Primeira Prova de Linguagens de Programação
- DCC024 -
Ciência da Computação

Nome: _____
"Eu dou minha palavra de honra que não trapacearei neste exame."

Número de matrícula: _____

As regras do jogo:

- A prova é sem consulta.
- Quando terminar, não entregue nada além do caderno de provas para o instrutor.
- Quando escrever código, a sintaxe correta é importante.
- Cada estudante tem direito a fazer uma pergunta ao instrutor durante a prova. Traga o caderno de provas quando vier à mesa do instrutor.
- A prova termina uma hora e quarenta minutos após seu início.
- Seja honesto e lembre-se: **você deu sua palavra de honra**.

Alguns conselhos:

- Escreva sempre algo nas questões, a fim de ganhar algum crédito parcial.
- Se não entender a questão, e já tiver gasto sua pergunta, escreva a sua interpretação da questão junto à resposta.
- Serão avaliadas somente as seis melhores respostas. Então sinta-se livre para abandonar alguma questão devido ao tempo.
- A prova não é difícil, ela é divertida, então aproveite!

Tabela 1: Pontos acumulados (para uso do instrutor)

Questão 1	Questão 2	Questão 3	Questão 4	Questão 5	Questão 6	Questão 7

1. (10 Pontos) Considere a implementação SML do algoritmo de ordenação *quicksort* abaixo:

```
fun leq a b = a <= b

fun grt a b = a > b

fun filter _ nil = nil
  | filter f (h::t) = if f h then h :: filter f t else filter f t

fun qsort nil = nil
  | qsort (h::t) = (qsort (filter (grt h) t)) @ [h] @ (qsort (filter (leq h) t))
```

O método de ordenação *quicksort* baseia-se em uma idéia relativamente simples: dada uma lista l com pelo menos um elemento h, separamos l em duas metades, l_1 e l_2, de forma que todos os elementos de l_1 são menores que h, e todos os elementos de l_2 são maiores. Então, ordenamos cada uma destas sublistas, produzindo l'_1 e l'_2. O resultado final é a concatenação de l'_1 com a lista formada somente por h, com a lista l'_2. Simples, não? E é possível ver este padrão claramente no programa acima. Por isto, dizemos que este programa é *de alto nível*. Considere agora a implementação da mesma função em C:

```
void quickSort( int a[], int l, int r) {
  int j;
  if( l < r ) {
    j = partition( a, l, r);
    quickSort( a, l, j-1);
    quickSort( a, j+1, r);
  }
}
int partition( int a[], int l, int r) {
  int pivot, i, j, t;
  pivot = a[l]; i = l; j = r+1;
  while( 1) {
    do ++i; while( a[i] <= pivot && i <= r );
    do --j; while( a[j] > pivot );
    if( i >= j ) break;
    t = a[i]; a[i] = a[j]; a[j] = t;
  }
  t = a[l]; a[l] = a[j]; a[j] = t;
  return j;
}
```

A função em C faz a mesma coisa: usa o algoritmo de ordenação *quicksort* para ordenar um arranjo de inteiros. Muitas pessoas vão achar a função escrita em C mais complicada que a outra, feita em SML. Por outro lado, ela é inegavelmente mais eficiente em termos de tempo e espaço: entre 50 e 100 vezes! Explique duas razões desta maior eficiência da função escrita em C.

2. Dizemos que uma linguagem é segura quando esta linguagem não permite que operações sejam aplicadas a argumentos que não possuam os tipos previstos por estas operações. C e C++ são linguagens inseguras, pois muitas vezes valores armazenados em memória são utilizados sem qualquer fiscalização de seus tipos.

   (a) (5 Pontos) Escreva um programa em C ou C++ que evidencie o caráter inseguro de uma destas linguagens.

   (b) (5 Pontos) Existem linguagens mais antigas que C ou C++ que são consideradas seguras, logo, a possibilidade de uso inseguro de tipos não é devido à ignorância sobre os perigos desta abordagem. ML, por exemplo, já havia sido definida dez anos antes de C++, porém enquanto ML é uma linguagem considerada segura, C++ não é. Cite um fator que motivou o desenho inseguro de C++.

3. (10 Pontos) Considere o tipo algébrigo `bx`, cuja implementação é dada abaixo:

```
datatype bx = TRUE | FALSE | AND of bx * bx | OR of bx * bx | NOT of bx
```

Este tipo algébrico representa expressões booleanas, usando a semântica tradicional dos operadores lógicos `AND`, `OR` e `NOT`. Escreva uma função `interp` em SML, de tipo `bx -> bool`, que receba um elemento do tipo `bx` e produza o valor booleano correspondente. Por exemplo:

```
- interp TRUE;
val it = true : bool

- interp FALSE;
val it = false : bool

- interp (NOT TRUE);
val it = false : bool

- interp (AND (TRUE, FALSE));
val it = false : bool

- interp (OR (FALSE, FALSE));
val it = false : bool

- interp (AND (OR (FALSE, FALSE), TRUE));
val it = false : bool

- interp (OR (TRUE, AND (TRUE, FALSE)));
val it = true : bool
```

4. (2.5 pontos cada) Uma expressão λ que não pode mais ser reduzida é dita estar em forma *normal*. Abaixo temos quatro expressões λ, nenhuma das quais está em forma normal. Reduza estas expressões, até que você encontre uma forma normal para elas, ou explique porque tal não pode ser feito, quando for o caso, isto é, a forma normal não existe. Atente para o problema da *captura de variáveis livres*. Você irá precisar das seguintes definições para a penúltima questão:

   - $T = \lambda x.\lambda y.x$
   - $F = \lambda x.\lambda y.y$
   - $NOT = \lambda b.bFT$

   (a) $(\lambda x.(\lambda y.(x(\lambda x.xy))))y$

   (b) $(\lambda m.\lambda n.\lambda x.\lambda y.mx(nxy))(\lambda a.\lambda b.b)(\lambda c.\lambda d.c(cd))$

   (c) $(\lambda x.x\ F\ NOT\ F)(\lambda a.\lambda b.a(a(ab)))$

   (d) $(\lambda x.xx)(\lambda y.yy)$

5. Considere as duas gramáticas abaixo, e diga se alguma delas é ambígua. O símbolo `E` significa a palavra vazia. Estas gramáticas reconhecem a mesma linguagem, formada por *strings* enclausuradas por parênteses e colchetes balanceados. Se a gramática for ambígua, escreva as suas duas árvores de derivação. Do contrário explique porque você acha que a gramática não apresenta ambiguidades.

   (a) (5 Pontos) Primeira gramática:

```
<string> ::= <string> <string>
           |  ( <string> )
           |  [ <string> ]
           |  E
```

   (b) (5 Pontos) Segunda gramática:

```
<string> ::= ( <string> ) <string>
           |  [ <string> ] <string>
           |  E
```

6. (10 Pontos) Nós vimos em sala de aula que a maior parte das linguagens modernas usam *registros de ativação* para armazenar dados tais como valores de parâmetros, endereço de retorno, valores das variáveis locais, etc. Além disto, linguagens que permitem que uma função seja declarada dentro de outra, como SML, também possuem um *link de aninhamento* em seu registro de ativação. O link de aninhamento em um registro de ativação de uma função f aponta para o registro de ativação de uma função g, que é a função dentro da qual f foi definida. A função do link de aninhamento é permitir que variáveis declaradas na função "de fora"sejam usadas na função "de dentro". Por exemplo, considere a função `foo` abaixo, implementada em SML:

```
fun foo n =
  let
    fun bar 0 = []
      | bar m = n :: bar (m - 1)
  in
    bar n
  end
```

O link de aninhamento de `bar` irá sempre apontar para a última ativação de `foo`. Para que `bar` possa encontrar o valor da variável `n`, esta função percorre seu link de aninhamento. Linguagens que possuem o conceito de *bloco de código* também usam uma pilha em memória para armazenar as variáveis locais criadas dentro de cada bloco. Considere, por exemplo, o programa abaixo, escrito em C, que ordena um arranjo de inteiros:

```
1  int sort(int a[], int size) {
2    int i = 0;
3    while (i < size - 1) {
4      int j = i + 1;
5      while (j < size) {
6        if (a[i] > a[j]) {
7          SWAP(a, i, j);
8        }
9        j++;
10     }
11     i++;
12   }
13 }
```

Note que o bloco que vai da linha 5 à linha 10 usa as variáveis `i` e `j`, que foram declaradas fora deste bloco. Por outro lado, *links de aninhamento não são necessários para implementar blocos de código.* Explique o porquê.

7. (10 pontos) Muitas linguagens de programação são compiladas diretamente para cóigo de máquina. Outras são interpretadas. E outras executam sobre uma máquina virtual. É possível um programa executando sobre uma máquina virtual ser mais eficiente que um programa que foi compilado diretamente para código de máquina? Justifique a sua resposta.