command from script. The user input from keyboard will be directly sent to spawned process and its output is sent to standard output .**

When error occurs during execution ftp commands, the user gets control back.

```
set timeout 30
…                          //establish connection
expect timeout {
    interact
    puts "unexpected…"
  } "ftp> " {
     send "do some ftp commands\n"
  } "^\[45]" {       //this is common error #
    interact
    puts "Error … "
}
…
```

When no matching found or no input available, expect waits for a period specified by *timeout* variable, (default 10s). It should be an integer.
We can use *timeout* as a special pattern in expect (no double quota). Example shown above. It catches *timeout* signal and performs related actions.

Glob Patterns of Expect

- **Three glob pattern *, ? and [ ] inherited from Tcl**
  - *: match the longest possible string
  - ?: match any single character.
  - [ ]: match any single character specified in [ ].
- **Examples**
  - a?b: match a2b, but not adcb.
  - [abcdef0123456789]: match any hexadecimal digit
  - [a-f0-9]: same as above
- **Special treatment for []**
  - Example doesn’t work
  - Two solutions

```
> expect “[a-f]” { …}
invalid command name "a-f"
>
```

```
> expect “\[a-f]” { …}
```

```
>expect {[a-f]} { …}
```

The * Wildcard

The * match *longest* string from input.

```
> expect "hi*" {
   send "$expect_out(0,string)--";
   send $expect_out(buffer) }
philosophical          //input from keyboard
hilosophical     //value of expect_out(0,string)
--
philosophical    //value of expect_out(buffer)
>
```

The * matches "losophical"

The * matches "losop"

```
> expect "hi*hi" …   //same as above
philosophical        //input from keyboard
Hilosophi--          //expect_out(0,string)
philosophi           //expect_out(buffer)
>
```

The first * matches "philosop"
The second * matches "cal"

```
> expect "*hi*" …     //same as above
philosophical         //input
philosophical         //expect_out(0,string)
--philosophical       //expect_out(buffer)
>
```

Using *expect "*"* match everything, including empty string. All input current available is thrown away.

Usage of Backslashes (1)

- **Rules of backslash:**
  - Tcl shell translate backslash string first in its way.
  - Expect pattern matcher evaluates translated result.
    - Backslash disable the function of wildcard.
    - Backslash does nothing for non-special characters.
- **Examples**
  - expect “\n”; #matches \n (linefeed char)
  - expect “\r”; #matches \r (return char)
  - expect “\z”; #matches z (literal z)
  - expect “\{”; #matches { (literal left brace)
  - expect “*”; #matches everything
  - expect “\\*”; #matches literal *
  - expect “*”; #matches what?

equivalent

Usage of Backslashes (2)

- **In the pattern matcher, linefeed character is not same as *\n* in Tcl interpreter. The pattern matcher never translates string “*\n*” to linefeed. Translation has been done by Tcl.**
- **Examples**
  – expect “n”;   #matches literal n
  – expect “\n”;   #matches \n (linefeed char)
  – expect “\\n”;   #matches n
  – expect “\\\n”;   #matches \n (linefeed char)
  – expect “\\\\n”;   #matches sequence of \ and n
  – expect “\\\\\n”;   #matches sequence of \ and \n (linefeed)
  – expect “\\\\\\n”;   #matches sequence of \ and n
  – expect “\\\\\\\n”;   #matches sequence of \ and \n (linefeed)
  – expect “\\\\\\\\n”;   #matches sequence of \ and \ and n

Usage of Backslashes (3)

- **The "[" is special for Tcl and pattern matcher.**
  - It can be command execution prefix for Tcl
  - It can be range matching pattern for expect
  - It can be common literal in a pattern
- **Examples**

  let XY denote a procedure which return a string "n*w"
  - expect "[XY]";   #matches n followed by anything followed by w.
  - expect "\[XY]";   #matches literal X or Y
  - expect "\\[XY]";   #matches n followed by anything followed by w.
  - expect "\\\[XY]";   #matches sequence [XY]
  - expect "\\\\[XY]";   #matches \ followed by n followed by …
  - expect "\\\\\[XY]";   #matches sequence of \ followed by X or Y

Handling Error and Timeout

Using *timeout* and other patterns to detect errors when establishing connection and using exit to terminate the execution of script.

Matches a new-line followed by a error code 4 or 5.

Matches the current buffered string start with 4 or 5. If there are unread string left after last expect, this may cause errors.

```
…
set timeout 60      //getting file takes time
spawn ftp somewhere
expect {
  timeout   {puts “timed out”; exit}
  “connection refused”  {exit 1}
  “unknown host”        {exit 1}
  “Name”
}
send “anonymous\r”
…
expect {
  timeout  {unexpected…}
  “\n\[45]”  {errors… }
  “^[45]”  {errors… }
  “ftp> ”
}
…
```

Handling End of File (*eof*)

- **Two cases:**
  - When the spawned process closes its connection to Expect, Expect sees an *eof*.
  - When expect closes its connections, spawned process sees an *eof* and "*hangup*" signal.
- ***close* command**
  - Whenever one side closes connection, the other side should close connection as.
  - Expect uses ***close*** command to close its connection.
  - In most case, expect closes connection automatically.

Using *eof* to catch the connection closing event.

```
…
expect {
  timeout   {puts "timed out"; exit}
  eof       {do some actions}
  …
}
```

Regular Expressions (regexp)

- **Regular expression is more powerful than Glob pattern.**

Patterns in common

glob	regexp	Meaning
s	s	literal s
^	^	beginning of string
$	$	end of string
[a-z]	[a-z]	range matching
?	.	any single character
*	.*	any string

Special in regexp

regexp	Meaning
[^a-z]	Any char not in the range
a*	any number of “a”
a+	Non-empty “a” sequence
[0-9]*	any decimal sequence
a?	Match “a” or “” only
|	Match any of branches

- **Example of matching any decimal, hex and octal.**

“-?[1-9][0-9]*|0x[0-9a-fA-F]+|0[0-7]*”

Identify Patterns

- **Identify regexp and glob pattern**
  - The default pattern in *expect* is glob pattern.
  - Using –re and –gl to identify regexp and glob pattern

    expect –re "a"* # match "", "a", "aa", …

    expect "a"* # match sequence of a followed by anything

    expect "- gl a"* # same as line above.
- **Mixed matching patterns**

  #using regexp

  #using glob pattern

```
expect {
  –re "a*" {action 1}
      "b*" {action 2}
}
```

- **Using regexp**
  - Backslash any characters that are special to Tcl.

    expect –re "-?\[1-9]\[0-9]*|0x\[0-9a-fA-F]+|0\[0-7]*"

Using Parenthesis in Pattern

- **Parentheses are used to group sub-pattern**
  - Regexp: ab+   matches “ab”, “abb”, “abbb”… not “abab”
  - To match “ababab”, using (ab)+
  - It is dangerous to use (ab)*, since it matches “”. *expect* will not wait for any input and directly terminate.

- **Using parentheses for feedback**
  - When the regexp successfully matches a string, each part of the string that matches a parenthesized sub-pattern is saved in the array expect_out. The first is stored in expect_out(1,string), the second in expect_out(1,string)…
  - Assume input buffer is “*junk abcbcd*” and matching pattern “*a(.*)c*”

    expect_out(0,string) has “abcbc”   #entire matched string

    expect_out(1,string) has “bcb”   #string matches (.*)

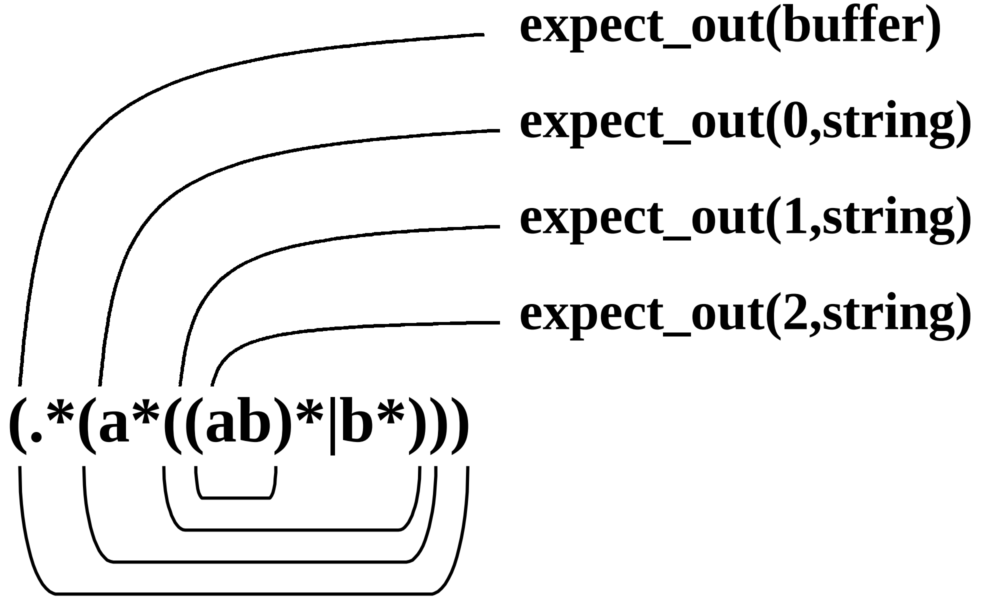
    expect_out(buffer) has “junk abcbc”   #all string buffer read so far.

Pattern Matching Rules of Regexp

- **Rule 1: a regexp matches at the first possible position in the string**
  - Pattern: “a?b”. Input buffer: “ba”.
  - Matching result: “b”
- **Rule 2: the left-most matching branch is used**
  - Pattern: “a|ab”. Input buffer: “ab”.
  - Matching result: “a”
- **Rule 3: the longest match is used**
  - Pattern: “a*b”. Input buffer: “abababa”.
  - Matching result: “ababab”
- **Rule 4: sub-expressions are considered from left to right**
  - Pattern: “a*(b*|(ab)*)”. Input buffer: “aabab”.
  - Matching result: “aab”
  - In above example, is it possible to match the first “a” with the pattern a*, and use sub-pattern “(ab)*” to match remaining “abab”?

Feedback of Nested-Parentheses

- **Determine stored position in the *expect_out* array of sub-pattern**



- **Count the left parentheses to determine feedback position: the sub-expression which starts with the *N*th left parenthesis corresponds with expect_out(*N*, string).**

Backslashes for Regexp

- **Backslash any characters that are special to Tcl when using Regexp.**
- **Examples**

  “$” is special for both Tcl and regexp pattern of expect.

  Assume variable *a* has value “x”

  - expect –re “$a”;   #match x.
  - expect –re “\$a”;   # match string end with literal a.
  - expect –re “\\$a”;   # match literal x.
  - expect –re “\\\$a”;   # match sequence $ followed by a.
  - expect –re “\\\\$a”;   # match sequence \ followed by x.
  - expect –re “\\\\\$a”;   # match sequence \ followed by $ followed by a
  - expect –re “\\\\\\$a”;   # match sequence \ followed by x
  - expect –re “\\\\\\\$a”;   # match ?