TERM PROJECT

Programming Languages/Survey of Programming Languages.
ITCS 4102/5102

Survey of Erlang Programming Language

Part -2

Common Programs

Team Members

1. Gargi Sarkar - #800822013
2. Naga Bijesh Roy Raya - #800846698
3. Venkata Satya Sai Ram Adusumilli - #800851145

Submitted to:

Dr. Dewan Tanvir Ahmed

dahmed@uncc.edu

Department of Computer Science

University of North Carolina at Charlotte

Charlotte

1. Steps to Compile the Program in Erlang

Step1: Change the Directory where to save the file to be executed using the command **cd("Filepath").**

```
Erlang/OTP 17 [erts-6.1] [64-bit] [smp:2:2] [async-threads:10]

Eshell V6.1  (abort with ^G)
1> cd("D:/Data/Erlang").
D:/Data/Erlang
ok
```

Step2: Compile the Program using the command **c(modulename).**

```
17> c(part2).
part2.erl:117: Warning: variable 'Playerset' is unused
part2.erl:118: Warning: variable 'P2' is unused
part2.erl:125: Warning: variable 'P2' is unused
{ok,part2}
```

2. Solutions to Common Programs in Erlang Programming Language:

Problem #1: Find the Missing Element

There is an array of non-negative integers. A second array is formed by shuffling the elements of the first array and deleting a random element. Given these two arrays, find which element is missing in the second array. [Linear searching is not allowed].

Logic used: An input array (in Erlang List) of integers is taken as input. The array is shuffled using the function nth_rest. The head of the element is removed and then displayed as output.

```
-module(part2).
-export([print_list/]).
print_list(List) ->List,Newlist = list(List),{List,Newlist,hd(List -- Newlist)}.
list([])    -> [];
list([Elem]) -> [Elem];
list(List)   ->    tl(list(List, length(List), [])).
list([], 0, Result) ->
   Result;
list(List, Len, Result) ->
```

```
    {Elem, Rest} = nth_rest(random:uniform(Len), List),
    list(Rest, Len - 1, [Elem|Result]).

nth_rest(N, List) -> nth_rest(N, List, []).

nth_rest(1, [E|List], Prefix) -> {E, lists:append(Prefix,List)};
nth_rest(N, [E|List], Prefix) -> nth_rest(N - 1, List, [E|Prefix]).
```

Input & Output:

```
18> part2:print_list([1,3,2,4,5,6,7,8,10]).
Input 1:[1,3,2,4,5,6,7,8,10]

Input 2:[7,2,1,4,3,6,10,8]

Output: 5
ok
```

Problem #2: Check Balanced Parentheses

Given a string of opening and closing parentheses, check whether it's balanced. We have 3 types of parentheses: round brackets: (), square brackets: [], and curly brackets: {}. Assume that the string doesn't contain any other character than these, no spaces words or numbers. Just to remind, balanced parentheses require every opening parenthesis to be closed in the reverse order opened.

Logic used: The solution for the above problem is implementation of stack on the set of parenthesis. When a opening parenthesis is encountered, push that opening parenthesis into the stack. When its immediate closing parenthesis is encountered, pop the opening parenthesis from the stack. If the list is empty, the output will be "YES", otherwise "NO".

```
-module(part2).
-export([checkinput/1]).

checkinput(Pattern) ->
        check(match(Pattern,[])).

match(Pattern, NewPattern) ->
        if
                hd(Pattern)=:=123 ->
                        match(tl(Pattern),lists:append(NewPattern,[123]));
```

```
                hd(Pattern)=:=40 ->
                        match(tl(Pattern),lists:append(NewPattern,[40]));
                hd(Pattern)=:=91 ->
                        match(tl(Pattern),lists:append(NewPattern,[91]));
                hd(Pattern)=:=125 ->
                        case lists:last(NewPattern) =:= 123 of
                                true ->
                                        match(tl(Pattern),lists:droplast(NewPattern));
                                false ->
                                        match([],NewPattern)
                        end;
                hd(Pattern)=:=41 ->
                        case lists:last(NewPattern) =:= 40 of
                                true ->
                                        match(tl(Pattern),lists:droplast(NewPattern));
                                false ->
                                        match([],NewPattern)
                        end;
                hd(Pattern)=:=93 ->
                        case lists:last(NewPattern) =:= 91 of
                                true ->
                                        match(tl(Pattern),lists:droplast(NewPattern));
                                false ->
                                        match([],NewPattern)
                        end;
                true ->
                        NewPattern
        end.

check(Input)->
        case length(Input) =:= 0 of
                true -> "YES";
                false -> "NO"
        end.
```

Input & Output:

```
35> part2:checkinput("{{{([])}}}").
"YES"
36> part2:checkinput("{{{([(])){}}}}").
"NO"
```

Problem #3: Permutations
Generate all permutations of a given string.

Logic used: Permutations of a String(in Erlang list of characters) can be calculated recursively by interchanging the positions of each and every element of the character list.

```
-module(part2).
-export([permutate/1]).

permutate(L) -> perms(atom_to_list(L)).
perms([]) -> [[]];
perms(L) -> [[H|T] || H <- L, T <- perms(L--[H])].
```

Input & Output:

```
25> part2:permutate('bijh').
["bijh","bihj","bjih","bjhi","bhij","bhji","ibjh","ibhj",
 "ijbh","ijhb","ihbj","ihjb","jbih","jbhi","jibh","jihb",
 "jhbi","jhib","hbij","hbji","hibj","hijb","hjbi","hjib"]
```

Problem #4: File I/O
You will read a text file. Then create another text file containing only the number of words in the first file.

Logic used: Erlang supports reading the contents of a file character by character till the pointer reaches the eof. The count of the words can be calculated by adding the number of spaces and the new line characters of the file. The result thus obtained will be written to a new file.

Note: In order to see the written file, change the file path in writetoafile(Len) to your desired file location.

```
-module(part2).
-export([readlines/1]).

readlines(FileName)->
        Newlist1 = readline(FileName),
        Length1= length(Newlist1),
```

```
        Length2= length(writefile(Newlist1)),
        Newlist2 = string:tokens(Newlist1, "\n"),
        Length3 = length(Newlist2),
        io:format("Count of Words are written to the file successfully.\n"),Len=(Length1-
        Length2)+Length3,writetoafile(Len).
readline(FileName) ->
  {ok, Device} = file:open(FileName, [read]),
  try get_all_lines(Device)
   after file:close(Device)
  end.

get_all_lines(Device) ->
  case io:get_line(Device, "") of
    eof  -> [];
    Line -> Line ++ get_all_lines(Device)
  end.
writefile(Newlist1) ->
        [X || X <- Newlist1, X /= 32].

writetoafile(Len)->
        {ok, IODevice} = file:open("D:/Data/Erlang/output.txt", [write]), file:write(IODevice,
lists:flatten(io_lib:format("~p", [Len]))), file:close(IODevice).
```

Input & Output:

```
40> part2:readlines("D:/Data/Erlang/sample.txt").
Count of Words are written to the file successfully.
ok
```

Problem #5: Three-of-a-crime

Three-of-a-crime is a simple logic game for up to 3 players. There are 7 different criminals. The computer randomly chooses three of these (the "perpetrators"), but doesn't tell the players which are chosen. The computer then puts down three random criminals. 0, 1, or 2 of these may be the actual perpetrators. The computer also tells the player how many (but not which) of the three criminals are perpetrators. The players may either guess which three criminals are the actual perpetrators or they may pass. If a player guesses wrong, she is out of the game and the other players continue. If no player chooses to guess, the computer puts down another three randomly chosen criminals (0, 1, or 2 of which may be actual perpetrators) and tells the players how many (but not which) of these are actual perpetrators. Players can again use logic to deduce the three actual criminals and may

guess. Play continues until some player guesses correctly or until all players have guessed incorrectly.

Logic used: The logic used to build the following Game is clearly given in the problem statement itself.

```
-module(part2).
-export([playGame/0]).

playGame()->Gameset=['a','b','c','d','e','f','g'],
                SplitGameset=playGameset(Gameset),
                {ActualGameset,Playerset}=lists:split(3,SplitGameset),
                startGame(ActualGameset,Playerset).
playGameset([])   -> [];
playGameset([Elem]) -> [Elem];
playGameset(Gameset)   ->   playGameset(Gameset, length(Gameset), []).
playGameset([], 0, Result) ->    Result;
playGameset(Gameset, Len, Result) ->
          {Elem, Rest} = n_th_rest(random:uniform(Len), Gameset),
          playGameset(Rest, Len - 1, [Elem|Result]).
n_th_rest(N, Gameset) -> n_th_rest(N, Gameset, []).
n_th_rest(1, [E|Gameset], Prefix) -> {E, lists:append(Prefix,Gameset)};
n_th_rest(N, [E|Gameset], Prefix) -> n_th_rest(N - 1, Gameset, [E|Prefix]).

startGame(ActualGameset,Playerset)->startGame(ActualGameset,Playerset,0,1).
startGame(ActualGameset,Playerset,3,1)->
                io:format("Players lost the game in first attempt. Now try again!!\n"),
                startGame(ActualGameset,Playerset,0,2);
startGame(ActualGameset,Playerset,3,2)->
                io:format("Players lost the game. Actual Perpetrators are
~p.",[ActualGameset]);
startGame(ActualGameset,Playerset,Count,1)->
                {S1,S2}=lists:split(1,ActualGameset),
                {P1,P2}=lists:split(2,Playerset),
                Results=lists:append(S1,P1),
                io:format("Perpetrators set is ~p of which one Perpetrator is
correct\n",[Results]),

                {ok, [AX]} = io:fread("Enter your first actual Perpetrator you guess:", "~c"),
                {ok, [BX]} = io:fread("Enter your second Actual Perpetrator you Guess :", "~c"),
                Result=lists:append(AX,BX),
                        case Result =:= S2 of
```

```
                    true -> startGame(ActualGameset);
                    false ->startGame(ActualGameset,Playerset,(Count+1),1)
            end;
startGame(ActualGameset,Playerset,Count,2)->
        {S1,S2}=lists:split(2,ActualGameset),
        {P1,P2}=lists:split(1,Playerset),
        Results2=lists:append(S1,P1),
        io:format("Perpetrators set is ~p of which two Perpetrators are correct \n",[Results2]),
        {ok, [X]} = io:fread("Enter your Guess now:", "~c"),
            case list_to_atom(X) =:= hd(S2) of
                    true -> startGame(ActualGameset);
                    false ->startGame(ActualGameset,Playerset,(Count+1),2)
            end.
startGame(ActualGameset)->
        io:format("Player won the Game, Actual Perpetrators are: ~p",[ActualGameset]).
```

Input & Output:

```
41> part2:playGame().
Perpetrators set is [g,d,f] of which one Perpetrator is correct
Enter your first actual Perpetrator you guess:e
Enter your second Actual Perpetrator you Guess :b
Perpetrators set is [g,d,f] of which one Perpetrator is correct
Enter your first actual Perpetrator you guess:e
Enter your second Actual Perpetrator you Guess :c
Perpetrators set is [g,d,f] of which one Perpetrator is correct
Enter your first actual Perpetrator you guess:c
Enter your second Actual Perpetrator you Guess :a
Players lost the game in first attempt. Now try again!!
Perpetrators set is [g,b,d] of which two Perpetrators are correct
Enter your Guess now:e
Perpetrators set is [g,b,d] of which two Perpetrators are correct
Enter your Guess now:a
Player won the Game, Actual Perpetrators are: [g,b,a]ok
```