# Segunda Prova de MAB 471 2013.1 — Compiladores I

Fabio Mascarenhas

31 de Julho de 2013

A prova é individual e sem consulta. Responda as questões na folha de respostas, a lápis ou a caneta. Se tiver qualquer dúvida consulte o professor.

Nome: _____

DRE: _____

| Questão: | 1 | 2 | 3 | Total |
|---|---|---|---|---|
| Pontos: | 4 | 3 | 3 | 10 |
| Nota: | | | | |

1. Considere a gramática a seguir, para um fragmento de TINY:

```
LISTA -> CMD ; LISTA | CMD
CMD -> id := EXP
EXP -> id | id ( ) | num
```

   (a) (2 pontos) Dê a sequência de ações (shift, reduce, accept) do analisador LR para o termo `id := num ; id := id ()`. Não é preciso dar o número dos estados nas ações de shift!

   (b) (2 pontos) Dê o estado inicial do autômato LR(0) dessa gramática, as transições que saem desse estado, e os estados alvo dessas transições. Pode abreviar os não-terminais LISTA, CMD e EXP como L, C e E. LISTA é o não-terminal inicial da gramática.

2. (3 pontos) Uma notação popular para o tipo de uma função que recebe $n$ argumentos de tipos $\tau_1, \ldots, \tau_n$ e retorna um valor com tipo $\tau_r$ é $\tau_1 \times \ldots \times \tau_n \to \tau_r$. Assim, a tipagem de uma chamada de função pode ser dada pela seguinte regra, onde $\Gamma$ é um mapa de nomes de variáveis para tipos, $\Phi$ é um mapa de nomes de funções para tipos, e $\Gamma, \Phi \vdash e : \tau$ significa que a expressão $e$ tem o tipo $\tau$, dados os mapas $\Gamma$ e $\Phi$, e $\upsilon \leq \tau$ quer dizer que o tipo $\upsilon$ é compatível com o tipo $\tau$:

$$\frac{\Phi(f) = \tau_1 \times \ldots \times \tau_n \to \tau_r \quad \Gamma, \Phi \vdash e_1 : \upsilon_1 \quad \ldots \quad \Gamma, \Phi \vdash e_n : \upsilon_n \quad \upsilon_1 \leq \tau_1 \quad \ldots \quad \upsilon_n \leq \tau_n}{\Gamma, \Phi \vdash f(e_1, \ldots, e_n) : \tau_r}$$

   Dada a classe `TipoFuncao` abaixo, implemente o método de verificação de tipos da classe `ChamadaFuncao`, seguindo a regra acima. O método vem da interface `Expressao`, e tanto `Expressao` quando `ChamadaFuncao` também são dados abaixo. Use a função `compativel` de `Tipo` para verificar a a compatibilidade (igualdade) dos tipos. Não se esqueça de verificar se o número de argumentos é igual ao número de parâmetros (verificação de aridade).

```
class Tipo {
  public static void compativel(String t1, String t2) {
    // lança um erro se t1 e t2 forem incompatíveis
  }
}

class TipoFuncao {
  String[] tipoArgs;
  String tipoRet;
}

interface Expressao {
  String tipo(Map<String, String> vars, Map<String, TipoFuncao> funcs);
  void geraCodigo(Contexto c);
}

class ChamadaFuncao implements Expressao {
  String nomeFunc;
  Expressao[] args;
  ...
}
```

3. A JVM é uma máquina de pilha, ou seja, as instruções operam não sobre registradores mas sobre valores em uma pilha. A instrução **ldc** $n$ empilha um número literal $n$, a instrução **iload** $n$ empilha o valor da $n$-ésima variável local (que deve ser inteira), a instrução **istore** $n$ desempilha o valor que está no topo da pilha e o guarda na $n$-ésima variável local, as instruções **iadd**, **isub**, **idiv** e **imul** desempilham dois valores e empilham o resultado da operação correspondente (soma, subtração, multiplicação e divisão, respectivamente), e a instrução **invokestatic** $f$ desempilha quantos parâmetros a função $f$ tiver e chama $f$ com esses argumentos, empilhando o valor de retorno.

   (a) (1 ponto) Escreva o código JVM para o seguinte comando TINY, assumindo que `a`, `b` e `c` são as variáveis locais de números 0, 1 e 2, e `sin` e `cos` são funções:

   ```
   c := cos(a) * sin(b) + sin(a) * cos(b)
   ```

   (b) (2 pontos) Escreva o método `void geraCodigo(Contexto c)` para a classe `ChamadaFuncao` da questão 2. Assuma que o objeto do tipo `Contexto` tem métodos correspondentes a cada instrução da JVM, que emitem o código para a instrução.

BOA SORTE!