CEFET-MG
CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS

CONCURSO PÚBLICO 2023

DOCENTE EBTT *CAMPUS* LEOPOLDINA

EDITAL Nº 30/2023

ETAPA DA PROVA ESCRITA

COMPUTAÇÃO

Número de inscrição do Candidato

ABRA SOMENTE QUANDO AUTORIZADO

INSTRUÇÕES

1. Confira se as informações sobre a Área do Concurso, o número do Edital e o *Campus*, que estão descritas na capa deste caderno de prova, estão de acordo com os dados do seu comprovante definitivo de inscrição.
2. Transcreva o número da sua inscrição para todas as páginas do caderno de prova, das folhas definitivas de respostas e folhas de rascunho, usando o espaço reservado no rodapé. Essa informação está disponível no seu comprovante definitivo de inscrição e/ou na lista de presença.
3. É proibido assinar e fazer qualquer tipo de marcação no caderno de prova, nas folhas definitivas de respostas e nas folhas de rascunho, sob a possibilidade de eliminação deste Concurso Público.
4. Responda às questões nas folhas definitivas de respostas fornecidas pelo aplicador.
5. Nenhuma folha deverá ser destacada do caderno de prova, das folhas definitivas de respostas ou do rascunho.
6. Não serão fornecidas folhas extras para rascunho ou para responder às questões de prova.
7. As folhas de rascunho não serão corrigidas pela Banca Examinadora. As folhas definitivas de respostas são o único instrumento que será avaliado e sob nenhuma hipótese serão substituídas.
8. O caderno de prova, as folhas definitivas de respostas e as folhas de rascunho deverão ser entregues juntos para o Aplicador.
9. O tempo regulamentar de prova é de 4:30 (quatro horas e trinta minutos) e será exigido um tempo mínimo de sigilo de 1 (uma) hora.
10. Esta prova contém 15 (quinze) questões, sendo 10 (dez) objetivas e 05 (cinco) dissertativas e será avaliada em 100,00 (cem pontos).
11. Durante a realização da Prova Escrita, o candidato poderá portar somente: caneta esferográfica de tinta preta ou azul, de corpo transparente.

QUESTÃO 01 (5,0 pontos)

Considere o problema de decidibilidade sobre linguagens livres de contexto. Seja $G = (E, \Sigma, R, P)$ uma gramática livre de contexto (GLC) e $L(G)$ a linguagem reconhecida por G. Qual dos seguintes enunciados descreve um problema DECIDÍVEL?

a) Determinar se $L(G) = \varnothing$ para qualquer GLC G.

b) Determinar se $L(G) = \Sigma^*$ para qualquer GLC G.

c) Determinar se $L(G)$ é ambígua para qualquer GLC G.

d) Determinar se $L(G_1) \subseteq L(G_2)$ para quaisquer GLCs G_1 e G_2.

e) Determinar se $L(G_1) = L(G_2)$ para quaisquer GLCs G_1 e G_2.

QUESTÃO 02 (5,0 pontos)

Sobre linguagens regulares, considere as afirmativas:

I. A união de uma linguagem regular com uma linguagem não regular é regular.

II. A união de uma linguagem regular com uma linguagem não regular não é regular.

III. A união de duas linguagens não regulares não é regular.

IV. O complemento de uma linguagem não regular não é regular.

A(s) sentença(s) correta(s) é(são)

a) apenas I.

b) I e II.

c) II e III.

d) II, III e IV.

e) apenas IV.

QUESTÃO 03 (5,0 pontos)

Sejam L_1 uma linguagem recursivamente enumerável (Turing-reconhecível) e L_2 uma linguagem recursiva (Turing-decidível). Analise as afirmativas classificando-as em verdadeira (**V**) ou falsa (**F**).

( ) $\overline{L_1}$ é recursivamente enumerável.

( ) $L_1 \cup L_2$ é recursivamente enumerável.

( ) $L_1 - L_2$ é recursivamente enumerável.

( ) $L_2 - L_1$ é recursivamente enumerável.

A sequência correta é

a) F, F, V, V.

b) F, V, V, F.

c) F, V, F, V.

d) V, F, F, V.

e) V, V, F, F.

QUESTÃO 04 (5,0 pontos)

No contexto da linguagem de programação C, analise as afirmativas que associam um conceito da linguagem a um tempo de amarração, classificando-as em verdadeira (**V**) ou falsa (**F**).

( ) A declaração de uma variável é feita em tempo de implementação.

( ) O tamanho do tipo inteiro é definido em tempo de compilação.

( ) A inclusão de uma biblioteca pela diretiva `#include` é feita em tempo de ligação.

( ) A associação de áreas de memória a variáveis globais é feita em tempo de carga.

( ) A chamada de uma função é feita em tempo de execução.

A sequência correta é

a) F, F, F, V, V.

b) F, F, V, F, V.

c) F, V, V, F, F.

d) V, F, F, V, F.

e) V, V, F, F, V.

QUESTÃO 05 (5,0 pontos)

Considere o programa a seguir em uma linguagem hipotética que possui escopo dinâmico, cujos arranjos são indexados começando pelo índice zero. A execução inicia na função `main`.

```
01:  fun f(var x):
02:      a[0] = 2
03:      i = 1
04:      x = x + 1
05:
```

```
06:  fun main():
07:      var arr[] = { 0, 1 }
08:      var i = 0
09:      f(arr[i])
10:      print(arr[0], arr[1], i)
```

Relacione o tipo de passagem de parâmetros com a saída esperada para esse programa caso a linguagem suporte esse tipo de passagem.

Passagem de Parâmetros	Saída Esperada
1. Por valor	( ) 1 1 1
2. Por valor-resultado.	( ) 2 1 1
3. Por referência.	( ) 2 2 1
4. Por nome.	( ) 3 1 1

A sequência correta da associação é

a) 1, 2, 3, 4.

b) 1, 3, 4, 2.

c) 2, 1, 4, 3.

d) 3, 2, 1, 4.

e) 4, 1, 2, 3.

QUESTÃO 06 (5,0 pontos)

Considere um programa escrito em Java, uma linguagem orientada a objetos que possui tipagem estática. Um programador criou uma hierarquia de classes conforme código a seguir: a classe `C` herda as propriedades da classe `B`, enquanto a classe `B` por sua vez herda as propriedades da classe `A`.

```
class A { … }
class B extends A { … }
class C extends B { … }
```

Considere as variáveis `a`, `b` e `c` que possuem, respectivamente, os tipos `A`, `B` e `C`. Analise se as atribuições a seguir são seguras ou não-seguras, ou seja, se podem ser compiladas ou não.

( ) `a = b;`

( ) `b = a;`

( ) `b = c;`

( ) `c = b;`

A sequência correta é

a) não-segura, não-segura, segura, segura.

b) não-segura, segura, não-segura, segura.

c) não-segura, segura, segura, não-segura.

d) segura, não-segura, não-segura, segura.

e) segura, não-segura, segura, não-segura.

QUESTÃO 07 (5,0 pontos)

Em linguagens de programação estaticamente tipadas, em qual fase da compilação é feita a verificação de tipos?

a) Análise léxica.

b) Análise sintática.

c) Análise semântica.

d) Otimização de código.

e) Geração de código.

QUESTÃO 08 (5,0 pontos)

Considerando as diferenças entre os analisadores sintáticos *top-down* e *bottom-up* em compiladores, qual das seguintes afirmações é verdadeira?

a) O *top-down* é mais eficiente que o *bottom-up* na resolução de conflitos empilhar/reduzir.

b) Somente o *bottom-up* é capaz de lidar com gramáticas recursivas à esquerda.

c) O *top-down* é baseado em *backtracking*, enquanto o *bottom-up* é baseado em *lookahead*.

d) O *bottom-up* usa uma pilha para acompanhar a estrutura sintática, enquanto o *top-down* usa uma árvore de derivação.

e) O *top-down* analisa a partir dos símbolos terminais da entrada e o *bottom-up* analisa a partir do símbolo inicial da gramática.

QUESTÃO 09 (5,0 pontos)

Qual das seguintes alternativas é verdadeira sobre análise léxica em compiladores?

a) A análise léxica é responsável por transformar a entrada de caracteres em uma sequência de tokens para a análise sintática.

b) A análise léxica é responsável por gerar a árvore de derivação da gramática.

c) A análise léxica é a segunda fase do processo de compilação.

d) A análise léxica não faz distinção entre palavras reservadas e identificadores.

e) A análise léxica é responsável por detectar erros sintáticos na entrada.

QUESTÃO 10 (5,0 pontos)

Em otimização de código, as seguintes técnicas são transformações função-preservantes, **EXCETO**:

a) Eliminação de subexpressões comuns.

b) Eliminação de código morto.

c) Propagação de cópias.

d) Transposição para constantes.

e) Remoção de restrições.

QUESTÃO 11 (10,0 pontos)

Construa:

a) Um autômato finito determinístico que reconheça $L = \{01,10\}^*\{00,11\}^*$. **(2,5 pontos)**

b) Uma gramática regular que reconheça $L = \{ w \in \{0,1\}^* \mid w$ começa e termina com o mesmo símbolo$\}$. **(2,5 pontos)**

c) Um autômato de pilha determinístico que reconheça $L = \{ 0^n1^{2n} \mid n \geq 0 \}$. **(2,5 pontos)**

d) Uma gramática livre de contexto que reconheça $L = \{ w \in \{0,1\}^* \mid w \neq w^R \}$

(w^R é o reverso de w: se $w = a_1a_2...a_n$, então $w^R = a_n...a_2a_1$). **(2,5 pontos)**

QUESTÃO 12 (10,0 pontos)

Considere a gramática a seguir em formato de Backus-Naur (BNF), versão não-estendida, que permite definir conjuntos de números naturais entre 0 (zero) e 9 (nove). Essa gramática suporta a criação de conjunto vazio (ex.: `{}`) e de conjunto com apenas um elemento (ex.: `{2}`). Para formar conjuntos com mais elementos, deve-se combinar outros conjuntos utilizando as operações de complemento (¬), interseção (∩) e/ou união (∪).

```
<set> ::= { }
    | { <number> }
    | ¬ <set>
    | <set> ∩ <set>
    | <set> ∪ <set>
<number> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

a) Mostre que essa gramática é ambígua. **(5,0 pontos)**

b) Modifique essa gramática, sem alterar sua linguagem, de forma a remover sua ambiguidade e para que os operadores sejam associativos à esquerda. Além disso, o operador ∪ (união) deve ter menor precedência e o operador ¬ (complemento) maior precedência. A gramática deve ser mantida no formato BNF não-estendido. **(5,0 pontos)**

QUESTÃO 13 (10,0 pontos)

Considere o seguinte trecho de código em Java para sacar um determinado valor do saldo de uma conta bancária.

```
public void sacar(double valor) {
    if (valor <= 0.0)
        throw new SaqueException("Valor inválido");
    else if (this.saldo < valor)
        throw new SaqueException("Saldo insuficiente");

    this.saldo -= valor;
}
```

a) Baseado nesse trecho de código, indique porque `SaqueException` é uma exceção não verificada. **(2,5 pontos)**

b) Mostre a implementação dessa exceção (`SaqueException`) caso ela fosse verificada. **(2,5 pontos)**

c) Reimplemente o método `sacar` considerando agora que essa exceção é verificada. **(5,0 pontos)**

QUESTÃO 14 (10,0 pontos)

Considere a função `codificar`, conforme vista a seguir à esquerda, que dado um caracter qualquer (`char c`) ela retorna uma codificação desse caractere a partir de uma chave (`int chave`) e um deslocamento (`int deslocamento`) dessa chave usando operações em bits. Essa função, quando compilada sem otimizações, produz o código em assembly para arquiteturas *x86-32* similar ao visto a seguir à direita.

```
char codificar(char c, int chave, int deslocamento) {
  return c ^ (chave >> ((deslocamento % 4) * 8));
}
```

```
pushl   %esp
subl    $4, %esp
movl    8(%esp), %eax
movb    %al, (%esp)
movl    16(%esp), %eax
cltd
shrl    $30, %edx
addl    %edx, %eax
andl    $3, %eax
subl    %edx, %eax
sall    $3, %eax
movl    12(%esp), %edx
movl    %eax, %ecx
sarl    %cl, %edx
movl    %edx, %eax
xorb    (%esp), %al
popl    %esp
ret
```

Considere uma versão simplificada desse assembly para *x86-32* que possui as seguintes características:

- Um programa é formado por uma sequência de instruções.
- Cada instrução é formada por um opcode (ex.: `pushl`) seguido de zero, um ou dois operandos separados por vírgula.
- Um operando pode ser uma constante representada por um número inteiro precedido do símbolo `$` (ex.: `$30`), um registrador representado por um identificador precedido do símbolo `%` (ex.: `%esp`) ou um acesso à memória, ou seja, um registrador entre parênteses precedido ou não de um número (ex.: `8(%esp)`) ou `(%esp)`).

a) Desenvolva um analisador léxico para essa linguagem em formato de autômato finito determinístico. Transições de erro não precisam ser modeladas. **(5,0 pontos)**

b) Desenvolva uma gramática livre de contexto não ambígua em formato de Backus-Naur estendido (EBNF) para essa linguagem. Suponha que já existam três regras auxiliares para essa arquitetura: `<opcode>` que produz um de todos os opcodes válidos para esse processador; `<register>` que produz o nome de um de todos os registradores disponíveis não precedido de `%`; e `<integer>` que obtém um número inteiro não precedido de `$`. Suponha também que as instruções podem usar quaisquer tipos de operandos. **(5,0 pontos)**

QUESTÃO 15 (10,0 pontos)

Considere uma linguagem de alto nível hipotética que possui passagem de parâmetros e retorno via cópia em uma memória em formato de pilha. Nessa linguagem foi escrita a função *`magic`*, conforme pode ser visto a seguir à esquerda. Essa função recebe um arranjo de inteiros cujos valores nunca são modificados. Ela faz algumas operações sobre os valores e retorna um determinado resultado. Essa função pode ser compilada em uma linguagem de código de três endereços hipotética sem otimizações, conforme pode ser visto a seguir à direita.

```
fun magic(const a):
  s = 0
  i = 0
  while (i <= len(a)):
    t = a[i] + 2
    s += t + 3
    i++
  return s
```

```
  $t0 <- pop       # variável a
  $t1 <- const 0   # variável s
  $t2 <- const 1   # variável i
%loop:
  push $t0
  call len
  $t3 <- pop       # retorno de len
  jgt $t2, $t3, %end
  $t4 <- load $t2($t0)
  $t5 <- const 2
  $t6 <- add $t3, $t5
  $t7 <- const 3
  $t8 <- add $t6, $t7
  $t1 <- add $t1, $t8
  $t2 <- inc $t2
  jmp %loop
%end:
  push $t1
  ret
```

Essa linguagem de três endereços utiliza variáveis temporárias, indicadas pelo nome `$tn`. Essas variáveis podem ser manipuladas livremente, ou seja, podem ser adicionadas e/ou removidas na geração ou otimização do código. Essa linguagem possui instruções que têm exatamente o mesmo custo de execução, independente de seu tipo. As instruções suportadas são somente:

- `push $tx`: coloca o valor da variável `$tx` na pilha;
- `pop`: retira e obtém o valor do topo da pilha;
- `call` *`name`*: chama a função de nome *`name`*;
- `ret`: retorna de uma função;
- `const n`: obtém o valor do literal `n`;
- `jmp %label`: pula incondicionalmente para o ponto indicado por `%label`;
- `jgt $tx, $ty, %label`: pula para o ponto indicado por `%label` somente se o valor da variável `$tx` for maior que `$ty`;
- `load $tx($ty)`: carrega o valor do endereço indicado pela variável `$ty` a partir do deslocamento indicado pela variável `$tx`;

- `add $tx, $ty`: adiciona os valores das variáveis `$tx` e `$ty` e obtém o resultado;
- `inc $tx`: incrementa em uma unidade o valor indicado por `$tx` e obtém o resultado.

a) Qual(is) otimização(ões) pode(m) ser realizada(s) nesse código de três endereços de forma a torná-lo o mais eficiente possível em termos de custo de execução. Dê uma breve explicação de como funciona(m) essa(s) otimização(ões). **(5,0 pontos)**

b) Mostre o código transformado após aplicação dessa(s) otimização(ões) descrita(s) no item (a). **(5,0 pontos)**