PROCESADORES DE LENGUAJES Examen (Primera Conv.)

Nombre: _____ Apellidos: _____

Sintáctico (Diseño). Lenguaje LPROC (2 puntos)

Supongamos un lenguaje llamado LPROC diseñado para expresar procedimientos como el mostrado en el siguiente ejemplo.

```
buscar(entero e, vector(entero)[1..n]  v) devuelve (booleano, entero)

variables locales:
   booleano resultado;
   entero elemento, i;

instrucciones:
   (resultado, i)=(falso, 1);
    mientras (resultado == falso y i<=numeroElementos(v)) hacer
       elemento = v[i];
       si (elemento == e) entonces
          resultado = cierto;
       sino
          i=i+1;
       finsi
    finmientras
    devuelve (resultado,i);
fin
```

El perfil de un procedimiento consta de nombre, parámetros y resultados. Internamente el procedimiento se estructura con un conjunto de variables locales y una secuencia de instrucciones. Los tipos elementales en LPROC son el tipo lógico y el tipo entero. El único tipo no elemental es el tipo vector (de elementos enteros o lógicos). La declaración de un vector siempre incluirá el tipo de sus elementos y el rango de sus índices. LPROC tiene 5 tipos de instrucciones: asignaciones simples, asignaciones múltiples, iteraciones, condicionales y devolución de resultados. Las expresiones pueden incluir llamada a funciones.

SE PIDE: Diseño de gramática independiente de contexto para LPROC. Responda con la versión definitiva de la gramática en lo que queda de esta página y en la página de atrás. **Sea claro y legible. Evite tachaduras.** (use los folios adicionales dados por el profesor para las versiones no definitivas).

Respuesta:

```
procedure: perfil variableslocales instrucciones


perfil: IDENT PA (parametros)? PC DEV PA (tipos)? PC

parametros : parametro COMA parametros
           | parametro

parametro : tipo IDENT

tipos: tipo COMA tipos
     | tipo

variableslocales : VARIABLES LOCALES DP variables

variables :  decl_vars variables
          |

decl_vars : tipo vars PyC
```

```
 vars: IDENT COMA vars
     | IDENT

instrucciones: INSTRUCCIONES DP (instruccion)* FIN

instruccion : asignacion
            | iteracion
            | seleccion
            | retorno

asignacion : asignacionsimple
           | asignacionmultiple

asignacionsimple: IDENT ASIG expr PyC

asignacionmultiple: PA vars PC ASIG PA exprs PC PyC

iteracion: MIENTRAS PA expr PC HACER bloque FINMIENTRAS

seleccion:  SI PA expr PC ENTONCES bloque SINO bloque FINSI
         |  SI PA expr PC ENTONCES bloque FINSI

bloque : (instruccion)*

retorno: DEV PA exprs PC PyC

tipo: VECTOR PA tipo PC CA indice RANGO indice CC
    | ENTERO
    | BOOLEANO

indice : NUMERO | IDENT

exprs : expr COMA exprs
      | expr

expr : expr_booleana | expr_entera

expr_booleana: expr_booleana (Y|O) expr_booleana
    | NO expr_booleana
    | expr_entera (MAYOR|MENOR|MAYORIGUAL|MENORIGUAL) expr_entera
    | expr (IGUAL|DISTINTO) expr
    | IDENT CA expr_entera CC
    | IDENT PA exprs PC
    | IDENT
    | FALSO
    | CIERTO

expr_entera : expr_entera (MAS|MENOS|POR|DIV) expr_entera
        | IDENT CA expr_entera CC
        | IDENT PA exprs PC
        | IDENT
        | NUMERO
```

PROCESADORES DE LENGUAJES Examen (Primera Conv.)

Nombre: _____ Apellidos: _____

Semántico (Diseño). Lenguaje KB (3 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`). Ejemplo,

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

El consecuente de una regla es un átomo unario cuyo parámetro es una variable (ej. `S(_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 (**restricción 1**). En KB, las relaciones usadas en los consecuentes nunca se usan en los antecedentes (**restricción 2**).

SE PIDE:
Diseño de un analizador semántico del lenguaje KB capaz de detectar el incumplimiento de alguna de las restricciones propuestas. Atribuya para ello la gramática previamente dada. Responda detrás de esta página con la versión definitiva de esta gramática (no se piden las decisiones). **Sea claro y legible. Evite tachaduras.** (use los folios adicionales dados por el profesor para las versiones no definitivas)

Nombre: _____ Apellidos: _____

DECISIONES: (NO SE PIDEN)
---

1) Para comprobar la restricción 1, se acumula el conjunto de variables del antecedente y se compara cada una de estas variables con la variable del consecuente. Si no coincide se producirá error.

2) Para comprobar la restricción 2, se acumula el conjunto de relaciones de los antecedente y el conjunto de relaciones de los consecuentes.
Al procesar todas las reglas, se busca si la intersección de ambos conjuntos no es vacía en cuyo caso se producirá error.

GRAMATICA ATRIBUIDA

```
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
   rels_ant,rels_con=regla
     (COMA rels_a,rels_c=regla {almacenar rels_a en rels_ant y rels_c en rels_con)*
   { si la intersección entre rels_ant y rels_con no es vacia entonces
        Error por no cumplir la restricción 2
     fsi };

regla dev rels_ant, rels_con:
   rels_ant,vars=antecedente IMPLICA rels_con=consecuente[vars] ;

consecuente[vars] dev rels: RELACION {añadir RELACION a rels}
                            PARENTESISABIERTO VARIABLE PARENTESISCERRADO
                  { si alguna variable en vars no coincide con VARIABLE entonces
                       Error por no cumplir la restricción 1
                    fsi } ;

antecedente dev rels,vars: r,v=literal Y rels,vars=antecedente
                               {añadir v a vars, añadir r a rels}
              | rels,vars=literal
              ;
literal dev r,v: NO r,v=atomo | r,v=atomo ;

atomo dev r,v: r,v=atomo_unario | r,v=atomo_binario ;

atomo_unario dev r,v: r:RELACION PARENTESISABIERTO v:VARIABLE PARENTESISCERRADO ;

atomo_binario dev r,v:
      r:RELACION PARENTESISABIERTO v:VARIABLE COMA CONSTANTE PARENTESISCERRADO
    | r:RELACION PARENTESISABIERTO v:VARIABLE COMA VARIABLE_ANONIMA PARENTESISCERRADO ;
```

PROCESADORES DE LENGUAJES Examen (Primera Conv,)

Nombre: _____ Apellidos: _____

Intérprete (Implementación). Lenguaje KB (2 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`).
Ejemplo:

```
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)
```

Los átomos en KB se definen sobre relaciones unarias (átomos unarios ej. `P(_x)`) o binarias (átomos binarios ej. `R(_x,b)`). Las reglas en KB son implicaciones. El consecuente de una regla es un átomo unario cuyo parámetro es una variable (ej. `S(_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 nunca se usan en los antecedentes.

Las reglas de una base permiten inferir nuevos hechos. Estas inferencias se consiguen instanciando sus reglas. 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.

La interpretación de una base consiste en inferir sus hechos. En la base de ejemplo, la interpretación infiere los siguientes hechos: `K(a), K(b), K(e), S(a), S(b)`

El siguiente diseño corresponde a un intérprete de KB.

```
OBJETIVO: Interprete que infiere todos los hechos de una base.

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};
```

SE PIDE:

Implementación de las declaraciones globales (`memoria de constantes` y `memoria de hechos`) y de las reglas `reglas` y `atomo_binario`. Responda con la versión definitiva de dicha implementación 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).

Nombre: _____ Apellidos: _____

```
//global:
//   memoria de constantes  (2)
//   memoria de hechos      (2)
Set<String>memoria_constantes=new HashSet<>();
Map<String,Set<String>> memoria_hechos_unarios=new HashMap<>();
Map<String,Set<Pair<String,String>>> memoria_hechos_binarios=new HashMap<>();


//reglas dev inferencias: REGLAS DOSPUNTOS inferencias1=regla {almacenar
                          inferencias1 en inferencias}
//       (COMA inferencia2=regla {almacenar inferencias2 en inferencias})* ;
//
@Override
public Set<Object> visitReglas(Anasint2.ReglasContext ctx) {
    Set<Object>inferencias=new HashSet<>();
    int i=0;
    Anasint2.ReglaContext regla=ctx.regla(i);
    while (regla!=null){
        inferencias.addAll(visitRegla(regla));
        i++;
        regla=ctx.regla(i);
    }
    return inferencias;
}


//
//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};
public Set<Object> visitAtomo_binario(Anasint2.Atomo_binarioContext ctx,
                                      boolean signo) {
    Set<Object>consts=new HashSet<>();
    String r=ctx.RELACION().getText();
    if (ctx.CONSTANTE()!=null){
        String d=ctx.CONSTANTE().getText();
        if (signo)
            for(Pair<String,String>aux:memoria_hechos_binarios.get(r)) {
                if (aux.b.equals(d))
                    consts.add(aux.a);
            }
        else
            for(String c:memoria_constantes) {
                if (!memoria_hechos_binarios.get(r).stream().
                        anyMatch(p->p.a.equals(c) && p.b.equals(d)))
```

```
                        consts.add(c);
                }
        }
        else{
            if (signo)
                for(Pair<String,String>aux:memoria_hechos_binarios.get(r))
                        consts.add(aux.a);
            else
                for(String c:memoria_constantes)
                    for(String d:memoria_constantes)
                        if (!memoria_hechos_binarios.get(r).stream().
                                anyMatch(p->p.a.equals(c) && p.b.equals(d)))
                            consts.add(c);
        }

        return consts;
    }
}
```

PROCESADORES DE LENGUAJES Examen (Primera Conv,)

Nombre: _____ Apellidos: _____

Compilador (Diseño). Lenguaje Q (3 puntos).

Q es un lenguaje para programar consultas desde un conjunto de reglas. El programa Q consta de 3 secciones: una primera sección para declarar hechos (sección `HECHOS`), una segunda para declarar reglas (sección `REGLAS`) y una tercera sección para programar una secuencia de consultas (sección `CONSULTAS`).
Ejemplo:

```
PROGRAMA

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

REGLAS:
  K(_x) <- P(_x) && R(_x);
  S(_x) <- Q(_x) && R(_x);
  T(_x) <- Q(_x);

CONSULTAS:
   K; // la consulta muestra por consola K(b)
   T; // la consulta muestra por consola T(c) T(d)
   S; // la consulta muestra por consola S(d)
   S; // la consulta muestra por consola S(d)
```

Los hechos en Q son átomos unarios con parámetro constante ej. `P(a)`. Las reglas en Q son implicaciones. El consecuente de la regla (parte derecha de la implicación) es un átomo unario cuyo parámetro es una variable (ej. `S(_x)`). A diferencia de las constantes, las variables tienen un subrayado como prefijo (ej. `_x`). El antecedente de la regla (parte izquierda de la implicación) es una conjunción (`&&`) de átomos unarios cuyo parámetro es la variable usada en el consecuente. Las relaciones usadas en los consecuentes (b) nunca se usan en los antecedentes y (b) sólo aparecen en una única regla en el programa.

Las reglas permiten inferir nuevos hechos. Estas inferencias se consiguen instanciando reglas. Una regla se dice que está instanciada si su variable se sustituye por una constante. Ejemplo, `K(b) <- P(b) && R(b)` es la instanciación de `K(_x) <- P(_x) && R(_x)` con `_x = b`. Un hecho se *infiere* desde una regla instanciada si el antecedente de ésta está formado por hechos. Ejemplo, `K(b)` se infiere desde la instanciación de la regla `K(_x) <- P(_x) && R(_x)` en el programa de ejemplo.

Las consultas en Q se formalizan mediante una secuencia de nombres de relaciones usadas en los consecuentes de las reglas ej. `K; T; S; S;`. La evaluación de estas consultas mostrará por pantalla todos los hechos inferidos correspondientes. Ver el resultado de las consultas en los comentarios del programa de ejemplo.

SE PIDE: Suponiendo el programa de ejemplo, establezca a continuación una traducción basada en método Java (ej. `public static Set<String> P()`) de los hechos de la relación `P`.
Respuesta:

```java
public static Set<String> P(){
   Set<String>r=new HashSet<>();
   r.add("a");
   r.add("b");
   return r;
}
```

SE PIDE: Suponiendo el programa de ejemplo, establezca a continuación una traducción basada en método Java (ej. `public static Set<String> K()`) de la regla `K(_x) <- P(_x) && R(_x);`
Respuesta:

```
public static Set<String> K(){
   Set<String>r=new HashSet<>();
   r.addAll(P());
   r.retainAll(R());
   return r;
}
```

Suponiendo la siguiente gramática del lenguaje Q:

```
programa: PROGRAMA  hechos reglas consultas;

hechos: HECHOS DOSPUNTOS (hechos_relacion)* ;

hechos_relacion: hecho (COMA hecho)* PUNTOYCOMA ;

hecho: RELACION PARENTESISABIERTO CONSTANTE PARENTESISCERRADO ;

reglas: REGLAS DOSPUNTOS (regla)* ;

regla: consecuente IMPLICA antecedente PUNTOYCOMA;

consecuente: RELACION PARENTESISABIERTO VARIABLE PARENTESISCERRADO ;

antecedente: RELACION PARENTESISABIERTO VARIABLE PARENTESISCERRADO (Y antecedente)? ;

consultas: CONSULTAS DOSPUNTOS (consulta)* ;

consulta: RELACION PUNTOYCOMA;
```

SE PIDE:

Diseño de un compilador del lenguaje Q a lenguaje Java. Siguiendo el método explicado en clase, establezca (a) las decisiones relevantes del diseño y a partir de éstas, (b) atribuir la gramática dada para completar el diseño. Suponga un procedimiento `escribir("texto…")` para escribir la traducción. Responda con la versión definitiva de este diseño 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 (Primera Conv,)

Nombre: _____ Apellidos: _____

DECISIONES
---

1) El conjunto de hechos de una relación se traduce a una función Java.
   El tipo devuelto por dicha función representa el conjunto de constantes desde las que se definen los hechos correspondientes y el cuerpo de dicha función es el código Java que construye dicho conjunto
   (ver ej. de traducción de los hechos de P).

2) De acuerdo con las restricciones impuestas a las reglas, cada regla también se puede traducir a una función Java. El tipo devuelto por dicha función es el conjunto de constantes inferidos para el consecuente de la regla.
   El cuerpo de dicha función es el código Java que construye la intersección de los conjuntos de constantes devueltos por las funciones que implementan el antecedente de la regla.

   (2,a) La cabecera de la función se escribe al procesar el consecuente.
   (2,b) Para generar correctamente el código del cuerpo de la función debe distinguir el primer átomo del antecedente del resto de átomos.
   (2,c) La terminación de la función se escribe al acabar de procesar la regla.

3) Las consultas se localizan en la funcion Java main(). Cada consulta es una llamada a la función correspondiente a la regla del consecuente consultado almacenando el resultado en un conjunto que se recorre para imprimir por pantalla el texto correspondiente a cada inferencia.

GRAMATICA ATRIBUIDA

```
programa: { escribir("import java.util.*;")
            escribir("public class Programa {") }
    PROGRAMA hechos reglas consultas;
          { escribir("}") }

hechos: HECHOS DOSPUNTOS (hechos_relacion)* ;

hechos_relacion: hecho[PRIMER HECHO]
           (COMA hecho[NO PRIMER HECHO])* PUNTOYCOMA ;
           {  escribir("      return r;")
              escribir("   }") }              (1)

hecho[centinela]: RELACION PARENTESISABIERTO CONSTANTE PARENTESISCERRADO ;
   {
      si (centinela == PRIMER HECHO) entonces                    (1)
         {escribir("   public static Set<String> RELACION(){")
          escribir("      Set<String>r=new HashSet<>();")}
      fsi
      escribir("   r.add(CONSTANTE);")  (1)
   }

reglas: REGLAS DOSPUNTOS (regla)* ;

regla: consecuente IMPLICA antecedente[PRIMER ATOMO ANTECEDENTE] PUNTOYCOMA;
  {  escribir("   return r;")  (2.c)
     escribir("}") }

consecuente: rel:RELACION PARENTESISABIERTO VARIABLE PARENTESISCERRADO ;
   { escribir("   public static Set<String>" +rel+" (){")
     escribir("      Set<String>r=new HashSet<>();") }
```

```
antecedente[centinela]:
   RELACION PARENTESISABIERTO VARIABLE PARENTESISCERRADO
   { si (centinela==PRIMER ATOMO ANTECEDENTE) entonces                (2,b)
        escribir("      r.addAll(RELACION());")
     sino
        escribir("      r.retainAll(RELACION());")
     fsi }
   (Y antecedente[NO PRIMER ATOMO ANTECEDENTE])? ;

consultas: CONSULTAS DOSPUNTOS
   { escribir("  public static void main(String args[]){")
     escribir("      Set<String>s=new HashSet<String>();")}  (3)
       (consulta)* ;
   { escribir("   }") }

consulta: r:RELACION PUNTOYCOMA;
   {escribir("   s.clear();")  (3)
    escribir("   s.addAll("+r+"());")
    escribir("   s.stream().forEach(e->System.out.print(r+"("+e+") "));")   (3)
    escribir("   System.out.println();"); }
```