MAC 0316/5754 — Conceitos de Linguagens de Programação

Primeiro Semestre de 2010

Terceira Prova — 15 de junho de 2010

Nome do aluno: _____

Assinatura: _____

Nº USP: _____

Instruções:

1. Preencha o cabeçalho acima.
2. Não destaque as folhas deste caderno.
3. A prova tem 7 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	2,0	
2	1,0	
3	1,0	
4	2,0	
5	1,5	
6	1,5	
7	1,0	
Total	10,0	

Questão 1 (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. 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 2 (1,0 pontos)

Explique o que é *l-value*. Dê um exemplo de *l-value* na linguagem VCFAE e outro na linguagem C.

Questão 3 (1,0 pontos)

Escreva o trecho do interpretador para a linguagem VCFAE que é responsável pela interpretação de expressões da forma

```
{set <id> <VCFAE>}
```

Em outras palavras, escreva a cláusula

```
[set (var value)
     ...
     ...
     ...]
```

do `type-case` mais externo do interpretador. Sugestão: Olhe a cláusula `setbox` do interpretador da linguagem BCFAE, dado na questão 1, e pense na diferença entre *boxes* e variáveis.

Questão 4 (2,0 pontos)

A função abaixo calcula o total de uma lista de valores digitados pelo usuário:

```
(define (totaliza lista-de-itens)
  (if (empty? lista-de-itens)
      0
      (+ (web-read (gera-prompt (first lista-de-itens))))
         (totaliza (rest lista-de-itens)))))
```

Transforme essa função para que ela possa ser usada por uma aplicação Web. Em outras palavras, escreva uma função `totaliza/k`, que chame `web-read/k`, em vez de chamar `web-read`, e que passe o total calculado a um receptor `k`, em vez de devolver o total como valor da função:

```
(define (totaliza/k lista-de-itens k) ... )
```

Questão 5 (1,5 pontos)

Considere a seguinte função `produto`, que recebe uma lista de números não vazia e devolve o produto dos números da lista:

```
(define (produto l)
  (if (empty? (rest l))
      (first l)
      (* (first l) (produto (rest l)))))
```

Suponha que a lista de números pode ser muito longa e que há grande chance dela conter algum zero. Com esses pressupostos, vale a pena comparar cada elemento da lista com zero. Caso apareça um zero, a função `produto` pode parar de fazer multiplicações e devolver zero imediatamente (já que o produto certamente será zero se algum dos fatores for igual a zero). Esta é uma tentativa de melhorar a função `produto`:

```
(define (produto l)
  (if (= (first l) 0)
      0
      (if (empty? (rest l))
          (first l)
          (* (first l) (produto (rest l))))))
```

A versão acima para de fazer chamadas recursivas quando encontra um zero, mas mesmo assim ela efetua multiplicações na volta de cada chamada recursiva. Assim, na avaliação de

```
(produto '(4 3 2 1 0 -1 -2))
```

são executadas quatro chamadas à função `*`: `(* 1 0)`, `(* 2 0)`, `(* 3 0)` e `(* 4 0)`.

Usando `let/cc`, escreva uma versão da função produto que não faça nenhuma chamada à função `*` se a lista `l` contiver algum zero. Dica: Tome como base o esqueleto abaixo.

```
(define (produto l)
  (let/cc k
    (local [(define (produto-aux l)
                      ...
                      ...
                      ...)]
      (produto-aux l))))
```

Questão 6 (1,5 pontos)

Explique qual a estratégia que o PLAI usou para construir um interpretador para a linguagem KCFAE (CFAE com suporte a continuações). Você não precisa escrever o código do interpretador para a linguagem KCFAE, mas deve explicar cuidadosamente qual foi a sequência de modificações que o Krishnamurthi aplicou num interpretador tradicional (sintático) para a linguagem CFAE, de modo a chegar num interpretador para a linguagem KCFAE.

Questão 7 (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. Comece no espaço abaixo e, se for o caso, continue no verso da folha.