Lista de Exercícios de Linguagens de Programação II -- 2015, 2o período
Universidade Federal do Amazonas / Instituto de Computação
Marco Cristo

Alunos: Gercidara da Silva Lira - 21454569
Hugo Conrado de Carvalho - 21456635

Funcional

1) Defina os seguintes conceitos como são compreendidos no paradigma funcional: **(a) funções anônimas: As funções anônimas são conhecidas também como "expressões lambda", é mais um recurso para deixar a sintaxe mais elegante, elas são bastantes utéis quando se precisa passar um função como parâmetro para uma outra função que não seja mais necessária após. São criadas normalmente porque a complexidade é pouca para usar um def. (b) funções de primeira classe: Também são conhecidos como objetos de segunda classe, é caracterizado o objeto de segunda classe quando ele pode ser construído em tempo de execução passado como parâmetro e devolvido como resultado de uma função. Números e strings são objetos de primeira classe.(c) funções de alta ordem: São funções que recebem outras funções como argumento e devolvem as mesmas como resultado,essas funções só podem existir em funções que são objetos de primeira classe, ela sempre recebe ou retorna uma função, seus principais exemplos é o mapear, filtrar e reduzir(map,filter,foldr1). , (d) funções recursivas: É uma função que se refere a si mesma, ela consiste em um caso base,onde já se sabe o resultado e um sub problema do problema abordado inicialmente o erro mais esperado da função recursiva é quando ela não para,acabando com a memória. São bastante utilizadas em problemas com definição matemáticas um dos maiores exemplos dessa função é o fatorial. , (e) funções puras: A principal ideia dessas funções é que elas podem ser substituídas pelo seu valor de retorno, programadores costumam dizer que essa função não possuí efeito colateral, pois não altera nenhuma variável ou dado fora da função. Além de consistente, como já foi dito, se der um valor de entrada, ela retornará o mesmo na saída. , (f) avaliação preguiçosa: É uma técnica utilizada para atrasar a computação até certo ponto em que o resultado seja suficiente, aumenta desempenhos e evita cálculos desnecessários, evitando erros nas avaliações, define melhor funções regulares. , (g) iteradores: Sequencia de elementos geradas por meio de avaliação preguiçosa. (h) geradores ou *streams*: Expressões ou funções que criam iteratores.**

2) Usando LCs, defina a função *divisor_proprio*(*n*) que retorna a lista dos números entre 1 e *n*/2 (incluso) que dividem *n* exatamente. Ex: `divisor_proprio(10) = [1, 2, 5]`.

```
Divisor_Proprio = lambda n: [i for i in range(1,n) if n%i==0]
print(Divisor_Proprio(10))
```

3) Defina a função *amigos*(*n1, n2*) que retorna True se n1 e n2 são números amigos, ou seja, n1 é a soma dos divisores próprios de n2 e vice-versa. Ex: `amigos(220, 284) = True; amigos(10, 11) = False`.

```
amigos = lambda n1,n2: True if sum([i for i in range(1,n1) if n1%i==0]) == n2 or
sum([i for i  in range(1,n2) if n2%i==0])==n1 else False
print(amigos(10,11))
```

4) Implemente a função *duas_vezes* que tem como parâmetros uma função *f* e um número *n* e retorna uma função que corresponde à dupla aplicação de *f* a *n*. Ex: `duas_vezes(lambda n: n/2, 4) = 1`.

```
duas_vezes = lambda f,n: f(f(n))
print(duas_vezes(lambda n: n/2, 4))
```

5) Implemente a função *composição* que tem como parâmetros uma lista de funções e um número e retorna o valor resultante da composição das funções aplicadas ao número. Ex: `composicao([], 4) = 4; composição([lambda n:n+2, lambda n:n*n], 3) = 25`. Dica: atenção para a ordem -- composição([f, g, h], n) = h(g(f(n))).

```
composicao = lambda fl,n: sum ([fl[i+1](fl[i](n)) for i in range(len(fl)-1)])
print(composicao([lambda n:n+2, lambda n:n*n], 3))
```

6) Usando LCs, implemente a função soma_*colunas*(*m*) que, dada uma matriz *m* em forma de lista de listas, retorna um vetor em que cada elemento corresponde à soma de uma das colunas de *m*. Ex: `soma_colunas([[1, 2], [3, 4], [5, 6]]) = [9, 12]`.

```
soma_coluna = lambda m: [reduce(lambda x, y: x + y, [i[j] for i in m]) for j in range(len(m[0]))]
m = [[1,2,3],[3,4,5],[5,6,7]]
print soma_coluna(m)
```

7) Usando LCs, escreva a função *conte_filtros*(*lf*, *la*) que retorne uma lista inteira onde o i-ésimo elemento corresponde ao número de elementos da lista de funções *lf* (os filtros) que retornam *True* para o i-ésimo elemento na lista de argumentos *la*. Ex: `conte_filtros([lambda n:n%2==0, lambda n:n%3==0], range(10)) = [2, 0, 1, 1, 1, 0, 2, 0, 1, 1]`. Neste exemplo, a lista de funções é formada pelos filtros para verificar par e múltiplo de 3. A lista de argumentos são os número de 0 a 9. A lista de resposta indica que 0 é par e múltiplo de 3, que 1 não é par nem múltiplo de 3, que 2 é ou par ou múltiplo de 3, etc. Dica: em Python, quando usados aritmeticamente, True = 1 e False = 0. Por exemplo, True + True = 2.

```
conte_filtro = lambda lf, la :[[sum([lf[j](la[i]) for j in range(len(lf))]) for i in range(len(la))]]
print (conte_filtro([lambda n:n%2==0,lambda n:n%3==0],range(10)))
```

8) Usando recursão, escreva uma função *map_elementos*(*f*, *lst*), que aplica a função *f* a cada elemento da lista *lst*. Se um dos elementos de *lst* for uma lista, *f* é aplicada a cada elemento desta lista e assim sucessivamente. Ex: `map_elementos(lambda n: 2*n, [1, [7, 3, [4, 5], 8], 7]) = [2, [14, 6, [8, 10], 16], 14]`. Dica: use a função type para saber o tipo de um objeto. Por exemplo `type([1,2]) is list = True`.

```
def map_elementos(f,lst):
  for i in range (len(lst)):
    if type (lst[i])==list:
      map_elementos(f,lst[i])
    else:
      lst[i]= f(lst[i])
  return (lst)
f= lambda n: 2*n
lst= [1, [7, 3, [4, 5], 8],7]
print (map_elementos(f,lst))
```

9) Resolva os seguintes problemas em Python, relacionados com o jogo de *Das Bohnenspiel*, usando o paradigma funcional

O jogo: *Das Bohnenspiel* (o Jogo do Feijao) é jogado em um tabuleiro com duas linhas de seis buracos. O jogo começa com seis feijões em cada buraco. Cada jogador é dono dos seis buracos do seu lado. O jogador da vez escolhe qualquer um dos buracos no seu lado que tenha pelo menos um feijão dentro. Ele então remove todos os feijões deste buraco e os 'semeia' em sentido anti-horário. Semear consiste em remover os feijões do buraco selecionado e jogá-los nos buracos subsequentes, um a um, até que todos os feijões tenham sido usados (exceto o buraco original. No caso da semeadura alcança-lo, ele é saltado de forma a permanecer sempre vazio). Se o último feijão cai em um buraco que, depois de semeado, tem *dois*, *quatro* ou *seis* feijões, todos os feijões neste buraco são capturados. Se uma captura é feita, o buraco anterior também é verificado (e seus feijões possivelmente capturados) de acordo com a mesma regra e assim por diante. Se o jogador da vez não pode se mover (pois todos os seus buracos estão vazios), o jogo termina e todos os feijões no tabuleiro vão para o outro jogador. O objetivo do jogo é capturar mais feijões que o adversário.

a) Dada a lista *pits* que indica os 12 buracos do tabuleiro e o identificador *player_id* do jogador (0 ou 1), crie a função `playable_pits(pits, player_id)` que retorna a lista das *n* posições que o jogador *player_id* pode escolher para jogar. Ex: `playable_pits([0, 2, 3, 2, 0, 0, 4, 2, 1, 1, 1, 0], 0) = [1, 2, 3], playable_pits([0, 2, 3, 2, 0, 0, 4, 2, 1, 1, 1, 0], 1) = [6, 7, 8, 9, 10];`

```
def playable_pits(pits, p_id):
    return [i for i in range(p_id * 6, (p_id + 1) * 6) if pits[i] > 0]
```

b) Dada a lista *pits* que indica os 12 buracos do tabuleiro e o identificador *player_id* do jogador (0 ou 1), crie a função `random_pit(pits, player_id)` que retorna um buraco aleatório entre os que o jogador pode escolher. Se nenhum buraco pode ser retornado, a função retorna -1. Exs: `random_pit([0, 2, 3, 2, 0, 0, 4, 2, 1, 1, 1, 0], 0) = 2; random_pit([0, 2, 3, 2, 0, 0, 4, 2, 1, 1, 1, 0], 1) = 7; random_pit([0, 0, 0, 0, 0, 0, 4, 2, 1, 1, 1, 0], 0) = -1`; Dica: utilize a função que você criou em (a) e *randint*, do pacote *random*.

```
def random_pit(pits, p_id):
    candidatos = playable_pits(pits, p_id)
    return candidatos[randint(0, len(candidatos)-1)] if len(candidatos) > 0 else -1
```

c) Crie a função `circular_rdistance(size, pos1, pos2)` que, dado o tamanho *size* de uma lista e duas posições na lista, pos1 e pos2, retorne a distância (para a direita) entre *pos1* e *pos2* considerando que a lista é circular. `Ex: circular_rdistance(12, 0, 11) = 11; circular_rdistance(12, 11, 0) = 1.`

```
def circular_rdistance(t, p1, p2):
    return t - p1 + p2 if p1 > p2 else p2 - p1
```

d) Usando LCs, crie uma função `sow(pits, pit)` que, dado uma lista de buracos *pits* e um buraco nesta lista, *pit*, ela retorna a lista de buracos resultante do processo de semeadura. Dica: note que para uma lista *pits* de tamanho *t* e um total de *n* feijões em *pit*, a lista

resultante (1) terá 0 feijões na posição *pit*; (2) no mínimo, *n* / (*t* - 1) feijões nos demais buracos e (3) mais 1 feijão nos buracos que estão a uma distância de *n* % (*t* - 1) do buraco *pit*. Ex: `sow([6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6], 7) = [7, 7, 6, 6, 6, 6, 6, 0, 7, 7, 7, 7]`; `sow([5, 9, 5, 5, 5, 5, 5, 5, 5, 5], 1) = [6, 0, 6, 6, 6, 6, 6, 6, 6, 6]`; `sow([5, 10, 5, 5, 5, 5, 5, 5, 5, 5], 1) = [6, 0, 7, 6, 6, 6, 6, 6, 6, 6]`; `sow([5, 20, 5, 5, 5, 5, 5, 5, 5, 5], 1) = [7 0 8 8 7 7 7 7 7 7]`; `sow([5, 18, 5, 5, 5, 5, 5, 5, 5, 5], 1) = [7 0 7 7 7 7 7 7 7 7]`. Obs: nestes exemplos, o número no buraco referenciado por *pit* é mostrado em negrito.

```
def sow(pits, target):
    return [(pits[i] + (pits[target] // 11) + (0 if pits[target] % 11 < circular_rdistance(12, target, i) else 1)) if i != target else 0 for i in range(0, 12)]
```

e) Dado o tamanho *t* de uma lista e uma posição *pos* nesta lista, escreva a função `cprevious(t, pos)` que retorna a posição anterior a *pos* em uma lista circular de tamanho *t*. Exs: `cprevious(4, 1) = 0`; `cprevious(4, 0) = 3`.

```
def cprevious(t, pos):
    return pos - 1 if pos != 0 else t - 1
```

f) Escreva a função `last_sown_pit(pits, pit)` que, dada a lista pits e a posição *pit*, ela retorna o último buraco semeado ao fim do processo de semeadura. Exs: ao fazer a semeadura dos *pits* [6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6], usando *sow* na posição 7, obtemos `sow([6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6], 7) = [7, 7, 6, 6, 6, 6, 6, 0, 7, 7, 7, 7]`; Logo, `last_sown_pit([6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6], 7) = 1`, pois a posição 1 foi a última a ser semeada. Dica: *pit* nunca pode ser o último semeado.

```
def last_sown_pit(pits, target):
    return ((target + (pits[target] % 11)) % 11) - (1 if ((target + (pits[target] % 11)) % 11) <= target else 0)
```

g) Usando LCs, escreva a função `pits_with_zero_at(pits, pos)` que retorna a lista *pits* com o valor da posição *pos* igual a zero. Exs: `pits_with_zero_at([6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6], 7) = [6, 6, 6, 6, 6, 6, 6, 0, 6, 6, 6, 6]`;

```
def pits_with_zero_at(pits, target):
    return [pits[i] if target != i else 0 for i in range(0,12)]
```

h) Escreva a função *recursiva* `sow_points(pits, pos)` que retorna uma tupla com (a) a nova lista de feijões e (b) o número de feijões capturados (pontos obtidos) a partir da posição *pos* depois do processo de semeadura na lista de buracos *pits*. Exs: `sow_points([6, 6, 6, 0, 7, 7, 7, 7, 7, 7, 6, 6], 9) = ([6, 6, 6, 0, 7, 7, 7, 7, 7, 7, 6, 6], 0)`; `sow_points([6, 6, 6, 0, 7, 7, 7, 7, 7, 6, 6], 10) = ([6, 6, 6, 0, 7, 7, 7, 7, 7, 0, 6], 6)`; `sow_points([6, 6, 6, 0, 7, 7, 7, 7, 7, 6, 6], 11) = ([6, 6, 6, 0, 7, 7, 7, 7, 7, 7, 0, 0], 12)`; `sow_points([6, 6, 6, 0, 7, 7, 7, 7, 7, 6, 6], 0) = ([0, 6, 6, 0, 7, 7, 7, 7, 7, 7, 0, 0], 18)`. Dica: *pits_with_zero_at* e *cprevious* podem ser úteis.

```
def sow_points_recursive(pits, pos, points):
    return (pits, points) if (pits[pos] not in [2, 4, 6]) else (sow_points_recursive(pits_with_zero_at(pits, pos), cprevious(12, pos), pits[pos] + points))

def sow_points(pits, pos):
    return (pits, 0) if (pits[pos] not in [2, 4, 6]) else (sow_points_recursive(pits_with_zero_at(pits, pos), cprevious(12, pos), pits[pos]))
```

Usando a função *sow_points*, é possível criar a função *play* que, dado o identificador de um jogador *p_id* e sua função de escolha de buracos *choose_id*, ela obtém o buraco a semear, determina a nova configuração dos buracos após a semeadura e calcula o número de pontos correspondentes:

```
# choose a pit, sow the beans and calculate the points for this move
# returns tuple (pits, points of player 0, points of player 1)
def play(p_id, choose_pit, pits, p0, p1):
    pit = choose_pit(pits, p_id)
    if pit == -1:
       # if no move is possible, all the beans are given to the opponent
       points = sum(pits)
    else:
       pits, points = sow_points(sow(pits, pit), last_sown_pit(pits, pit))
    return (pits, p0 + points, p1) if p_id == 0 else (pits, p0, p1 + points)
```

i) Escreva as funções (a) `end_game(p0, p1)` que, dados os números de pontos feitos pelos jogadores 0 (*p0*) e 1 (*p1*), determina se o jogo terminou ou não e (b) `report_result(p0, p1)` que, dados os números de pontos feitos pelos jogadores 0 (*p0*) e 1 (*p1*), exibe as mensagens "Jogador 0 venceu", "Jogador 1 venceu" ou "Empate". Dica: como há no máximo 12 * 6 = 72 feijões, quem alcançar 37 primeiro, ganha.

```
def end_game(p0, p1):
        return (p0 > 36 or p1 > 36)

def report_result(p0, p1):
        print(("Jogador 0 venceu" if p0 > 36 else ("Jogador 1 venceu" if p1 > 36 else "Empate")))
        return 1 if p0 > 36 else 0
```

Uma vez que você escrever as funções `end_game()` e `report_result()` pode usar as funções abaixo para simular um jogo entre dois jogadores aleatórios:

```
# game seen as a recursive change of turns
# -- player 0 plays in even turns while player 1 in odd turns
def match_r(turn, players, pits, p0, p1):
    pits, p0, p1 = play(turn % 2, players[turn % 2], pits, p0, p1)
    print turn, turn%2, ':', [p0, p1, pits]
    if end_game(p0, p1):
       report_result(p0, p1)
    else:
       match_r(turn + 1, players, pits, p0, p1)

# start game between player0 and player1
def match(choose_pit0, choose_pit1):
    # start state with 6 beans in each pit
    pits, p0, p1 = [6 for i in range(12)], 0, 0
    print 'inicio: ', [p0, p1, pits]
    # start recursion at turn 0
    match_r(0, [choose_pit0, choose_pit1], pits, p0, p1)

if __name__ == '__main__':
    match(random_pit, random_pit)
```

j) Escreva a função `heuristic_pit(pits, player_id)` que, dentre os buracos possíveis para serem escolhidos, escolhe um movimento não aleatório, com base em alguma heurística que você criar. Por favor, explique claramente a sua heurística nos comentários do código. Para ela ser considerada correta, é necessário que ela vença um jogador aleatório um

número razoavelmente convincente de vezes (por exemplo, 75% das vezes). Dica: pense em uma estratégia gulosa que procure maximizar a pontuação na rodada.

```
def heuristic_pit(pits, p_id):
        prio = sorted([(pos, sow_points(pits, pos)[1]) for pos in playable_pits(pits, p_id) if sow_points(pits, pos)[1] > 0], key = lambda p: p[1])
        return random_pit(pits, p_id) if len(prio) <= 0 else prio[-1][0]
```

A função heuristica calcula a quantidade de pontos aferida para cada jogada possivel naquele turno, e coloca essas jogadas numa lista, que é ordenada de acordo com a pontuação a ser obtida. Então, joga-se na melhor jogada daquele turno. Quando não há jogadas que pontuam, joga-se aleatoriamente. Caso não haja jogadas possíveis, retorna-se -1, conforme estabelecido para a função play. Para mais detalhes, o código do jogo completo possui mais comentários sobre a forma de execução.