Lista de Exercícios de Linguagens de Programação II
Universidade Federal do Amazonas
Instituto de Computação/Ciência da Computação
Marco Cristo

Teoria sobre Paradigma Funcional

1) Defina funções anônimas, avaliação preguiçosa, recursão, funções puras, efeito colateral, iteradores e geradores.

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 uma 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.

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.

Recursão: 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 matemática um dos maiores exemplos dessa função é o fatorial.

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.

Efeito Colateral: Na presença de efeitos colaterais, o comportamento de uma função pode depender da sua ordem de execução no programa, uma vez que alguma outra expressão ou função pode modificar o valor de alguma variável que está faz uso.

Iteradores: Sequência de elementos geradas por meio de avaliação preguiçosa.

Geradores: Expressões ou funções que criam iteratores.

2) Por que programas funcionais são mais fáceis de paralelizar?

Mostra a aplicação de funções, em contraste de programação imperativa, que mostra mudanças no estado do programa. E identificação da maior parte do programa responsável pelo uso processador, ou seja, onde a maior parte está sendo feita e estes pontos de demandas de processamento são as principais partes que devem ser parelizadas, já que grande parte da execução do programa.

Programação Funcional

3) Usando recursão, crie a função *tem_duplicatas*(*lst*) que, dada a lista *lst*, ela retorna *True* se, ao menos um elemento de *lst* ocorre mais de uma vez. Por exemplo, tem_duplicatas([1,2,3]) = False; tem_duplicatas([1,2,2,3]) = True.

4) Usando recursão, crie a função *remove_duplicatas*(*lst*) que, dada a listas *lst*, ela retorna *lst* sem elementos duplicados. Por exemplo, remove_duplicatas([1, 2, 2, 3]) = [1, 2, 3].

```
1 = [1, 2, 4, 5, 6, 1, 3, 4, 5, 6]
Def unique ( a):
    Return list(set(a))
```

5) Uma tríade de Pitágoras é uma tripla (*a*, *b*, *c*) tal que $a^2 + b^2 = c^2$. Usando LCs, crie a função *triades*(*n*, *m*) que retorne todas as tríades para valores de *a*, *b* e *c* entre *n* e *m* (inclusive). Por exemplo, triades(1,10) = [(3, 4, 5), (4, 3, 5), (6, 8, 10), (8, 6, 10)].
def triades(n,m):

```
while c in range (n,m+1):

for b in range (n,m+1):

while a in range(n,m+1):

if ((x*x) + (y*y)) == z*z:

print (x, y, z)
```

6) Reescreva a seguinte função usando funções *map* e *filter*: x2 = lambda l: x + 2 for x in l if x > 3].

7) Escreva a função *mapseq*(*fs*, *arg*) que, dada uma lista de funções *fs* e um argumento inicial *arg*, ela mapeia as funções em *fs* sucessivamente para *arg*. Ou seja *map*([*f*, *g*, *h*], *x*) é equivalente a *f*(*g*(*h*(*x*))). Por exemplo, mapseq([lambda l: [x*x for x in l], sum, sqrt], range(5)) = 5.4772255750516612.

8) Escreva a função *repeatf*(n, *fs*, *arg*) que, dado um número de iterações *n*, uma lista de funções *fs* e um argumento inicial *arg*, ela mapeia as funções

em *fs* sucessivamente para *arg*, *n* vezes. Por exemplo, seja *printf* definida como:

```
def printf(arg):
    print arg
    return arg
```

Então, repeatf(4, [printf, lambda n: n-1], 5) deve imprimir os números 5, 4, 3, 2 e então retornar o valor 1.

Desafio Prático: um interpretador uScheme

Scheme é um dialeto do LISP projetado para ser simples e elegante. A sintaxe de Scheme é diferente da maioria das linguagens que vocês têm usado. Vamos tomar Java como referência. Java tem muitas convenções sintáticas (palavras-chave, vários operadores, parênteses, chaves, vários tipos de precedência, notação de pontos, aspas, virgulas, ponto-e-vírgula, etc) enquanto Scheme é *muito* mais simples. Abaixo, temos um exemplo de código em Java e Scheme:

Java	Scheme

if (x.val() > 0) {
  return fn(A[i] + 1,
  new String[] {"one", "two"});
}
	(if (> (val x) 0)
	  (fn (+ (aref A i) 1)
	  (quote (one two)))

Um programa em Scheme é formado apenas por expressões. Números e símbolos são expressões atômicas. Todo o resto são expressões em listas -- um “(” seguido por zero ou mais expressões seguido de um “)”. O primeiro elemento da lista, determina a sua semântica. Por exemplo, a expressão em lista (+ 1 2) indica a soma (+) dos elementos 1 e 2. O valor de (+ 1 2) é, portanto, 3. Se o primeiro elemento foi uma palavra não reservada, a expressão corresponde a uma chamada de função, ex: (fn 1 2) corresponde a chamar a função *fn* com parâmetros 1 e 2.

Neste exercício, você deve construir um interpretador para um pequeno dialeto de Scheme, chamado uScheme, através de várias tarefas intermediárias.

PARTE I – uScheme como uma calculadora

Um interpretador é formado basicamente por um *parser* e um de *avaliador*. O *parser* pega o programa de entrada como uma string, verifica se ele está

de acordo com as regras da linguagem e o traduz para uma representação interna como, por exemplo, uma árvore sintática abstrata. O *avaliador* processa a representação interna de acordo com as regras semânticas da linguagem, avaliando assim a expressão. A seguir temos um exemplo desses módulos (*parse* e *eval*) processando o programa (begin (define r 10) (* 3.14 (* r r))):

```
> parse("(begin (define r 10) (* 3.14 (* r r)))")
  ['begin', ['define', 'r', 10], ['*', 3.14, ['*', 'r', 'r']]]
> eval(parse(program))
  314.1592653589793
```

O *parser*, normalmente, faz a análise léxica (string de caracteres é separada em *tokens*) e a análise sintática (em que os *tokens* são organizados em uma árvore sintática abstrata). Os *tokens* em uScheme são parênteses, símbolos e números. Vamos começar implementando o analisador léxico.

9) Usando recursão, escreva a função get_token(*lst*) que, dada uma lista de caracteres *lst*, retorna um par formado com o primeiro token em *lst*, seguido do restante de *lst*. Por exemplo, get_token([c for c in '(sqrt 3)']) = ('(', ['s', 'q', 'r', 't', ' ', '3', ')']); get_token([c for c in 'sqrt 3)']) = ('sqrt', ['3', ')']).

10) Usando recursão e a função get_token(*lst*), escreva a função *tokenize(str)* que, dada uma string *str*, retorna a lista de tokens em *str*. Por exemplo, tokenize('(eq2grau -1 (fatorial 4) 3)') = ['(', 'eq2grau', '-1', '(', 'fatorial', '4', ')', '3', ')']; tokenize('(3 4)') = tokenize('(3 4)') = tokenize('(3 4)') = ['(', '3', '4', ')'] .

O analisador sintático deve então pegar a lista de *tokens* e traduzi-la para uma árvore sintática abstrata. Esta árvore pode ser representada por uma lista como em get_semantic_tree(['(', 'eq2grau', '-1', '(', 'fatorial', '4', ')', '3', ')']) = ['eq2grau', '-1', ['fatorial', '4'], '3']. A seguir, temos uma implementação recursiva bem simples de um analisador sintático. Para facilitar a compreensão, não há nenhuma verificação de erro.

```
def get_semantic_tree_r(input, L):
    if len(input) == 0:
        return L[0]
    token = input.pop(0)    # consome um token de L
    if token == '(':
        return get_semantic_tree_r(input, L + [get_semantic_tree_r(input, [])])
    elif token == ')':
```

```
        return L
    else:
        return get_semantic_tree_r(input, L + [getatom(token)])

def get_semantic_tree(input):
    return [] if not input else return get_semantic_tree_r(tokenize(input), [])

def getatom(token):
    """ atomos podem ser inteiros, reais ou strings apenas """
    try:
        return int(token)
    except ValueError:
        try:
            return float(token)
        except ValueError:
            return token
```

11) No analisador sintático, descrito anteriormente, e especificamente na função *get_semantic_tree_r*, há um padrão de programação não recomendado, do ponto de vista de programação funcional. Qual é e por que é não recomendado?

Nossa primeira versão de uScheme é restrita a formas sintáticas que vão nos permitir usá-la como uma calculadora de notação prefixada, com suporte para a manipulação de listas e definição de variáveis. A tabela abaixo apresenta uma especificação *informal* do que desejamos interpretar.

Expressão	**Sintaxe**	**Semântica**
Referência a variável	var	Símbolo interpretado como nome de variável. Seu valor é o da variável. Ex: x
Literal	numero	Valor numérico (inteiro ou ponto flutuante) Ex: 5
Definição	(define var exp)	Define variável e atribui valor corresponde à avaliação de exp Ex: (define x 5)
Chamada de Procedimento	(proc arg ...)	Se *proc* é qualquer coisa que não *define*, ele é uma função. Após avaliar *proc* e seus argumentos, a função proc é aplicada ao valor dos argumentos. Ex: (+ (* 4 5) (+ 2 1))

Para suportar a definição dada, e considerando as funções já construídas para o *parser*, nós agora implementamos o avaliador. Para isso, é necessário antes definir um ambiente de avaliação:

```
def create_env():
   # ambiente suporta operadores +, -, *, /, além de funções
   # padrão car, cdr, list e length
   dic = {}
   dic['+'] = lambda x,y: x+y
   dic['-'] = lambda x,y: x-y
   dic['*'] = lambda x,y: x*y
   dic['/'] = lambda x,y: x/y
   dic['car'] = lambda x: x[0]
   dic['cdr'] = lambda x: x[1:]
   dic['list'] = lambda *elements: list(elements)
   dic['length'] = lambda l: len(l)
   return dic

def find_ref(ref, env):
   if ref in env:
      return env[ref]
   else:
      raise StandardError(ref + ' not found!')
```

A função *create_env*() define o ambiente global (contexto global) da linguagem uScheme. Ou seja, ele define o contexto onde as expressões primitivas da linguagem são válidas. Em termos de implementação, o ambiente é simplesmente um mapa que associa nomes de variáveis (referências) a seus valores (literais ou funções). O ambiente pode ser estendido com variáveis criadas pelo usuário através da expressão (define var valor). A função *create_env*() implementa o ambiente como um dicionário Python de tuplas {var: valor}. A função find_ref(ref, env) é usada para localizar uma variável no ambiente dado. Se a variável não existe, uma exceção é gerada. Dado o ambiente, agora é possível implementar o avaliador:

```
def eval(x, env):
   if type(x) == str:                  # referencia a variável
      return find_ref(x, env)
   elif type(x) == int or type(x) == float: # literal
      return x
   elif x[0] == 'define':               # definição
      [_, var, exp] = x
```

```
        env[var] = eval(exp, env)
    else:                           # chamada de funcao
        proc = eval(x[0], env)
        args = [eval(arg, env) for arg in x[1:]]
        return proc(*args)
```

A função *eval*() avalia uma expressão *x* em um ambiente *env*. Note que o ambiente inicial já inclui os operadores para soma, subtração, multiplicação e divisão, bem como primitivas *Scheme* para acessar listas:

Expressão	**Descrição**
car	Retorna primeiro elemento de lista. Exemplo: (car (list 1 2 3)) = 1
cdr	Retorna resto de uma lista. Exemplo: (cdr (list 1 2 3)) = (2 3)
list	Retorna uma lista. Exemplo: (list 1 2 3) = (1 2 3)
length	Retorna tamanho de uma lista. Exemplo: (length (list 1 2 3)) = 3

Finalmente, as funções abaixo fornecem uma interface *bem* simples para o interpretador (note que a implementação não é funcional – para interfaces, uma abordagem procedural é mais simples):

```
def formatted(exp):
    # exibe saída na forma usual do scheme.
    # Ex: [1,2,3] é impresso como (1 2 3)
    if type(exp) == list:
        return '(' + ' '.join([formatted(e) for e in exp]) + ')'
    else:
        return str(exp)

def scheme(prompt='> '):
    print 'Minimum scheme interpreter'
    print 'Type ^C to quit'
    global_env = create_env()    # cria ambiente global
    while True:
        try:
            user_input = get_semantic_tree(raw_input(prompt))
            val = eval(user_input, global_env)
            if val is not None:
                print(formatted(val))
        except KeyboardInterrupt:
            print 'Bye!'
            return
```

```
        except Exception as e:   # exibe mensagem da exceção, caso ocorra
            print '%s: %s' % (type(e).__name__, e)

if __name__ == '__main__':
    scheme()
```

12) Escreva as funções dadas e use seu interpretador para avaliar as seguintes expressões. Que resultados foram obtidos?

```
(+ 1 2)
(+ (- 1 2) (* 3 4))
(length (list 1 2 3))
(car (list 1 2 3))
(cdr (list 1 2 3))
(define x 10)
x
(+ x 2)
```

13) Estenda seu interpretador para suportar as expressões descritas na tabela abaixo.

Expressão	**Descrição**
%	Resto de uma divisão. Exemplo: (% 4 2) = 0
>, <, >=, <=, =	Operadores relacionais maior, menor, maior ou igual, menor ou igual e igual. Exemplo: (= 1 2) = False
begin	Retorna valor de última expressão. Exemplo: (begin (define x 10) (+ x 2)) = 12
cons	Insere elemento em lista. Exemplo: (cons 1 (list 2 3)) = (1 2 3); (cons (list 1 2) (list 3 4)) = ((1 2) 3 4)
null?	Retorna True se elemento é nulo. Exemplo: (null? (list)) = True; (null? (quote ())) = True
list?	Retorna True se parâmetro é uma lista. Exemplo: (list? (list 1 2 3)) = True
number?	Retorna True se parâmetro é um número. Exemplo: (number? 1) = True
symbol?	Retorna True se parâmetro é um símbolo. Exemplo: (symbol? (quote a)) = True
quote	Retorna parâmetros sem avalia-los. Exemplo: (quote (a b)) = (a b)
if	O valor da expressão (*if cond then else*) é dado pela avaliação da sub-expressão *then* caso a sub-expressão *cond* seja avaliada como True. Caso contrário, é dada pela

	sub-expressão *else*. Exemplo: (if (> 1 0) (quote (um eh maior)) (quote (zero eh maior))) = (um eh maior)

14) Usando seu novo interpretador, avalie as expressões dadas:

```
(begin (define pi 3.14) (* 2 (* pi pi)))
(cons 3 (list 1 2))
(define rest (cdr (list (quote a) (quote b) (quote c))))
rest
(if (null? (cdr (list 1 2))) (quote vazio) (quote (nao vazio)))
(if (> (* 11 11) 120) (* 7 6) oops)
```

PARTE II – funções definidas pelo usuário

O grande poder de uScheme está na possibilidade de estender a linguagem com novas expressões. Para isso, vamos modificar novamente nossa definição para suportar funções do usuário.

Expressão	**Descrição**
Funções	A expressão (*lambda* (*arg* ...) *body*) define uma nova função (sem nome) com conjunto de argumentos *arg*... e corpo *body*. Uma vez que a função é definida, ela pode ser posteriormente usada, com valores sendo atribuídos aos argumentos. Exemplo: > (define delta (lambda (a b c) (- (* b b) (* 4 (* a c))))) > (delta 1 2 3) -8 > (define a 7) > (delta 1 2 3) -8 > a 7

Uma expressão lambda em uScheme define, de fato, um novo ambiente onde uma função deve ser avaliada. No exemplo dado, o novo ambiente (*local*) define as variáveis *a*, *b* e *c*. Assim, quando (delta 1 2 3) é avaliada, as variáveis locais *a*, *b* e *c* recebem os valores 1, 2 e 3, respectivamente, e a função é avaliada como -8. Note que mesmo que uma definição global seja feita para *a* (define a 7), quando chamando (delta 1 2 3), o *a* local a *delta* é avaliado como 1 e não 7. Note que uma função pode definir uma função local, como no exemplo abaixo:

```
> (define imposto (lambda (taxa) (lambda (val) (* val taxa))))
```

```
> (define icms (imposto 0.18))
> (icms 100)
  18.0
```

Logo, podemos ter uma hierarquia de ambientes. Neste exemplo, o ambiente local de *icms* está dentro do ambiente local de *imposto* que, por sua vez, está dentro do ambiente *global*.

Assim, para dotarmos nosso interpretador do suporte a funções do usuário é necessário: (1) que um ambiente sempre saiba quem é o ambiente de referência em que ele foi criado, seu *ambiente pai* (ex: no exemplo dado, o ambiente pai de *icms* é o ambiente local de *imposto*; e o de *imposto*, é o *global*); (2) *find_ref*(*var*, *env*) deve ser modificada para que, caso *var* não seja encontrada em *env*, ela deve ser procurada no ambiente pai de *env* e, assim, *recursivamente*; (3) *eval*() deve suportar a expressão (*lambda* (*arg* ...) *body*). Ao encontrar essa expressão, *eval* deve retornar uma função que seja capaz de, quando chamada, construir o ambiente local, formado pelos argumentos e seus valores, e avaliar *body* nesse ambiente.

15) Estenda seu interpretador para suportar expressões *lambda*.

16) Usando seu novo interpretador, avalie as expressões dadas:

```
(define fatorial (lambda (n) (if (<= n 1) 1 (* n (fatorial (- n 1))))))
(fatorial 10)
(define cria-imposto (lambda (aliquota) (lambda (valor) (* valor aliquota))))
(define iss (cria-imposto 0.05))
(define icms (cria-imposto 0.18))
(iss 100)
(icms 100)
(define range (lambda (ini fim) (if (= ini fim) (quote ()) (cons ini (range (+ ini 1) fim)))))
(range 1 10)
(define map (lambda (f L) (if L (cons (f (car L)) (map f (cdr L))) (quote ()))))
(map (lambda (n) (* n n)) (range 0 10))
```

17) No exercício anterior, entre as funções definidas, está a função de alta ordem *map*. Agora, defina as funções de alta ordem *filter* e *reduce* em uScheme e as teste em seu interpretador.

18) Que modificações você deveria fazer em seu interpretador uScheme, para que fosse possível escrever um interpretador uScheme em uScheme?

19) Baseado em sua experiência com Python Funcional e uScheme, compare-as considerando os seguintes critérios:

   a. Simplicidade, Ortogonalidade, Tipos de Dados, Projeto de Sintaxe, Suporte à Abstração, Expressividade, Checagem de Tipos, Manipulação de Exceções e Restrição de *Aliases*

   b. Legibilidade, escrita e confiabilidade

Obs:

- Resposta pode ser entregue em dupla;
- Respondam com as próprias palavras. Plágio resultará em cancelamento das listas em que a cópia for observada;
- Data de entrega a ser definida no site da disciplina;
- *ATENÇÃO*: Este trabalho é baseado em um ensaio do professor Peter Norvig (Google). A diferença principal entre o ensaio original e o trabalho aqui proposto é que a implementação original é orientada a objetos. Aqui, estou solicitando uma implementação funcional do interpretador.