Primeira Prova de Construção de Compiladores.
Primeiro Semestre de 2005, DC-UFSCar
Prof. José de Oliveira Guimarães
Turma A (Terça-feira).

1. (5.5) Faça:

a) métodos

```
RepeatStatement repeatStatement()
Variable variableDeclaration()
```

que fazem a análise sintática, semântica e construção da ASA para as seguintes regras de gramática:

```
RepeatStatement ::= "repeat" CommandList "until"  Expr
VariableDeclaration ::= IDENT ":" Type
```

Naturalmente, a expressão do repeat deve ser inteira e o identificador não pode ter sido declarado antes. Assuma que já existam métodos

```
CommandList commandList()
Expr expr()
Type type()
```

que analisam uma lista de comandos, uma expressão e um tipo. `IDENT` é um terminal que representa um identificador. A classe `Compiler` possui um método `nextToken` responsável pela análise léxica. O token corrente é colocado na variável de instância `token` de `Compiler`. As constantes da análise léxica estão na classe `Symbol`. Há uma variável de instância `symbolTable` na classe `Compiler` do tipo `Hashtable` (lembre-se de que esta classe possui métodos `Object get(Object key)` e `Object put(Object key, Object value)`). O método `error` de `Compiler` deve ser utilizado para emitir mensagens de erro. `IDENT` e `NUMBER` são terminais. Se o token corrente for `Symbol.IDENT`, a variável de instância `stringValue` de `Compiler` guarda o identificador (uma string). Se `token` for `Symbol.NUMBER`, a variável de instância `numberValue` de `Compiler` guarda o valor do número, que é inteiro. Os tipos são representados por objetos de `IntegerType`, `BooleanType` e `CharType` e existe um único objeto de cada um dos tipos, referenciados pelas variáveis estáticas `Type.integerType`, `Type.booleanType` e `Type.charType`;

b) as classes da ASA RepeatStatement e Variable. Declare as variáveis de instância e os métodos genC para gerar código em C — não se preocupe com construtores e outros métodos. O método genC de Variable deve gerar código para a declaração da variável.

2. (1.5) Faça o analisador sintático para a seguinte gramática:

P ::= A { A }
A ::= '@' ['w'] B | '$' C
C ::= ['~'] D

Utilize um método para cada não terminal. Cada método deve analisar somente o que está na regra de mesmo nome da gramática — isto é, em um método a(), por exemplo, não chame nextToken para eliminar um terminal que não está na regra A da gramática. Ou teste, em a(), pela presença de um terminal que não está na regra A. Assuma que há métodos já prontos para os não terminais cujas regras não são mostradas. Assuma que o método nextToken() do analisador léxico já está pronto. Não crie a ASA. Teste pela presença de um terminal usando algo como

```
if ( token == 'a' ) ...
```

3. (1.5) Faça o código em Java que crie a ASA do seguinte trecho de código da linguagem do compilador 9:

```
var a : integer;
begin
read(a);
write(a);
end
```

4. (3.5) Sobre a tabela de símbolos, responda às questões abaixo.
a) (1.5) Quando símbolos devem ser eliminados da tabela ? Explique utilizando um exemplo.

b) (2.0) Adicione no trecho de código abaixo instruções para a análise semântica de uma atribuição. Utilize as informações dadas na questão 1, item a). Assuma que a linguagem é a do compilador 9, que contém três tipos: char, boolean e integer. Em uma atribuição, os tipos da variável e expressão devem ser iguais.

```
private AssignmentStatement assignmentStatement() {
      // token == Symbol.IDENT
      // name é o nome da variável
   String name = (String ) lexer.getStringValue();
   lexer.nextToken();
   Variable v;
   if ( lexer.token != Symbol.ASSIGN )
     error.signal("= expected");
   ...
   Expr right = orExpr();
   return new AssignmentStatement( v, right );
}
```