Exame de Linguagens de Programação II

18 Janeiro 2006

(duração: 2h)

I [8]

O objectivo deste exercício é desenvolver um interpretador para a linguagem AKA, definida pela seguinte gramática

```
Expr ::=  Expr {+} Expr | Expr {-} Expr |
          Expr {*} Expr | Expr {/} Expr |
          {fun} Id {->} Expr |
          Expr(Expr) |
          {try} E {catch} E |
          Num | Id  | {(} Expr {)}
```

onde os símbolos terminais estão representados entre { } e `Num` e `Id` representam respectivamente os literais numéricos (ex: 0, 98) e os identificadores (ex. i, xpto, marmota). Os programas nesta linguagem são expressões, gerados pela categoria sintáctica `Expr`. Os únicos valores manipulados pela linguagem são os números inteiros, produzidos pela avaliação das expressões.

Para simplificar, as funções aceitam apenas um parâmetro. Dentro da função, o seu parâmetro representa um valor inteiro, não uma variável de estado (tal como um parâmetro `const` em C).

A resolução de nomes na linguagem AKA usa a regra correcta de resolução estática. Durante a execução de um programa AKA pode ocorrer um erro de divisão por zero. Tais erros podem ser detectados torneados usando a construcção `try` E `catch` F. Tal como sucede com as excepções na linguagem Java, se durante a avaliação de E ocorrer uma tentativa de divisão por zero, a avaliação de E será interrompida, continuando a avaliação com a F. Se um erro não for detectado, o valor final da expressão é indefinido (o interpretador aborta). Por exemplo, a expressão

```
2 / (1-1)
```

não tem valor (o interpretador aborta com um erro).

A expressão

```
(fun x -> 2 + (try 4 / x catch x+2)) (4-4)
```

tem o valor 2:

A expressão

```
try 4 / (2 * try 2-2 catch 2) catch 0
```

tem o valor 0.

1. [2] Explique como representar a sintaxe abstracta da linguagem IMP usando um conjunto de classes na linguagem Java.
2. [6] Especifique a semântica operacional da linguagem AKA, programando em cada uma das classes que apresentou em (1) acima, os métodos de avaliação

   ```
   ? evaluate(a: Environment)
   ```

   de tal modo que, dada uma AST guardada na variável `exp`, a chamada

   ```
   exp.evaluate(new Environment())
   ```

   o execute completamente, ou assinale um erro de interpretação.

   Importante: `Environment` **implementa a interface "standard" usada nas aulas.**

II [6]

Este exercício trata de sistemas de tipos. Imagine que se pretende desenvolver um sistema de tipos para a linguagem AKA, com o objectivo de garantir estaticamente a ausência de erros de divisão por zero. A linguagem resultante é a poderosa AKAT! A ideia é considerar os tipos

$$\texttt{Type} ::= ANY \mid POS \mid NEG \mid ZERO$$

ANY é o tipo das expressões que não produzem erros, $ZERO$ é um tipo atribuído a expressões que têm como valor zero, POS é um tipo atribuído a expressões com valor estritamente positivo, e NEG um tipo atribuído a expressões com valor estritamente negativo. A sintaxe da linguagem AKAT é a mesma da linguagem AKA, excepto que agora será necessário declarar o tipo do parâmetro das funções

```
Expr ::=  ...
          {fun} Id:Type {->} Expr
          ...
```

1. [2] Descreva um sistema de tipos que atribuia um tipo razoável a cada expressão da linguagem AKAT. Preferivelmente, apresente o sistema de tipos usando um conjunto de regras (poderá indicar mais que uma regra para cada construção da linguagem, de modo a cobrir várias alternativas). Por exemplo, o seu sistema poderá derivar a asserção de tipificação

   $$x : NEG \vdash x * x : POS$$

   indicando que em qualquer contexto onde x tem tipo NEG, a expressão $x * x$ terá tipo POS.

2. [2] Usando o sistema de tipos anterior, indique qual o tipo esperado das expressões, apresentando as respectivas derivações:

   (a) `(fun x:NEG -> x * x)` POS

   (b) `(fun x:NEG -> x + x)` NEG

3. [1]Para além dos erros de divisão por zero, o seu sistema de tipos impede a ocorrência de outro tipo de erros?

4. [1] Indique uma expressão da linguagem AKAT que não produz nenhum erro de divisão por zero, mas que não é tipificável usando o seu sistema de tipos.

III [6]

1. [3] Considere o programa CORE

   ```
   decl
     x = 1
     y = decl z = 1 in z+3 end
         +
         decl x =2 in x * 3 end
   in print(x+y) end
   ```

   Apresente uma sequência de instruções CIL para este programa, usando apenas (preferivelmente duas) variáveis locais de tipo `int32`.

2. [3] Explique porque a compilação da linguagem CORE estudada na disciplina necessita de recorrer a uma pilha-ambiente alojada no heap (spaghetti-stack).