UNIVERSIDADE DO ESTADO DA BAHIA – UNEB
SISTEMAS DE INFORMAÇÃO – CAMPUS I
FUNDAMENTOS DE COMPILADORES – PROVA II

Aluno/a: GABARITO – POSSÍVEIS RESPOSTAS _____

Instruções: Esta prova está dividida em duas partes: questões de cunho teórico e questões de aplicação prática dos conceitos em compiladores. Responda as 10 (dez) questões da prova exclusivamente na folha de respostas pautada que acompanha esta folha de questões; nenhuma resposta indicada nesta folha de questões será considerada; identifique cada questão com o seu respectivo número na folha de respostas; respostas à lápis não darão direito à revisão da correção; assine também a folha de respostas. Qualquer questão tem valor de 1 (um) ponto.

Parte teórica

1. Desenvolva e apresente uma gramática livre de contexto (GLC), G, para a seguinte linguagem: $\{w \in \{a, b\}^* \mid w = a^n b^n, \text{ para } n \geq 0\}$. Indique na sua resposta quais são os símbolos não-terminais e qual deles é o símbolo de partida, assim como as respectivas regras de produção de G.

   $S \rightarrow aSb \mid \varepsilon$, com {S} o conjunto dos símbolos não-terminais e S o símbolo de partida

2. Para a gramática G, apresentada como resposta da questão anterior, demonstre o processo de derivação (por construção de árvore sintática ou por formas sentenciais), a partir do símbolo de partida, para 2 (dois) exemplos de cadeias válidas da linguagem geradas pela gramática.

   Por formas sentenciais:

   $S \Rightarrow aSb \Rightarrow a\varepsilon b \Rightarrow ab$

   $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aa\varepsilon bb \Rightarrow aabb$

3. Ainda considerando a gramática G das questões anteriores, apresente uma outra gramática G', que seja equivalente a G. Na sua resposta, indique as mesmas informações para G' que foram solicitadas para a apresentação de G.

   G': $(\{a, b\}, \{S, V\}, P, S)$, com {a, b} o alfabeto, {S, V} o conjunto de símbolos não-terminais, S o símbolo de partida e P as regras de produção como a seguir.

   P: $S \rightarrow V \mid \varepsilon$
   $V \rightarrow aVb \mid ab$

4. Demonstre que a gramática de operadores lógicos a seguir é ambígua. $GL = (\{v, f, e, ou\}, \{X\}, P, X)$, com {v, f, e, ou} o conjunto dos símbolos terminais, no qual "ou" e "e" são operadores e "v" e "f" são operandos da linguagem, {X} é o conjunto de símbolos não-terminais, X é o símbolo de partida e P é formado pelas seguintes regras de produção: $X \rightarrow X$ **e** $X \mid X$ **ou** $X \mid v \mid f$.

Derivando a cadeia “v ou f e v” por formas sentenciais, temos:

- X => X e X => X ou X e X => v ou f e v
- X => X ou X => X ou X e X => v ou f e v

Assim, temos a mesma cadeia sendo gerada a partir de derivações de formas sentenciais distintas, caracterizando a ambiguidade da gramática.

5. Para a gramática GL anterior, encontre e apresente uma GL’ não ambígua, equivalente a GL, que, adicionalmente, implementa a precedência do operador “e” maior que a precedência do operador “ou” nas árvores sintáticas produzidas para cadeias válidas da linguagem.

   GL’: ({v, f, e, ou}, {X, T, F}, P, X), com {v, f, e, ou} o alfabeto, {X, T, F} o conjunto dos símbolos não-terminais, X o símbolo de partida e P como a seguir.

   P: X → X ou T | T
   T → T e F | F
   F → v | f

   Dessa forma GL’ é não ambígua e sempre que o operador “e” ocorrer numa cadeia, que também contenha o operador “ou”, ele terá precedência de execução (a execução do “ou” dependerá da execução, antes, da operação “e” associada).

6. Sabendo-se que gramática GL anterior possui como característica a recursividade à esquerda na montagem de suas árvores sintáticas e que, genericamente, se a gramática possui regras de produção do tipo A→Aα | β, estas podem ser transformadas em A→βR e R→αR | ε, encontre GL’’ equivalente, a partir de GL, aplicando esses passos, de modo que GL’’ não seja recursiva à esquerda. Não se preocupe, nessa transformação, se GL’’ permanecer ambígua tal qual GL.

   Tomando-se as regras de produção X→X **e** X | v de GL como ponto de partida e aplicando-se os passos acima, temos que $A = X$, $\alpha = eX$ e $\beta = v$; portanto, para essas regras de produção, temos para GL’’: X→vR e R→ eXR | ε. Generalizando-se para as outras regras de produção de GL, temos que as regras de produção (P) de GL’’ ficam:

   X→vR | fR
   R→ e XR | ou XR | ε, com {v, f, **e**, **ou**} como alfabeto, {X, R} como conjunto de símbolos não-terminais e X como símbolo de partida.

Parte prática

7. Seja GD a gramática que gera **cadeias de caracteres** que representam números naturais no sistema decimal, definida como GD = ({‘0’,’1’,’2’,’3’,’4’,’5’,’6’,’7’,’8’,’9’}, {S, N}, P, S), com {‘0’,’1’,’2’,’3’,’4’,’5’,’6’,’7’,’8’,’9’}, o conjunto dos símbolos terminais, {S, N} o conjunto de símbolos não-terminais, S o símbolo de partida e P as regras de produção no formato: S→SN | N e N→’0’ | ‘1’ | … | ‘9’. Apresente uma possível definição dirigida pela sintaxe (DDS), ou seja, as regras semânticas para cada regra de produção de GD, que converta qualquer cadeia de caracteres gerada por GD no seu respectivo número decimal.

A DDS solução a seguir opera fazendo uso de atributos sintetizados:

Regra de Produção	Regra semântica
$S \rightarrow S_1 N$	$S.t = S_1.t*10 + N.t$
$S \rightarrow N$	$S.t = N.t$
$N \rightarrow$ '0'	$N.t = 0$
$N \rightarrow$ '1'	$N.t = 1$
...	...
$N \rightarrow$ '9'	$N.t = 9$

8. A gramática de uma determinada linguagem de programação possui o comando condicional **if** apresentado a partir dos seguintes fragmentos do seu conjunto de regras de produção: <cmd> → **if** <expr_cond> **then** <bloco_cmds>, <cmd> → **if** <expr_cond> **then** <bloco_cmds$_1$> **else** <bloco_cmds$_2$>. Ou seja, só é possível saber se um comando **if** terá um **else** associado ou não após processar toda a sua parte inicial. Usando a técnica da fatoração de regras de produção, apresente um novo conjunto de regras de produção para o comando **if** dessa linguagem de modo que resolva esse problema.

   Fatorando-se a parte comum das duas regras de produção de <cmd> apresentadas, temos:

   <cmd> → <cmd_if> <else_opc>
   <cmd_if> → if <expr_cond> then <bloco_cmds>
   <else_opc> → else <bloco_cmds> | ε

9. Na linguagem Do::Block, o comando **while** está assim definido na especificação de sua gramática: *cmd* ::= **while (** *expr* **)** { *cmd* } **endwhile**. Apresente o código da função em linguagem C que pode ser usada no processo de análise sintática descendente recursiva para esse comando. Utilize as chamadas do analisador léxico necessárias tal qual você usou no seu próprio projeto do compilador de Do::Block.

```
void trata_while() {

      if (tk.cat == PR && tk.codigo == WHILE) {
            tk = analex();
            if (tk.cat != SN || tk.codigo != ABRE_PAR)
                  erro(1);   // Abre parênteses esperado
            tk=analex();
            expr();
            if (tk.cat != SN || tk.codigo != FECHA_PAR)
                  erro(2);   // Fecha parênteses esperado
            tk=analex();
            while (tk.cat != PR || tk.codigo != ENDWHILE)
                  cmd();
      }
}
```

10. Traduza o trecho de código Do::Block a seguir para o respectivo código da máquina de pilha, como se gerado por esquemas de tradução fosse, em um processo de compilação por tradução dirigida pela sintaxe que aplica o modelo descendente recursivo. Como referência, caso necessite, as regras de produção do comando **if** e do comando de atribuição são as seguintes:

cmd ::= **if (** *expr* **)** { *cmd* } { **elseif** **(** *expr* **)** { *cmd* } } [**else** { *cmd* }] **endif**

atrib ::= **id** { **[** *expr* **]** } = *expr*

```
if (a > b)
        c = a + b
else
        c = a – b
endif
```

Sucesso!

```
...
LOAD    a
LOAD    b
SUB
GOTRUE  L1      // compara e deixa 1 ou 0 no topo de pilha
PUSH    0       // a depender se “a” é ou não é maior que “b”.
GOTO    L2
LABEL   L1
PUSH    1
LABEL   L2
GOFALSEL3       // teste da condição do if
LOAD    a
LOAD    b
ADD
STOR    c
GOTO    L4
LABEL   L3      // else
LOAD    a
LOAD    b
SUB
STOR    c
LABEL   L4
...
```