Segunda Prova de Construção de Compiladores.
Primeiro Semestre de 2003.
Departamento de Computação – UFS-Car.
José de Oliveira Guimarães.
Turma B (Quinta).

Lembre-se: justifique tudo a menos de menção em contrário.

Entregue apenas a folha de respostas. Isto é, não entregue esta folha ou o rascunho.

1. (3.0) Dada a gramática

```
E ::= E + T | T | id
T ::= T*F | F
F ::= N | id
```

faça os itens a seguir:

(a) encontre o first/follow de cada não terminal;

(b) baseado no first/follow, explique porquê esta gramática é ambígua;

`first`(X) é computado como:

1. **se X for um terminal, `first`(X) = { X };**
2. **se X ::= ϵ for uma produção, adicione ϵ a `first`(X);**
3. **se X for não terminal e X ::= $Y_1 Y_2 ... Y_k$ for uma produção, então coloque f em `first`(X) se, para algum i, f $\in$ `first`(Y_i) e $\epsilon \in$ `first`(Y_j), $j = 1, 2, ...i-1$. Se $\epsilon \in$ `first`(Y_j), $j = 1, 2, ... k$, então coloque ϵ em `first`(X).**

`follow`(X) é computado como:

1. **coloque eof em `follow`(S), onde S é o símbolo inicial;**
2. **se houver uma produção da forma A ::= $\alpha X \beta$, então `first`(β), exceto ϵ, é colocado em `follow`(X);**
3. **se houver uma produção da forma A ::= αX ou uma produção A ::= $\alpha X \beta$ e $\epsilon \in$ `first`(β), então colocaremos os elementos de `follow`(A) em `follow`(X).**

2. (2.5) Faça o autômato finito que reconheça a seguinte expressão regular (ER):

```
0+1*([A-Z]|w)
```

Faça o desenho e o código em Java que reconhece a ER. Este código deve chamar a função `error()` se houver um erro mas não deve fazer nada se a entrada for reconhecida pela ER. Um modelo de código é dado abaixo.

```
void analyze( char in[] ) {
   int s = 1; // estado inicial
   int k = 0;
   ...
}
```

3. (2.0) Escolha e faça UM e apenas UM dos itens abaixo.

(a) Cite duas aplicações de técnicas de compilação (este curso !) que não sejam traduzir um programa escrito em uma linguagem de programação para outra (a aplicação mais comum).

(b) Cite três vantagens de interpretadores sobre compiladores.

4. (2.0) Retire a recursão à esquerda e fatore a gramática

```
E ::= E + T | T * E | T
T ::= T F | F
F ::= N | id
```

5. (2.5) Monte a tabela de análise descendente não recursiva da seguinte gramática:

```
S ::= A | B d
A ::= a A | c
B ::= b | ϵ
```

6. (2.0) Otimize os seguintes trechos de código. Indique na sua resposta exatamente onde você fez as otimizações.

(a) (duas otimizações)

```
    cmp x, 5
    goto== L1
    goto L2
L1: mov a, 0
L2:
```

(b) (sete otimizações)

```
{
   int i = 0, j = 0, n = 0;
   const Max = 100;
   n = Max/10;
   a += f(i);
   while ( j < n ) {
      if ( g(i, 160*j) > 0 ) {
         break;
         j--;
      }
      j++;
      a += f(i);
   }
}
```

7. (1.0) Algumas linguagens possuem um comando `for`, como Pascal, Ada, Green, Algol, etc. Este comando possui a forma

```
for i = low to high do
   body
```

Onde o `i` assume todos os valores entre `low` e `high`. Gerando código em C para este comando, teríamos

```
i = low;
limit = high; // high pode ser uma expressao
while ( i <= limit ) {
   body  // comandos
   i++;
}
```

Contudo, esta geração de código está ERRADA. Explique porquê e corrija o erro. Dica: você precisará utilizar `goto`.