

MAC 0316/5754 — Conceitos de Linguagens de Programação

PRIMEIRO SEMESTRE DE 2010

Primeira Prova — 23 de março de 2010

Versão revisada, com pequenas correções na questão 1(c) e 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 4 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	3,0	
2	3,5	
3	2,5	
4	1,0	
Total	10,0	

Questão 1

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

- (a) Escreva uma função `filtra` que funcione como a função `filter` da biblioteca padrão de Scheme: dados um predicado `p` e uma lista `l`, a chamada `(filtra p l)` devolve a lista formada pelos elementos de `l` para os quais o predicado `p` é *true*. Exemplo de chamada a `filtra`:

```
(filtra even? (list 1 2 3 4 5 6))  
> (list 2 4 6)
```

- (b) Sua função `filtra` certamente faz uma ou mais chamadas recursivas a si mesma. Indique em qual/quais dessas chamadas a recursão é de cauda e em qual/quais a recursão não é de cauda.
- (c) Reescreva a sua função `filtra` de modo que a área de memória empregada pela pilha de execução tenha tamanho constante (em vez de ser proporcional ao comprimento da lista que se deseja filtrar). Sugestão: Crie uma função auxiliar, seguindo uma estratégia análoga à vista em classe para a função `map`, e faça a nova função `filtra` chamar a função auxiliar.

Questão 2 (1,5 + 0,5 + 1,5 pontos)

É dada uma função **preco**, que recebe um símbolo de um produto e devolve o preço do produto. (Você **não** precisa escrever essa função. Suponha que alguém já a escreveu.) Exemplos de chamadas a **preco**:

(uma dúzia de bananas custa 4 reais)

```
'([jornal 1] [duzia-de-bananas 3] [litro-de-leite 2] [lata-de-cerveja 6] ... )
```

(a) Escreva uma função **total**, que receba uma lista de compras e devolva o preço total dos produtos listados, levando em conta as quantidades dos produtos. Para obter o preço de cada produto, a função **total** deve chamar a função **preco**.

- (b) Escreva uma função **soma**, que recebe uma lista de números e devolve a soma de todos os números da lista. A função **soma** é uma função auxiliar para o próximo ítem. Implemente-a de modo bem simples. (Não precisa usar recursão de cauda.)
- (c) Reescreva a função **total**, agora usando a função **map** de Scheme e a função **soma** do ítem anterior. Dica: Converta a lista de compras numa lista de números, sendo cada número o custo (já levando em conta a quantidade) do ítem correspondente da lista de compras.

Questão 3

(2,5 pontos)

Este analisador sintático (*parser*) para a linguagem AE foi extraído do PLAI:

```
;; parse: s-expression -> AE
(define (parse sexp)
  (cond
    [(number? sexp) (num sexp)]
    [(list? sexp)
     (case (first sexp)
       [(+) (add (parse (second sexp))
                  (parse (third sexp)))]
       [(-) (sub (parse (second sexp))
                  (parse (third sexp)))]))]))
```

Usando a função acima como base, escreva um analisador sintático para a linguagem WAE. Esta é a definição, em BNF, da sintaxe concreta da linguagem WAE:

```
<WAE> ::= <num>
        | {+ <WAE> <WAE>}
        | {- <WAE> <WAE>}
        | {with {<id> <WAE>} WAE}
        | <id>
```

Esta é a definição, em PLAI Scheme, da sintaxe abstrata da linguagem WAE:

```
(define-type WAE
  [num (n number?)]
  [add (lhs WAE?) (rhs WAE?)]
  [sub (lhs WAE?) (rhs WAE?)]
  [with (name symbol?) (named-expr WAE?) (body WAE?)]
  [id (name symbol?)])
```

No que diz respeito à detecção de erros, seu analisador sintático deve ser melhor que o extraído do PLAI. Coisas como `{+ 1 2 3}` devem fazer o *parser* parar com uma mensagem de erro. Para obter esse efeito, use chamadas (`error 'parse string-com-a-mensagem`).

Questão 4

(0,5 + 0,5 pontos)

- (a) Esta é a definição, em BNF, da sintaxe concreta da linguagem F1WAE:

```
<F1WAE> ::= <num>
          | {+ <F1WAE> <F1WAE>}
          | {with {<id> <F1WAE>}} F1WAE}
          | <id>
          | {<id> F1WAE}
```

No caso de uma aplicação de função, a expressão passada como argumento à função é do tipo F1WAE. Por que essa expressão não é do tipo WAE, em vez de ser do tipo F1WAE? Dê um exemplo de programa que seja razoável, que seja permitido pela sintaxe na qual o argumento de função é F1WAE e que seja rejeitado pela sintaxe na qual o argumento de função é WAE.

- (b) A linguagem F1WAE não inclui uma forma de definir funções. As definições de funções tem de ser feitas fora da linguagem e são passadas ao interpretador numa lista de **FunDefs**. Esta é a estrutura de uma **FunDef**:

```
(define-type FunDef
  [fundef (fun-name symbol?)
          (arg-name symbol?)
          (body F1WAE?)])
```

Por que o corpo (**body**) de uma definição de função é do tipo F1WAE, em vez de ser do tipo WAE? Dê um exemplo de definição de função que seja razoável, que seja possível se o corpo da definição for F1WAE e que não seja possível se o corpo da definição for WAE.

