PREGUNTAS

1) Considere la siguiente gramática `CFG` teórica `G` escrita en notación de "*papel-y-lápiz*". Donde se tiene que: `X` y `Y` son no terminales; `X` es el símbolo de arranque, y `a` y `b` son no terminales.

```
X → Y
X → a
Y → b Y
Y → X
```

a) Describa cuál es el lenguaje generado por `G`. Sea preciso y conciso.
b) Reescriba `G` usando gramáticas `ANTLR` (estilo `EBNF`)

2) Escriba una gramática en `ANTLR` que genere hileras que representan listas de números enteros anidadas a cualquier nivel. Por ejemplo: `[]`, `[[1, [2], []]`, `[[[]]], 3]`. Llame `Listas` a su gramática.

Llamemos `#List` al contexto que visita una lista así. Escriba el visitador (sólo el método) usando Java8 que imprima la altura de una lista así.

```
@Override void visit(ListasParser.ListContext ctx){
  // Implemente aquí…
}
```

2) Queremos tener un tipo de `while` como expresión en `KoKoslan`. Por ejemplo permitir:

```
let n = 10
let s = 0
while ( n > 0 ){
  let s = s + n
  let n = n - 1
}
then print(s);
```

Tiene sintácticamente la forma `while (…){…} then …` siendo en cada caso … una expresión cualquiera (incluso un `while`). Excepto en el cuerpo del while pueden venir `let`s.
Semánticamente funcionaría similar a un `while` usual excepto que cuando el `while` termina se ejecuta la expresión del `then`.
Escriba los cambios requerido en su gramática `ANTLR` (la del proyecto `Kokoslan`) que permita *parsear* un `while` así definido. Escriba sólo el/los cambio(s) requeridos, no toda la gramática.

3) Considere la sumatoria: $\sum_{n=1}^{\infty} nx^{n-1}$ (para $|x| < 1$). Escriba un método en `Java8` `double sum(int max, double epsilon)` que usando FP aproxime la sumatoria. El cálculo termina cuando se alcanzan `max` iteraciones ó el término general de la sumatoria es menor en valor absoluto que `epsilon`. Condiciones: El número de

multiplicaciones debe ser $O(max)$. No use recursión, ni `forEach`. Se califica eficiencia y claridad de la solución, a criterio del profesor. Si no usa `FP` recibe 0. No valide `x` asuma cumple lo requerido.

4) Lo mismo anterior pero en `Kotlin-FP`. Signatura: `fun sum(max:Int, epsilon: Double): Double`.

5) Lo mismo anterior en `Prolog`. Predicado `sum(+Max, +Epsilon, -Res)`. Siendo `Res` el cálculo de la sumatoria y `Max` y `Epsilon` como antes.

6) Escriba un método en `Java8` `double gcd(List<Integer> nums)` que usando programación funcional (`FP`) encuentre el máximo común divisor (`gcd`) de los números en `nums`. Asuma que no hay negativos y que la lista no está vacía. No use recursión, ni `reduce`, ni `forEach`. Se califica eficiencia y claridad de la solución, a criterio del profesor. Si no usa `FP` o no cumple lo pedido recibe 0 aunque esté correcta su respuesta funcionalmente.

7) Implemente lo mismo el ejercicio anterior pero en `Kotlin-FP`. Signatura `fun gcd(nums: List<Int>): Int`.

8) Implemente en `Prolog` el mismo algoritmo de la pregunta anterior por medio de un predicado `gcdList(+Nums, -Gcd)`. Dada es `Nums` una lista de enteros positivos y no vacía. `Gcd` es el resultado. Haga dos versiones a) Recursiva de cola y b) No recursiva. Valen las condiciones pedidas de eficiencia y claridad anteriores. Sugerencia: `SWI-Prolog` ya tiene una función `gcd`:
Ejemplo:

```
?- D is gcd(4, 6).
D = 2
```

Considere la interfaz `Student` y `enum Career`.

```
enum Career {
    CS,   // Computer Science
    MATH, // Mathematics
    PHI,  // Phisycs
    CHEM  // Chemistry
    //...

}
interface Student {
    String getId();
    String getName();
    Career getCareer();
    double getCareerGrade();
}
```

Usando (principalmente) `Collectors`, escriba en `Java8` un método `Map<Career, Double> averageByCareer(List<Student> students)` que dada una lista de estudiantes, retorne un mapa asocie las carrera con el promedio de notas de estudiantes en esa carrera. Sólo se consideran las carreras que ocurran en la lista.

9) Usando sólo `append`, escriba un predicado `noJuntos(+L, -X, -Y)` que dada una lista `L` retorne en `X` y `Y` pares de elementos en `L` que no ocurren juntos, respetando el orden en que ocurren en `L`. Solo puede usar `append` (ningún otro predicado, ni recursión). El algoritmo falla si no hay elementos no juntos.

Ejemplos de dos corridas:

```
?- noJuntos([a, b, c, d], X, Y).

X = a
Y = c;
X = a
Y = d;
X = b
Y = c;
false.

?- noJuntos([a, b], X, Y).
false.
```

10) Escriba un predicado `solve(+N, -S)` que en `S` retorne todas las soluciones `x`, `y` de la ecuación `x+y=N` donde `N` es un entero no negativo. No use recursión use backtracking.
Ejemplo de uso:

```
:-solve(3, S)

S=0+3
S=1+2
S=2+1
S=3+0
```

Escriba `solveList(+N, -L)` que recolecte en `L` todas esas soluciones generadas por `solve`.

11) ¿Qué hace el siguiente predicado `foo(+L, +N)`? Resuelva sin correrlo en Prolog.

```
foo(L, N) :- member(X, L), X=N, write(X), nl, fail.
```