**MAC 0316/5754 — Conceitos de Linguagens de Programação**

Primeiro Semestre de 2011

Terceira Prova — 21 de junho de 2011

Nome do aluno: _____

Assinatura: _____

Nº USP: _____

**Instruções:**

1. Preencha o cabeçalho acima.
2. Não destaque as folhas deste caderno.
3. A prova tem 8 questões. Antes de começar a trabalhar verifique se o seu caderno de questões está completo.
4. A prova deve ser resolvida individualmente. Não é permitida a consulta a livros, apontamentos ou colegas.
5. Não é permitido o uso de folhas avulsas para rascunho.

**Boa prova!**

| Questão | Valor | Nota |
|---|---|---|
| 1 | 1,5 | |
| 2 | 1,5 | |
| 3 | 1,0 | |
| 4 | 1,0 | |
| 5 | 1,0 | |
| 6 | 2,0 | |
| 7 | 1,0 | |
| 8 | 1,0 | |
| Total | 10,0 | |

## Questão 1 (1,5 pontos)

O PLAI apresenta uma forma de conversão para CPS na qual as funções do usuário são reescritas de modo a receberem um parâmetro adicional `k`, mas certas funções básicas (como as funções aritiméticas `+`, `*`, etc.) permanecem inalteradas. Em vez de devolver algum valor, cada uma das funções convertidas para CPS efetua uma chamada à função `k`, passando como parâmetro o valor que seria devolvido pela função original. Já as funções básicas continuam devolvendo seus valores normalmente.

Nesta questão e na próxima questão você trabalhará com uma conversão para CPS "mais radical", na qual **todas** as funções (inclusive as básicas) devem ser reescritas.

Considere a seguinte função:

```
(define (sum-of-squares x y)
   (+ (* x x) (* y y)))
```

Converta essa função para uma forma de CPS na qual até as funções aritméticas estão em CPS. Em outras palavras, reescreva a função acima como

```
(define (sum-of-squares/k x y k)
   ... ; aqui vai o corpo da função em CPS
   ... )
```

A função reescrita **não** deve fazer chamadas às funções `+` e `*`. Em vez de chamar `+` e `*`, ela deve chamar as funções `+/k` e `*/k` (as versões em CPS de `+` e `*`), cujas definições são dadas a seguir.

```
(define (+/k x y k) (k (+ x y)))
(define (*/k x y k) (k (* x y)))
```

Uma chamada à função `sum-of-squares/k` deve fazer com que seu receptor `k` seja chamado com parâmetro igual à soma dos quadrados de `x` e `y`. A avaliação da soma dos quadrados deve ocorrer como na função `sum-of-squares` original, ou seja: primeiro o quadrado de `x`, depois o de `y`, e depois a soma.

## Questão 2 (1,5 pontos)

Considere esta função "comprimento de uma lista":

```
(define (length l)
   (if (empty? l)
       0
       (+ 1 (length (rest l)))))
```

Converta essa função para uma forma de CPS na qual até as funções `empty?`, `rest` e `+` estão em CPS. Em outras palavras, reescreva a função acima como

```
(define (length/k l k)
   ... ; aqui vai o corpo da função em CPS
   ... )
```

O corpo da função `length/k` pode conter chamadas **somente** à própria função `length/k` e às funções `empty?/k`, `rest/k` e `+/k`, cujas definições são dadas a seguir.

```
(define (+/k x y k) (k (+ x y)))
(define (empty?/k l k) (k (empty? l)))
(define (rest/k l k) (k (rest l)))
```

## Questão 3 (1,0 pontos)

Implemente `web-read` em termos de `let/cc` e `web-read/k`.

Para quem não lembra, estes são os contratos de `web-read` e `web-read/k`:

```
;; web-read: prompt-message -> value-read
(define (web-read prompt)
   ...) ;; sua tarefa é escrever esta função

;; web-read/k: prompt-message receiver-function -> [never returns]
(define (web-read/k prompt k)
   ...) ;; suponha que esta função já existe -- não a escreva!
```

## Questão 4 (1,0 pontos)

O PLAI apresenta a seguinte versão da codificação de Church para valores booleanos e da forma condicional:

```
yes ≡ (lambda (T F) T)
no  ≡ (lambda (T F) F)

(if C T F) ≡ (C T F)
```

Há um problema na forma `if` acima: ela pressupõe o uso de avaliação preguiçosa. No regime de avaliação ávido, essa forma `if` tem semântica diferente da forma `if` de Scheme. Conserte as coisas de modo que a forma `if` tenha a mesma semântica do `if` de Scheme, mesmo que o regime de avaliação seja ávido.

## Questão 5 (1,0 pontos)

Esta definição do combinador Y foi extraída do PLAI:

```
make-recursive-procedure ≡
   (lambda (p)
      ((lambda (f)
          (f f))
       (lambda (f)
          (p (f f)))))
```

Há um problema nessa definição. Explique qual é o problema e o corrija.

## Questão 6 (2,0 pontos)

O interpretador para a linguagem BCFAE usa uma função `next-location`, que o PLAI implementa da seguinte maneira:

```
;; next-location: Store -> location
(define next-location
  (local ([define last-loc (box -1)])
    (lambda (store)
      (begin
        (set-box! last-loc (+ 1 (unbox last-loc)))
        (unbox last-loc)))))
```

O PLAI observa que essa implementação de `next-location` é "extremamente insatisfatória", pois ela depende de uma *box* de Scheme.

Escreva uma implementação de `next-location` que não tenha efeitos colaterais. Você talvez precise modificar o contrato da função `next-location` e o interpretador da linguagem BCFAE. Se forem necessárias alterações no interpretador, indique claramente quais são essas alterações. Se você precisar de algum novo tipo de dados, ou se precisar de alguma mudança numa definição de tipo de dados que já existe, não se esqueça de escrever o `define-type` correspondente.

O código do interpretador aparece na página seguinte, para referência. Ele usa os seguintes tipos e funções auxiliares:

```
(define-type Env
  [mtSub]
  [aSub (name symbol?)
        (location number?)
        (env Env?)])
```

```
(define-type Store
  [mtSto]
  [aSto (location number?)
        (value BCFAE-Value?)
        (store Store?)])
```

```
(define-type Value*Store [v*s (value BCFAE-Value?) (store Store?)])

;;env-lookup: symbol Env -> location

;;store-lookup: location Store -> BCFAE-Value
```

Não reescreva toda a função `interp`! Caso você precise mexer nessa função, indique apenas que mudanças devem ser feitas nela.

### Interpretador para a linguagem BCFAE

```
;; interp : BCFAE Env Store -> Value*Store
(define (interp expr env store)
  (type-case BCFAE expr
    [num (n) (v*s (numV n) store)]
    [add (l r)
         (type-case Value*Store (interp l env store)
           [v*s (l-value l-store)
                (type-case Value*Store (interp r env l-store)
                  [v*s (r-value r-store)
                       (v*s (num+ l-value r-value)
                            r-store)])])]
    [id (v) (v*s (store-lookup (env-lookup v env) store) store)]
    [fun (bound-id bound-body)
         (v*s (closureV bound-id bound-body env) store)]
    [app (fun-expr arg-expr)
         (type-case Value*Store (interp fun-expr env store)
           [v*s (fun-value fun-store)
                (type-case Value*Store (interp arg-expr env fun-store)
                  [v*s (arg-value arg-store)
                       (local ([define new-loc (next-location arg-store)])
                         (interp (closureV-body fun-value)
                                 (aSub (closureV-param fun-value)
                                       new-loc
                                       (closureV-env fun-value))
                                 (aSto new-loc
                                       arg-value
                                       arg-store)))])])]
    [if0 (test truth falsity)
         (type-case Value*Store (interp test env store)
           [v*s (test-value test-store)
                (if (num-zero? test-value)
                    (interp truth env test-store)
                    (interp falsity env test-store))])]
    [newbox (value-expr)
            (type-case Value*Store (interp value-expr env store)
              [v*s (expr-value expr-store)
                   (local ([define new-loc (next-location expr-store)])
                     (v*s (boxV new-loc)
                          (aSto new-loc expr-value expr-store)))])]
    [setbox (box-expr value-expr)
            (type-case Value*Store (interp box-expr env store)
              [v*s (box-value box-store)
                   (type-case Value*Store (interp value-expr env box-store)
                     [v*s (value-value value-store)
                          (v*s value-value
                               (aSto (boxV-location box-value)
                                     value-value
                                     value-store))])])]
    [openbox (box-expr)
             (type-case Value*Store (interp box-expr env store)
               [v*s (box-value box-store)
                    (v*s (store-lookup (boxV-location box-value)
                                       box-store)
                         box-store)])]
    [seqn (e1 e2)
          (type-case Value*Store (interp e1 env store)
            [v*s (e1-value e1-store)
                 (interp e2 env e1-store)])]))
```

## Questão 7 (1,0 pontos)

Vimos em classe a linguagem FWAE, cuja sintaxe (concreta) é definida da seguinte maneira, em BNF:

```
<FWAE> ::= <num>
         | {+ <FWAE> <FWAE>}
         | {with {<id> <FWAE>} FWAE}
         | <id>
         | {fun {<id>} FWAE}
         | {FWAE FWAE}
```

Em FWAE os identificadores têm escopo estático. Agora considere a linguagem "FWAE com escopo dinâmico", ou FWAE/D. Essa é uma linguagem com sintaxe idêntica à de FWAE e semântica muito parecida com a de FWAE. A única diferença é que em FWAE/D o escopo dos identificadores é dinâmico.

(a) Dê um exemplo de programa que rode sem problemas tanto sob um interpretador FWAE quanto sob um interpretador FWAE/D, mas produza um resultado numérico diferente em cada uma dessas linguagens.

(b) Dê um exemplo de programa que gere um erro de "identificador livre" quando executado por um interpretador FWAE, mas rode sem problemas e produza um resultado numérico quando executado por um interpretador FWAE/D.

## Questão 8 (1,0 pontos)

Nesta questão você receberá um ponto por qualquer coisa que escrever sobre esta disciplina. Críticas e sugestões de melhorias são bem vindas. Você não receberá esse ponto, entretanto, se deixar a questão em branco ou se escrever algo que não tenha nenhuma relação com Conceitos de Linguagens de Programação. Use o espaço abaixo e, se for o caso, continue no verso da folha.