PROCESADORES DE LENGUAJES Examen (Ev. Continua)

Nombre: _____ Apellidos: _____

Ev. continua: ¿Asiste regularmente a clase? (Marcar con una X) SÍ: ___ NO: ___
Ev. continua: ¿Estudia regularmente la asignatura? (Marcar con una X) SÍ: ___ NO: ___

Intérprete (Diseño). Lenguaje KB (3.5 puntos)

KB es un lenguaje para programar bases de conocimiento (bases en adelante). La base consta de 3 secciones: una primera sección para declarar constantes (sección `CONSTANTES`), una segunda para declarar hechos (sección `HECHOS`) y una tercera sección para declarar reglas (sección `REGLAS`). La sintaxis del lenguaje se define en la siguiente gramática:

```
base_de_conocimiento: BASE DE CONOCIMIENTO constantes hechos reglas ;

constantes: CONSTANTES DOSPUNTOS CONSTANTE (COMA CONSTANTE)* ;

hechos: HECHOS DOSPUNTOS hecho (COMA hecho)* ;

hecho: RELACION PARENTESISABIERTO CONSTANTE PARENTESISCERRADO
     | RELACION PARENTESISABIERTO CONSTANTE COMA CONSTANTE PARENTESISCERRADO ;

reglas: REGLAS DOSPUNTOS regla (COMA regla)* ;

regla: antecedente IMPLICA consecuente ;

consecuente: RELACION PARENTESISABIERTO VARIABLE PARENTESISCERRADO ;

antecedente: literal Y antecedente
           | literal
           ;
literal: NO atomo | atomo ;

atomo: atomo_unario | atomo_binario ;

atomo_unario: RELACION PARENTESISABIERTO VARIABLE  PARENTESISCERRADO ;

atomo_binario: RELACION PARENTESISABIERTO VARIABLE COMA CONSTANTE PARENTESISCERRADO
     | RELACION PARENTESISABIERTO VARIABLE COMA VARIABLE_ANONIMA PARENTESISCERRADO ;
```

Los átomos en KB se definen sobre relaciones unarias (átomos unarios ej. `P(_x)`) o binarias (átomos binarios ej. `R(_x,b)`). Los hechos de una base se formalizan con átomos definidos sobre constantes declaradas en la sección `CONSTANTES`. Ejemplo:

```
CONSTANTES:
   a,b,c,d,e

HECHOS:
   P(a), R(a,b), Q(d), Q(c), P(b), R(b,d), R(e,b)
```

Las reglas en KB son implicaciones (`regla: antecedente IMPLICA consecuente`). El consecuente es un átomo unario cuyo parámetro es una variable (ej. `K(_x)`). A diferencia de las constantes, las variables tienen un subrayado como prefijo (ej. `_x`). El antecedente de una regla es una conjunción (`&&`) de literales (átomos o átomos negados) (ej `P(_x) && !R(_x,?)`). La negación se formaliza con el símbolo `!` Estos literales tienen como primer parámetro la variable usada en el consecuente. Si los literales son binarios, el segundo parámetro es una constante o una variable anónima (`?`) (ej. `P(_x) && !R(_x,?) -> S(_x)`). En KB, las relaciones usadas en los consecuentes (1) nunca se usan en los antecedentes y (2) pueden usarse en más de una regla. Ejemplo,

```
REGLAS:
     P(_x) && R(_x,b) -> K(_x),
    P(_x) && !R(_x,?) -> S(_x),
               !Q(_x) -> K(_x)
```

Semánticamente, las reglas de una base permiten inferir nuevos hechos.
Una regla se dice que está instanciada si todas sus variables se sustituyen por constantes. Ejemplos,
`P(a) && R(a,b) -> K(a)` es la instanciación de `P(_x) && R(_x,b) -> K(_x)` con `_x = a`
`P(a) && !R(a,e) -> S(a)` es la instanciación de `P(_x) && !R(_x,?) -> S(_x)` con `_x = a, ? = e`
`!Q(b) -> K(b)` es la instanciación de `!Q(_x) -> K(_x)` con `_x = b`

Un hecho se infiere desde una regla instanciada si el antecedente de ésta está formado por hechos existentes en la base o por negaciones de hechos no existentes en la base.
Por ejemplo, suponiendo las siguientes constantes y hechos:

```
CONSTANTES:
   a,b,c,d,e

HECHOS:
   P(a), R(a,b), Q(d), Q(c), P(b), R(b,d), R(e,b)
```

desde `P(_x) && R(_x,b) -> K(_x)` se infiere `K(a)` instanciando con `_x = a`
desde `P(_x) && !R(_x,?) -> S(_x)` se infiere `S(a)` instanciando con `_x = a, ? = e`
desde `!Q(_x) -> K(_x)` se infiere `K(b)` instanciando con `_x = b`
…

Interpretar una base consiste en inferir todos sus posibles hechos.

SE PIDE:

(1) ¿Cuál es la interpretación de la siguiente base?

```
BASE DE CONOCIMIENTO
CONSTANTES:
      a,b,c,d,e

HECHOS:
      P(a), R(a,b), Q(d), Q(c), P(b), R(b,d), R(e,b)
REGLAS:
  P(_x) && R(_x,b) -> K(_x),
 P(_x) && !R(_x,?) -> S(_x),
            !Q(_x) -> K(_x)
```

Resultado: K(a), K(b), K(e), S(a), S(b)

(2) Gramática atribuida para intérprete de KB. Siguiendo el método explicado en clase, establezca las decisiones relevantes del diseño y a partir de éstas, atribuya la gramática previamente dada. Responda con la versión definitiva de la gramática en la siguiente página (use las dos caras si lo necesita). **Sea claro y legible. Evite tachaduras.** (use los folios adicionales dados por el profesor para las versiones no definitivas)

PROCESADORES DE LENGUAJES Examen (Ev. Continua)

Nombre: _____ Apellidos: _____

DECISIONES:
---

(1) Las reglas se interpretan al vuelo: se calcula el conjunto de constantes que hacen cierto el antecedente de la regla. La instanciación del consecuente con tales constantes permite inferir los hechos de la base.

(2) Para conseguir calcular el conjunto de constantes que hacen cierto un antecedente, es necesario memorizar las constantes y hechos de una base.

(3) Para calcular el conjunto de constantes que hacen cierto un literal positivo en un antecedente,se consulta la memoria de hechos que instancien el literal positivo.
Por ejemplo, suponiendo la memoria de constantes:

a	b	c	d	e

la memoria de hechos:

P(a)	R(a,b)	Q(d)	Q(c)	P(b)	R(b,d)	R(e,b)

y la regla: P(_x) && R(_x,b) -> K(_x)

Los hechos { P(a),P(b) } instancian P(_x), por tanto, cualquier _x en {a,b} hace cierto P(_x)
Los hechos { R(a,b) } instancian R(_x,b), por tanto, cualquier _x en {a} hace cierto R(_x,b)

(4) Para calcular el conjunto de constantes que hacen cierto un literal negativo, se calcula el conjunto de constantes que hacen cierto dicho literal sin la negación (ver 3) y por diferencia con todas las constantes de la base se obtiene el conjunto resultante.
Por ejemplo, suponiendo las constantes y hechos en 3) y la regla !Q(_x) -> K(_x)
Las constantes que hacen cierto Q(_x) es {c,d} (ver 3)
La diferencia {a,b,c,d,e} - {c,d} genera el conjunto resultante: {a,b,e}
por tanto, cualquier _x en {a,b,c} hace cierto !Q(_x)

(5) La conjunción de literales en el antecedente se interpreta como la intersección de los conjuntos de constantes calculados para sus respectivos literales (ver 3 y 4)
Ejemplo, suponiendo las constantes y hechos en 3) y la regla: P(_x) && R(_x,b) -> K(_x), cualquier _x en {a,b} hace cierto P(_x) y cualquier _x en {a} hace cierto R(_x,b). Conclusión: cualquier _x en {a} hace cierto el antecedente P(_x) && R(_x,b).

```
GRAMATICA ATRIBUIDA
-------------------
global:
   memoria de constantes  (2)
   memoria de hechos      (2)

base_de_conocimiento dev inferencias:
   BASE DE CONOCIMIENTO constantes hechos inferencias=reglas ;

constantes: CONSTANTES DOSPUNTOS c:CONSTANTE {almacenar c en memoria de constantes}
                      (COMA d:CONSTANTE {almacenar d en memoria de constantes})* ; (2)

hechos: HECHOS DOSPUNTOS hecho (COMA hecho)* ;

hecho: r:RELACION PARENTESISABIERTO c:CONSTANTE PARENTESISCERRADO {almacenar r(c) en
memoria de hechos}
     | r:RELACION PARENTESISABIERTO c:CONSTANTE COMA d:CONSTANTE PARENTESISCERRADO
       {almacenar r(c,d) en memoria de hechos} ;  (2)

reglas dev inferencias: REGLAS DOSPUNTOS inferencias1=regla
       {almacenar inferencias1 en inferencias}
       (COMA inferencia2=regla {almacenar inferencias2 en inferencias})* ;

regla dev inferencias: consts=antecedente IMPLICA inferencias=consecuente[consts] ; (1)

consecuente[consts] dev inferencias: r:RELACION PARENTESISABIERTO VARIABLE
PARENTESISCERRADO
   {para cada constante c en consts añadir r(c) a inferencias} ;

antecedente dev consts: consts1=literal Y const2=antecedente
                        {consts es la interseccion de consts1 y consts2} (5)
           | consts=literal
           ;
literal dev consts: NO consts=atomo[NEGATIVO] | consts=atomo[POSTIVO] ;

atomo[signo] dev consts: consts=atomo_unario[signo] | consts=atomo_binario[signo] ;

atomo_unario[signo] dev consts: r:RELACION PARENTESISABIERTO VARIABLE PARENTESISCERRADO
   {si signo==POSITIVO entonces
       para cada hecho r(c) en la memoria de hechos añadir c a consts  (3)
    sino
       para cada constante c en la memoria de constantes añadir c a consts si r(c) no
       está incluida en la memoria de hechos (4)
    fsi};

atomo_binario[signo] dev consts:
       r:RELACION PARENTESISABIERTO VARIABLE COMA d:CONSTANTE PARENTESISCERRADO
       {si signo==POSITIVO entonces
          para cada hecho r(c,d) en la memoria de hechos añadir c a consts  (3)
        sino
          para cada constante c en la memoria de constantes añadir c a consts si
          r(c,d) no está en la memoria de hechos
        fsi}
     | RELACION PARENTESISABIERTO VARIABLE COMA VARIABLE_ANONIMA PARENTESISCERRADO
       {si signo==POSITIVO entonces
          para cada hecho r(c,d) en la memoria de hechos añadir c a consts  (3)
        sino
          para cada constante c en la memoria de constantes añadir c a consts si
          r(c,d) no está en la memoria de hechos para alguna constante d en la memoria
          de constantes  (4)
        fsi};
```

PROCESADORES DE LENGUAJES Examen (Ev. Continua)

Nombre: _____ Apellidos: _____

Intérprete (Implementación) (1.5 puntos).

(a) Implemente con métodos de clase Listener Antlr4 las siguientes reglas de la gramática atribuida de un intérprete. Responda en el espacio dejado a tal fin con la versión definitiva de la implementación. Evitará tachaduras y errores haciendo uso de los folios adicionales dados por el profesor como borrador:

Regla:
```
ruptura[centinela] devuelve centinela2: RUPTURA PUNTOYCOMA
      {si centinela es igual a hay_que_interpretar entonces
         centinela2 = no_hay_que_interpretar
       si no
         centinela2 = centinela
       fsi };
```

Implementación:

```
public void exitRuptura(Anasint.RupturaContext ctx, Centinela centinela,
                        Centinela centinela2) {
    if (centinela.valor().equals("hay_que_interpretar"))
       centinela2.asignar("no_hay_que_interpretar");
    else
       centinela2.asignar(centinela.valor());
}
```

Regla:
```
programa :  {generar codigo declaracion clase}
               PROGRAMA variables instrucciones
            {generar codigo fin main
             generar codigo fin clase} ;
```

Implementación:

```
public void enterProg(Anasint.ProgContext ctx) {
    generar_codigo_declaracion_clase ();  }

public void exitProg(Anasint.ProgContext ctx) {
    generar_codigo_fin_main(); generar_codigo_fin_clase(); }
```

(b) Implemente con método de clase Visitor Antlr4 la siguiente regla de la gramática atribuida de un intérprete. Responda en el espacio dejado a tal fin con la versión definitiva de la implementación. Evitará tachaduras y errores haciendo uso de los folios adicionales dados por el profesor como borrador:

Regla:

```
expresion[estado] dev valor:
       valor1=expresion[estado] MAS valor2=expresion[estado] {valor=valor1+valor2}
     | valor1=expresion[estado] MENOS valor2=expresion[estado] {valor=valor1-valor2}
     | valor1=expresion[estado] POR valor2=expresion[estado] {valor=valor1*valor2}
     | PA valor=expresion[estado] PC
     | IDENT {valor=consultar el valor de IDENT en estado}
     | NUMERO {valor=NUMERO}
     ;
```

Implementación:

```
public Integer visitExpresion(Anasint.ExpresionContext ctx,
                              Estado estado) {
    Integer valor,valor1,valor2;
    if (ctx.getChildCount()==1) {
        if (ctx.NUMERO() != null)
           valor=Integer.valueOf(ctx.NUMERO().getText());
        else
            valor=consultar_valor_en_estado(ctx.IDENT().getText());
    }
    else
        if (ctx.MAS()!=null){
           valor1=visitExpresion(ctx.expresion(0),estado);
           valor2=visitExpresion(ctx.expresion(1),estado);
           valor=valor1+valor2;
        }
        else
           if (ctx.MENOS()!=null){
              valor1=visitExpresion(ctx.expresion(0),estado);
              valor2=visitExpresion(ctx.expresion(1),estado);
              valor=valor1-valor2;
           }
           else
              if (ctx.POR()!=null){
                 valor1=visitExpresion(ctx.expresion(0),estado);
                 valor2=visitExpresion(ctx.expresion(1),estado);
                 valor=valor1*valor2;
              }
              else
                 valor=visitExpresion(ctx.expresion(0),estado);

        }

    return valor;
}
```