PROCESADORES DE LENGUAJE (LABORATORIO) *Septiembre-2008*
4º Curso Ingeniería Informática Tiempo: 2 horas
Nombre: Apellidos: Grupo:

(3 ptos) Pregunta 1:

Queremos generar listas enumeradas en HTML. Para ello, tenemos las siguientes palabras reservadas: la pareja **TKN_OL** y **TKN_END_OL** que indican el inicio y final de una lista y la pareja **TKN_LI** y **TKN_END_LI** que encierran un ítem. Las expresiones regulares que definen los componentes léxicos de nuestro lenguaje son:

```
TKN_OL:         "<OL>"
TKN_END_OL:     "</OL>"
TKN_LI:         "<LI>"
TKN_END_LI:     "</LI>"
TKN_ID:         [a-zA-Z]+
TKN_BLANCO:     [\n\t ]
```

Un ejemplo de las sentencias a generar es:

```
<OL>
<LI> este es el primer ítem </LI>
      <OL>
      <LI> un ítem de otra lista </LI>
      <LI> otro ítem de otra lista </LI>
      </OL>
<LI> más ítems </LI>
</OL>
```

La gramática debe cumplir que:
- En el fichero de datos entrada debe haber al menos una lista.
- Pueden existir listas dentro de listas.
- No pueden existir listas vacías, sin ítems u otras listas.
- Detrás o delante de una lista pueden ir un ítem u otra lista.
- A un ítem le puede seguir otro ítem o una lista.

Se pide:

a) (1,0 ptos.) Diseñar una gramática LL(1) que genere dicho lenguaje.

b) (1,5 ptos.) Implementar los procedimientos correspondientes según el método de análisis descendente recursivo incorporando la recuperación en modo de pánico.

c) (0,5 ptos.) Diseñar un atributo que permita contar el número de listas anidadas que aparecen e implementar el código correspondiente para evaluarlo. Emitir un mensaje si el número de anidamientos es superior a 3 niveles.

PROCESADORES DE LENGUAJE (LABORATORIO) *Septiembre-2008*
4º Curso Ingeniería Informática Tiempo: 2 horas
Nombre: Apellidos: Grupo:

(3 ptos) Pregunta 2:

Dada la siguiente gramática reducida de la práctica 2 en donde se han eliminado la definición de vectores, de funciones, de algunas de las sentencias de control y de algunas expresiones:

```
Program → module id ; Declarations begin StatementList end id .

Declarations → type id = TypeDenoter ; TypeList Declarations
      | var id IdList : TypeDenoter ; VarList Declarations | ε

TypeList → id = TypeDenoter ; TypeList | ε

VarList → id IdList : TypeDenoter ; VarList | ε

IdList → , id IdList | ε

SimpleType → char | integer |real | boolean

TypeDenoter → SimpleType | id | record id IdList : SimpleType ; FieldList end

FieldList → id IdList : SimpleType ; FieldList | ε

StatementList → Statement ; StatementList | ε

Statement → AssignStm  | WriteStm | ReadStm

AssignStm → LeftValue = Expression

LeftValue → id | id . id

WriteStm → write Expression

ReadStm → read LeftValue

Num → num_integer | num_real

Expression → LeftValue  | ( Expression ) | true | false | Num | Expression * Expression |
          Expression + Expression | Expression or Expression | Expression and Expression
          | not Expression | Expression < Expression | Expression == Expression
```

(1,5) Realiza un **programa en BISON** que reconozca dicha gramática.

Utilizando tu tabla de símbolos y funciones definidas para su construcción y manipulación, implementa la comprobación de tipos en el uso de los registros. En concreto, se deben realizar las siguientes comprobaciones semánticas:

(0,5 ptos.) Se pueden asignar variables de tipo estructura entre sí, siempre que los campos tengan el mismo nombre, tamaño y tipo de dato.

(0,5 ptos.) No se pueden hacer operaciones relacionales o artimético-lógicas sobre las estructuras.

(0,5 ptos.) No se pueden leer ni escribir variables de tipo registro, aunque si los campos de uno en uno.

No es necesario indicar toda la información de la estructura símbolo sólo la que vais a utilizar para implementar esta pregunta. Es imprescindible toda la estructura del fichero de Bison: definiciones de tokens, precedencia de operadores, producciones y acciones semánticas.

(4 ptos) Pregunta 3:

Dada la siguiente gramática que corresponde con una versión muy reducida de la utilizada en la práctica 3:

```
Program → module id ; begin StatementList end id .

StatementList → Statement ; StatementList | ε

Statement → IfStm | ReadStm | AssignStm

AssignStm → id = Exp

IfStm → if BoolExp then StatementList ElseStm end

ElseStm → else StatementList | ε

ReadStm → read Exp

BoolExp → true | false | BoolExp and BoolExp | RelExp

RelExp→ Exp < Exp | Exp == Exp

Exp → id | num | - id | - num
```

a) (1,0 ptos.) Compacta la gramática y modifícala si es necesario para tener en cuenta la precedencia de operadores escribiéndola dentro de la clase del PCCTS MiParser. El operador `and` es asociativo a izquierdas.

b) (1,0 ptos.) Introduce los operadores para la creación del árbol sintáctico de forma que, para el siguiente ejemplo, se genere exactamente el árbol que se muestra a continuación. Modifica la forma de las producciones según sea necesario.

```
module  Ejemplo ;
begin
      read a;
      read b;
      read c;
      if a < -1 and b < 10 and c == 0 then
            read aux;
      else
            error = 1;
      end ;
end Ejemplo .
```

Recorrido en pre-orden del árbol:

```
(module (begin (read a) (read b)(read c) (if (and (and (< a (UNARIO 1 ))(< b 10 ))(== c 0)) (then (read aux ))( else ( = error 1)))))
```

c) (2,0 ptos.) Implementa la función `genera_codigo` en PCCTS para toda la gramática de esta pregunta de forma que genere el código intermedio con la lista de operadores usados en la práctica 3. Por ejemplo para el código anterior, tendríamos:

```
READ  a  NULL  NULL    // inst. 0
READ  b  NULL  NULL
READ  c  NULL  NULL
ASSIG_C  t0  1  NULL
NEG  t1  t0  NULL
IS_LESS  t2  a  t1
ASSIG_C  t3  10  NULL
IS_LESS  t4  b  t3
AND  t5  t2  t4
ASSIG_C  t6  0  NULL
IS_EQUAL  t7  c  t6
AND  t8  t5  t7
NOT  t9  t8  NULL
GOTO_REL  t9  2  NULL // salto a 2 intrs. más abajo
READ aux
GOTO 3 // salto para ignorer el else
ASSIG_C  t10  1  NULL
ASSIG_V error t10
HALT  NULL  NULL  NULL
```