UNIVERSIDADE FEDERAL DE SANTA MARIA
CIÊNCIA DA COMPUTAÇÃO
PROVA DE TEORIA DA COMPUTAÇÃO 2017/1

Nome: Adonai Gabriel Loreto Peres Gonçalves **Matrícula:** 201410139

1.

(a) V - São teoremas equivalentes.
(b) V - Os dois identificam formalmente a noção de procedimento computável.
(c) V - Variável ligada muda, mas a livre continua a mesma.
(d) V - Reduz em n passos de beta-redução para xxz.
(e) F

Resposta: Letra **(e)**.

2.

- I -> F - Pode-se codificar números, booleanos e também listas.
- II -> V - É natural a definição de funções de alta ordem no cálculo-λ.
- III -> F - Não limita o poder de expressão.

Resposta: Letra **(b)**.

3.

Uma estratégia de avaliação é um conjunto de regras para determinar a avaliação de expressões em uma linguagem de programação. A ênfase é colocada, tipicamente, em funções ou operadores. Uma estratégia de avaliação define quando e em que ordem os parâmetros para uma função são avaliados, quando eles são substituídos dentro da função, e de que forma tal substituição ocorre. Diversas linguagens de programação modernas convergiram para uma estratégia de avaliação de chamada-por-valor ou chamada-por-referência para chamadas de função, por exemplo, C# e Java. Uma linguagem pode combinar várias estratégias de avaliação, por exemplo, C++, combina chamada-por-valor com chamada-por-referência.

Call-by-value: É uma das estratégias de avaliação mais comuns, usada em linguagens como C e Scheme. Na chamada por valor, o argumento é avaliado e o valor de resultado é vinculado à variável correspondente na função, copiando o valor em uma nova região de memória.
Normal Order: Avaliação de ordem normal é uma estratégia onde as expressões mais externas à esquerda (redexes) são sempre reduzidas em primeiro lugar, aplicando funções antes da avaliação dos argumentos.
Call-by-name: Os argumentos não são avaliados de maneira completa, tais argumentos são substituídos diretamente dentro do corpo da função usando substituição do tipo capture-avoiding. Se um parâmetro não é usado na avaliação da função, este nunca será avaliado, e se o parâmetro é usado várias vezes, este é reavaliado a cada vez. Haskell é a

linguagem mais conhecida que usa a estratégia de avaliação call-by-need, uma versão otimizada da call-by-name.
Full beta-reduction: Estratégia de avaliação onde qualquer expressão (redex) pode ser reduzida em qualquer momento. A relação de avaliação não é uma função, pois um termo pode avaliar para mais de um termo em um passo.

4.
Como a estratégia de avaliação call-by-name não precisa garantir que um valor é aplicado a uma função, sua definição torna-se mais simples:

Call-by-name lambda calculus (small-step semantics)

$$\frac{}{(\lambda x.M)\ N \rightsquigarrow_n M\{x/N\}}$$

$$\frac{M \rightsquigarrow_n M'}{M\ N \rightsquigarrow_n M'\ N}$$

5.
```
data LambExp
        = LambAbs Char LambExp
        | LambApp LambExp LambExp
        | Var Char
deriving (Show, Eq)

containsBetaRedex :: LambExp -> Bool
containsBetaRedex (LambApp (LambAbs x M) N) = True
containsBetaRedex (LambApp M N) = ((containsBetaRedex M) || (containsBetaRedex N))
containsBetaRedex (LambAbs x t) = containsBetaRedex t
containsBetaRedex (Var x) = False
```

6.
Combinador de ponto fixo é uma função y de alta ordem que, ao aplicar y a uma função arbitrária f, produz o mesmo resultado que f aplicada para o resultado da aplicação f para y. Um ponto fixo de uma função f é um valor que não é alterado sob a aplicação da função f. O combinador Y geralmente não termina quando aplicado a uma função com uma variável. Resultados mais interessantes são obtidos através da aplicação do combinador Y para funções de duas ou mais variáveis. A segunda variável pode ser usada como um contador, ou índice. A função resultante se comporta como um while ou um loop for em uma linguagem imperativa. Usado desta forma o combinador Y implementa recursão simples. No cálculo

lambda não é possível remeter para a definição de uma função de um corpo da função. Recursão só pode ser alcançada por passagem de uma função como um parâmetro. Um combinador de ponto fixo, como o combinador Y, pode ser aplicado a uma função gerando uma sequência recursiva. Por exemplo:

Considerando o combinador Y:
Y = (λx . λy . y (x x y)) (λx . λy . y (x x y))

Considerando Rec como a função a ser aplicada recursivamente, e aplicando Y à Rec, temos:

Y Rec
Expandindo: (λx . λy . y (x x y)) (λx . λy . y (x x y)) Rec
Beta: (λy . y ((λx . λy . y (x x y)) (λx . λy . y (x x y)) y)) Rec
Beta: Rec ((λx . λy . y (x x y)) (λx . λy . y (x x y)) Rec)
Contraindo: Rec (Y Rec)
Avalia para: Rec (Rec (Y Rec))
Avalia para: Rec (Rec (Rec (Y Rec)))
Avalia para: Rec (Rec (Rec (Rec (Y Rec))))
... e assim sucessivamente.