MAC 0316/5754 — Conceitos de Linguagens de Programação

Primeiro Semestre de 2011

Segunda Prova — 17 de maio de 2011

Versão revisada, com uma pequena correção na questão 2

Nome do aluno: _____

Assinatura: _____

Nº USP: _____

Instruções:

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

Questão 1 (1,5 pontos)

Considere a linguagem CFAE (*Conditionals, Functions and Arithmetic Expressions*), cuja sintaxe concreta (definida em BNF) é a seguinte:

```
<CFAE> ::= <num>
         | <id>
         | {+ <CFAE> <CFAE>}
         | {fun {<id>} CFAE}
         | {CFAE CFAE}
         | {if0 CFAE CFAE CFAE}
         | {with {<id> CFAE} CFAE}
```

Esta é a sintaxe abstrata da linguagem CFAE:

```
(define-type CFAE
  [num (n number?)]
  [id (name symbol?)]
  [add (lhs CFAE?) (rhs CFAE?)]
  [fun (param symbol?) (body CFAE?)]
  [app (fun-expr CFAE?) (arg-expr CFAE?)])
  [if0 (test CFAE?) (truth CFAE?) (falsity CFAE?)])
```

Considere o seguinte esqueleto de analisador sintático (*parser*) para a linguagem CFAE:

```
;; parse: s-expression -> CFAE
(define (parse sexp)
  (cond
    [(number? sexp) (num sexp)]
    [(symbol? sexp) (id sexp)]
    [(list? sexp)
     (case (length sexp)
       ...
       [(3) (case (first sexp)
              ...
              [(with) ... ] ; <--- análse sintática da forma with
              ...              )]
       ...  )]))
```

Implemente a parte do *parser* que faz a análise sintática da forma `with`. Em outras palavras, escreva o trecho de código que ocupará o lugar da linha assinalada com `<---` no esqueleto acima. (Note que há uma forma `with` na sintaxe concreta da linguagem CFAE, mas não há variante `with` na sintaxe abstrata!)

Questão 2

(1,0 + 1,5 + 1,5 pontos)

Este é um interpretador para a linguagem CFAE:

```
1  ;; interp : CFAE Env -> CFAE-Value
2  (define (interp expr env)
3    (type-case CFAE expr
4               [num (n) (numV n)]
5               [add (l r) (numV (+ (numV-n (interp l env))
6                                   (numV-n (interp r env))))]
7               [if0 (test truth falsity)
8                    (if (zero? (numV-n (interp test env)))
9                        (interp truth env)
10                       (interp falsity env))]
11              [id (v) (lookup v env)]
12              [fun (bound-id bound-body)
13                   (closureV bound-id bound-body env)]
14              [app (fun-expr arg-expr)
15                   (local ([define fun-val (interp fun-expr env)])
16                     (interp (closureV-body fun-val)
17                             (aSub (closureV-param fun-val)
18                                   (interp arg-expr env)
19                                   (closureV-env fun-val))))]))
```

Deseja-se mudar a semântica da linguagem CFAE para que o regime de avaliação passe a ser avaliação preguiçosa.

(a) Como deve ser o tipo `CFAE-Value` para que a avaliação seja preguiçosa? Escreva a definição (`define-type`) do novo tipo `CFAE-Value`.

(b) Mostre que modificações devem ser feitas no interpretador acima para ele implementar avaliação preguiçosa. (Indique apenas as linhas que mudarão e o novo conteúdo dessas linhas.) Suponha que é dada uma função `strict`, a qual deve ser chamada nos *strictness points* da linguagem. Este é o contrato da função `strict`:

```
;; strict :  CFAE-Value -> CFAE-Value [exceto fechamento de expressão]
```

(c) Implemente a função `strict`.

Questão 3 (1,0 pontos)

Considere novamente a funçao `interp` do enunciado da questão anterior — a função original, sem as modificações para avaliação preguiçosa. Deseja-se fazer um teste rápido com uma linguagem semelhante (e sintaticamente igual) à CFAE, porém com escopo dinâmico. Como modificar rapidamente aquela função `interp` para que a linguagem passe a ter escopo dinâmico? Indique apenas as linhas que mudarão e o novo conteúdo dessas linhas.

Como o objetivo é fazer um teste rápido, não é necessário remover todo o código que deixa de fazer sentido no caso de uma linguagem com escopo. O que se deseja é a "menor" modificação que produza o efeito desejado.

Questão 4 (2,0 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.

O código do interpretador para a linguagem BCFAE está na última página deste caderno de prova, para referência. Sugestão: Olhe o interpretador BCFAE e pense na diferença entre *boxes* e variáveis.

Questão 5 (1,5 pontos)

Simule a execução do seguinte programa RCFAE:

```
{rec {fac {fun {n}
             {if0 n
                  1
                  {* n {fac {+ n -1}}}}}}
  {fac 2}}
```

Em outras palavras, simule a execução de `(interp (parse '`*`progr`*`) (mtSub))` onde *progr* é o programa RCFAE acima. Mostre o ambiente (*environment*) passado como parâmetro em cada chamada recursiva à função `interp`. Mostre também o ambiente contido em cada fechamento criado por alguma chamada a `interp`.

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)])]))
```