Lab1 Design Project

Implementation on While, Until, For, If and ! Commands

Xiaoxuan Wang (804406399)
Qinyi Yan (704406413)

1. Design Objectives

I. Brief Description:

Our program is enabled to distinguish “while”, “until”, “for”, “if” and “!” commands by extracting key tokens, and build the commands into command trees. By using “-p” flag, the command can be printed out in the form of binary tree, with precedence constraints. By executing “./timetrash” without flag, the commands can be executed in serial, meaning without time travelling. Executing “./timetrash” with flag “-t” will execute the commands with time travelling. The commands which have no dependencies on other commands will be executed in parallel. The parallelism of execution is obtained by enabling commands to execute in forked processes.

II. Specifications:

a. The print tree command should be able to print out “while”and “until” commands in the form as in the following example:

```
while
    echo a
      do
        echo b
      done
```

b. The print tree command should be able to print out “if” command in the form as in the following example:

```
if
   echo a
     then
       echo b
     else
       echo c
fi
```

c. The print tree command should be able to print out “for” commands in the form as in the following example:

```
for
    a in 1 2 3
      do
        echo $a
      done
```

d. The print tree command should be able to print out “for” commands in the form as in the following example:

```
!
  (echo a)
```

e. All the commands should be able to be printed when they are in nest relation, say while command nested in if command.

f. All the commands should be able to execute either with parallelism or without, depending on with or without "-t" flag. (Which is not required in the spec, but we also design and complete the execution for these commands in order to make an integral implementation.)
g. Error cases should be detected.

2. Implementing Strategy

I. In read-command.c

1) Enable the read-command.c read "if" command, parse it and print it.
    a) In the case of token type is word, when reading a word "if", build new inner token stream until read "then".
    b) Recursively call the function make_command_tree while passing the new token stream. Making it to a command called condition and add it to this IF_COMMAND u.command[0].
    c) In the same way, build the token stream between "then" to "else" and "else" to "fi", making it as if body command and else body command, separately add them to u.command[1] and u.command[2].
    d) Our implementation well considered about nest. If there is if or while/until/for command is nested in this if command, track the number of nested. Recursively call the make_command_tree until the nest number is zero.
2) Enable the read-command.c read "for/while/until" command, parse it and print it.
    a) While the case of token type is word, if reading a word "until" or "while" or "for", build a new innner token stream until read "do".
    b) Recursively call the function make_command_tree while passing the new token stream. Making it to a command called condition and add it to this WHILE_COMMAND or UNTIL_COMMAND or FOR_COMMAND's u.command[0].
    c) In the same way, build the token stream between "do" to "done", making it as loop body command add it to u.command[1].
    d) Our implementation well considered about nest. If there is if or while/until/for command is nested in this while/if/until command, track the number of nested. Recursively call the make_command_tree until the nest number is zero.
3) Enable the read-command.c read "!" command (NOT_COMMAND), parse it and print it.
    a) While the case of token type is word, if reading a word "! ", build a new inner token stream until next token. NOT_COMMAND only consider the following token.
    b) If the next token type is SUBSHELL or WORD, recursively call make_command_tree and add the follow_cmd to u.command[0]. Otherwise, return error.
4) Add delete_newline function and using the modifided token_stream to make_command_stream.
    a) In order to support newline in one single IF_COMMAND or WHILE_COMMAND/FOR_COMMAND/UNTIL_COMMAND, we build this function to delete newline in this particular situations. The basic idea is delete the HEAD token between tokens since when we have a newline, we build a new token stream which starts with the token type HEAD.
    b) We well considered about the nest. If there exists if or while/until/for command which nest in this command, track the number of nested, which increment by 1 while reading if/do, decrement by 1 while reading fi/done. If the number of nest is not zero, which means the if/fi or do/done is not in pairs.

II. In print-command.c

1) Enable the print-command.c print if/while/for/until command.
    c) In function command_intended_print, print corresponding word - if/until/while/for. Then recursively call command_indented_print which print u.command[0] with indent.
    d) If the command type is IF_COMMAND, print “then”, otherwise, print ”do”. Then recursively call command_indented_print which print u.command[1] with indent..
    e) If the command type if IF_COMMAND, print “else”, then recursively call command_indented_print which print u.command[2] with indent.
    f) Print “fi” or “done”.
2) Enable the print-command.c print not command.
    g) In function command_intended_print, print “!”. Then recursively call command_indented_print which print u.command[0] with indent.

III. In execute-command.c

1) Executing “while”command:

We first execute the while condition, which is command->u.command[0]. By checking the status of u.command[0] after execution, we can tell if the condition is true or not. If the condition is true, execute the u.command[0] which is the loop body then execute u.command[0] again and check the status, otherwise leave while command. The while commands status will be set with the status of the u.command[1].

2) Executing “until” command:

The execution of “until” commands is similar to the execution of while command, except that when only when the condition is false, the loop body, u.command[1] will be executed.

3) Executing “for” command:

The execution of “for” command requires substituting the variable with the instances following keyword “in”. We provided a function “substitute_tokens” to do the job substituting the variable, say $a in u.command[1] with the instances during each loop. After that, by executing u.command[1] until the end of the loop, the results can be obtained.

4) Executing “if” command:

We first execute the if condition, u.command[0]. If the condition is true, the first if body, u.command[1] will be executed. Otherwise, if the if condition is false, the second if body, u.command[2] will be executed. The command’s status will be set correspondingly.

5) Executing “!” commands:

The execution of “!” command first executes u.command[0]. If the result of the execution is true or false (positive numbers or zero), set the command status to its opposite value.

3. Result Display

1) Print command tree and execution without time travelling:
    a. “while”, “until” command:

```
# 1
  while
    echo condition
  do
    echo loopbody
  done
```

```
# 2
  until
    echo condition
  do
    echo loopbody
  done
```

Note: because we are not change the value of the loop condition in the middle of the loop, the loop will be infinite loop if the condition is true, or not entering loop body when

the condition is fault. By executing the above test cases, we got infinite “condition \n loop body” for the first test case, and a “condition” for the second test case.

b. “for” command:

```
# 3
  for
    a in 1 2 3
  do
    echo $a
  done
```

Execution result:

```
[xiaoxuan@lnxsrv01 ~/cs111/Designsubmission]$ ./timetrash for.sh
1
2
3
```

c. “if” command:

```
# 4
  if
    echo a
  then
    echo pass condition
  else
    echo notpass condition
  fi
```

Execution result:

```
[xiaoxuan@lnxsrv01 ~/cs111/Designsubmission]$ ./timetrash if.sh
a
pass condition
```

Note: Since “echo a” is a true statement, the “echo pass condition” command will execute.

d. “!” command:

```
# 2
  !
    (
      echo a \
    ||
      echo b
    )
```

```
# 1
  if
    (
     !
       (
        echo a
       )
    )
  then
    echo pass condition
  else
    echo notpass condition
  fi
```

Execution result:

```
[xiaoxuan@lnxsrv01 ~/cs111/Designsubmission]$ ./timetrash if.sh
a
notpass condition
```

Note: to test "!" command, we put "!(echo a)" in an if command. Theoretically, "!(echo a)" should give us a false value, which leads to the else body, printing out "not pass condition".

e. "if" command nested in "while" loop:

```
# 6
  while
    echo condition
  do
      ehco b \
    ;
      if
        echo if_command
      then
        echo then_command
      else
        echo else_command
      fi
  done
```

f. "while" command nested in "if" command

```
# 7
  if
    echo condition
  then
    until
      echo until_command
    do
      echo a
    done
  else
    echo a
  fi
```

2) Execute commands with time travelling:

```
[xiaoxuan@lnxsrv01 ~/cs111/Designsubmission]$ ./timetrash -t for.sh
1
test timetravel
2
3
4
5
6
7
8
9
10
```

Note: This test case demonstrated that the time travel mode is working correctly since the output order is random. (First print variable a in 1 to 10 then print test time travel)

3) Tested error cases:
   Look up details in "test-design-bad.sh".

4. Demo Note:

1) Print command trees for "while", "until", "for", "if" and "!" commands. The test cases are in "test-design-ok.sh";

2) Execute commands with/ without time travelling for “for”, “if” and “!” commands. Demonstrate infinite loop and not-once cases for “while” and “until” commands;
3) Demonstrate error cases. The error test cases are in “test-design-bad.sh”.