Trabajo Práctico N° 1

Introducción a los Lenguajes y Paradigmas de Programación

Ejercicio 1

Considerando la gramática del lenguaje Kotlin que se encuentra disponible en https://kotlinlang.org/docs/reference/grammar.html y además es posible descargarla de aquí.

a) ¿Es correcto afirmar que los bloques del código en el cuerpo de un if pueden ser los mismos que los del cuerpo de una función?

b) Responder si las siguientes sentencias son sintácticamente correctas o no, fundamentando la respuesta de acuerdo a lo que observe en la gramática del lenguaje.

I. if(1==0); else println("else")
II. if(1==0) else println("else")
III. if(1==0) println("else")
IV. if(1==0) println("if") else
V. if (1==1) println("if")
VI. if (1==1)
VII. if (1==1);
VIII. if(1==0) println("if") else println("else")
IX. if(1==0) println("if"); else println("else")
X. if(1==0) println("if"); else if(1==0) println("else_if"); else println("else_else")
XI. if(1==0) println("if") else;

c) Considerar una función que contiene todas las sentencias del inciso anterior. Teniendo en cuenta el análisis hecho previamente, indicar si la función es aceptada por el lenguaje.

d) Si esa misma función se escribiera en Java (realizando el reemplazo de “println()” por “System.out.println()”), ¿sería aceptada por el lenguaje? ¿Qué diferencias puede identificar en cuanto a la definición de su gramática?

e) Indicar cuál lenguaje le parece más restrictivo y cuál más flexible en cuanto a las diferentes posibilidades que ofrecen para escribir las sentencias de selección. ¿Cuál tiene mayores posibilidades de generar un programa con funcionamiento inesperado?

Ejercicio 2

Dada la gramática de Python 3.X, Identificar todas las reglas (desde el nivel superior hasta el nivel de la regla “expression”) que permiten detectar la siguientes instrucciones:

i)	a=1 def f(x): x=2 f(a)	ii)	a = lambda x:x+2 print(a(1))

iii)	def f(x,y):
	  a = x(y)
	def g(z):
	  print (z+1)
	b = 5
	f(g,b)
		iv)	def f():
			  def g(z):
			  print (z+1)
			  return g
			c = f()
			c(3)

Ejercicio 3

Complete el cuadro referido a las verificaciones que realiza un lenguaje de programación.

- ¿Cuáles de estas verificaciones se refieren a la sintáctica, cuáles a la semántica y cuáles a la pragmática del lenguaje?
- ¿Cuáles de estas verificaciones se pueden hacer en tiempo de compilación y cuáles se pueden hacer en tiempo de ejecución? Justifique brevemente cada una en el cuadro.

	Sintáctica (SX), Semántica (SM) o Pragmática (P)		**Compilación (C) o ejecución (E)**	
Los subíndices de los arreglos nunca pueden exceder los límites definidos para los mismos.				
El tipo de los parámetros reales sea siempre exactamente el mismo que el de los parámetros formales.				
La concatenación de dos variables de tipo cadena de caracteres no exceda nunca los 255 caracteres.				
No pueda existir un comienzo de comentario (/*) en medio de un comentario ya empezado.				
En la instrucción “a = b;” si el código ASCII del símbolo de asignación es válido				
En dos programas producidos por diferentes compiladores/intérpretes, la instrucción “a = b;” se ejecuta con diferente velocidad (en las mismas condiciones de ejecución)				

Ejercicio 4

Indicar para cada uno de los siguientes códigos:

a) Cuál es la funcionalidad del programa

b) El o los paradigmas que se encuentran presentes, fundamentando la respuesta en base a los conceptos relacionados a paradigmas que puede identificar en los programas.

c) Indicar las líneas donde se encuentren:

i. Definiciones de funciones (indicando la función que se define)

ii. Aplicaciones de funciones (indicando la función que se aplica y el parámetro de la función si lo tuviere)

iii. Afirmaciones

iv. Reglas

v. Preguntas

vi. Invocación a métodos (indicando el objeto que recibe el método, el método invocado y los parámetros del método si los tuviere)

Codigo 1

```
const a = [1,2,3,4,5,6]
let r = []
for (let i=0;i<6;i++) {
if (a[i] % 2 == 0) {
   r = [...r,a[i]]
}
}
```

Codigo 2

```
console.log([1,2,3,4,5,6].filter(x => x % 2 == 0))
```

Codigo 3

```
suma([],0).
suma([H|T],R) :-
  suma(T, N),
  R is H+N.
```

Codigo 4

```
(define (sum elem)
  (if
    (null? elem)
    0
        (+ (car elem) (sum (cdr elem)))
  )
)
(sum (list 1 2 3 4))
```

Ejercicio 5

Teniendo el problema: “dada una lista de números, contar la cantidad de elementos hasta encontrar un cero” y las siguientes soluciones en cada uno de los siguientes paradigmas:

#	Imperativo	#	Lógico	Funcional

1
2
3
4
5
6
7
8
9
10
 	
11
	func main() {
	  var lista = []int{3, 7, 2, 0, 5}
	  var contador int = 0
	  for i := 0; i < len(lista); i++ {
	  if lista[i] == 0 {
	  break
	  }
	  contador = contador + 1
	  }
	  fmt.Printf("La cantidad es: %d\n", contador)
	}
		1
		2
		3
		4
		5
		6
			contar([], 0).
			contar([0 | _], 0).
			contar([_ | Cola], C) :-
			  contar(Cola, CA),
			  C is CA + 1.
			contar([3, 7 , 2 , 0, 5], contador).
				(defun contar (lista)
				  (if (null lista)
				  0
				  (if (zerop (car lista))
				  0
				  (+ 1 (contar (cdr lista))))
				  ))
				(contar '(3 7 2 0 5))

Identificar en cada uno de los programas, línea por línea los no-terminales que se reconocen del lado izquierdo según las reglas del BNF de los lenguajes Go (imperativo), Prolog (lógico), Common Lisp (Funcional).

Ejercicio 6

Para los programas del ejercicio 4 y el siguiente programa en un lenguaje orientado a objetos, realizar una simulación de la ejecución explicando la ejecución de cada línea. En el caso del lenguaje

orientado a objetos, indicar detalladamente cual es el objeto que recibe el método, cuál es el método y cuales son los parámetros.

#	Orientado a Objetos
1	Object subclass: Contador [
2	| lista |
3	Contador class >> newWith: aList [
4	^ self new initializeWith: aList.
5	]
6	initializeWith: aList [
7	lista := aList.
8	]
9	contarHastaCero [
10	| cuenta |
11	cuenta := 0.
12	lista do: [:num |
13	num = 0 ifTrue: [^ cuenta].
14	cuenta := cuenta + 1.
15	].
16	^ cuenta.
17	]
181	]
9	| micontador resultado |
20	micontador := Contador newWith: #(3 7 2 0 5).
21	resultado := micontador contarHastaCero.
22	Transcript show: 'Cuenta: ', resultado printString; nl.

Ejercicio 7

Para el siguiente programa, identificar línea por línea si se está utilizando un paradigma imperativo, funcional, lógico, orientado a objetos u orientado a eventos, indicando la ocurrencia de los siguientes elementos: definición de clase, declaración de objeto, declaración de función, declaración de método, declaración de atributo, declaración de variable, asignación a variable, asignación a atributo, selección, iteración, condición, expresión, parámetro, invocación a función, invocación a método, establecimiento de evento,

```
1   class A {
2     constructor(x) {
3           this.x = x;   }
4     calcular() {
5           const sumacuadrados = [1, 2, 3, 4].
6                 filter(num => num % 2 === 0).
7                       map(even => even * even).
8                             reduce((x, y) => x + y);
9           console.log(`Resultado del cálculo: ${sumacuadrados}`);
10          setTimeout(() => {this.res(sumacuadrados);}, 0);    }
11    res(resultado) {
```

```
12          console.log(`Resultado ${resultado}`);  }    }
13  function ejecutar() {
14    let a1 = new A(3);
15    a1.calcular();  }
16  ejecutar();
```

Ejercicio 8

Para las siguientes implementaciones en Python del cálculo del factorial de un número:

1
```
def suma_lista(numeros):
  if not numeros:
    return 0
  else:
    return numeros[0] + suma_lista(numeros[1:])
numeros = [1, 2, 3, 4, 5]
resultado = suma_lista(numeros)
```

2
```
class SumaLista:
  def __init__(self, numeros):
    self.numeros = numeros
    def calcular_suma(self):
      suma = 0
      for numero in self.numeros:
        suma += numero
      return suma
numeros = [1, 2, 3, 4, 5]
o1 = SumaLista(numeros)
resultado = o1.calcular_suma()
```

3
```
def suma_lista(numeros):
  suma = 0
  for numero in numeros:
    suma += numero
  return suma
numeros = [1, 2, 3, 4, 5]
resultado = suma_lista(numeros)
```

4
```
def suma_lista(numeros, continuacion):
  if not numeros:
    continuacion(0)
  else:
    cabeza = numeros[0]
    cola = numeros[1:]
    suma_lista(cola, lambda
resultado_restante: continuacion(cabeza +
resultado_restante))
def manejar_resultado(resultado):
  print(f"La suma total es: {resultado}")
numeros = [1, 2, 3, 4, 5]
suma_lista(numeros, manejar_resultado)
```

5
```
import queue
eventos = queue.Queue()
suma_acumulada = 0
def registrar_evento(numero):
  def evento_sumar():
    global suma_acumulada
    suma_acumulada += numero
    eventos.put(evento_sumar)
def procesar_eventos():
  while not eventos.empty():
  evento = eventos.get()
  evento()
```

6
```
import threading
def suma_numero(numero, resultado, idx):
  resultado[idx] = numero
def suma(numeros):
  resultados = [0] * len(numeros)
  hilos = []
  for i, numero in enumerate(numeros):
    hilo =
threading.Thread(target=suma_numero,
args=(numero, resultados, i))
    hilos.append(hilo)
    hilo.start()
```

	numeros = [5, 10, 15, 20, 25]
	for numero in numeros:
	  registrar_evento(numero)
	procesar_eventos()
	print(f"Suma total: {suma_acumulada}")
			for hilo in hilos:
			  hilo.join()
			return sum(resultados)
			numeros = [1, 2, 3, 4, 5]
			resultado = suma(numeros)

7	def suma_lista():
	  suma = lambda numeros: 0 if not
	numeros else numeros[0] + suma(numeros[1:])
	  return suma
	print(suma_lista()([1, 2, 3, 4, 5]))		

a) Indicar a cuál paradigma se acerca cada solución, fundamentando la respuesta por medio de la presencia de los elementos distintivos de cada paradigma en cada programa.

b) En cada solución indicar cómo se reemplaza la iteración que resuelve la suma por otro mecanismo si es que esto sucede.

c) Identificar cuáles paradigmas se aplican de forma pura (sin hacer uso de elementos de otros paradigmas)

d) Identificar en cada solución si las funciones se utilizan como de primera, segunda o tercera clase.

e) Para cada solución, en caso que exista, identificar al menos una regla del BNF que no se aplique en las demás soluciones.

Ejercicio 9

Teniendo en cuenta que la gramática completa del cálculo lambda es la siguiente:

1 <Exp> ::= (λVar.<Exp>) (definición de abstracción)

2 <Exp> ::= (<Exp> <Exp>) Aplicación (sustitución o concatenación)

3 <Exp> ::= (Var) | (Cte) Definición de Variables o constantes según se vió en la teoría
(En lugar del símbolo λ es posible utilizar la palabra *lambda*)

Además, cuando la parte izquierda de la segunda regla es una abstracción, se produce una reducción por sustitución. Es decir, cuando se tiene una expresión de la forma: ((λx.M) N), se sustituye en M todas las ocurrencias de x por N y eliminando el símbolo λ al finalizar.

Nota: *Puede usar el intérprete https://jacksongl.github.io/files/demo/lambda/ para ejecutar las expresiones lambda.*

a) Realizar el árbol de parsing bajo la forma de una lista en la que cada elemento consiste de *regla aplicada, fragmento de código sobre el cual se aplica y resultado de la aplicación*, para la siguiente expresión: “(lambda x.1 (x 3)) ((lambda y.(lambda z.y z)) 1 2)”
Por ejemplo, el primer elemento de la lista podría ser:
regla número: 2,
fragmento: (lambda x.1 (x 3)) ((lambda y.(lambda z.y z)) 1 2)
resultado: (1 (((lambda y.(lambda z.y z)) 1 2) 3))
y así sucesivamente.

b) En las siguientes líneas de código, identificar (utilizando la terminología de los lenguajes imperativos) si se trata de una:

- definición de función
- invocación de función con parámetro real literal

- invocación a función con el resultado de haber llamado a otra función
- pasaje de una función como parámetro
- retorno de una función como resultado

i) (lambda x.x 2) 1
ii) (lambda x.1 (x 2)) (lambda y.y)
iii) (lambda x.x)
iv) (lambda x.(lambda y.x y)) 1 2
v) (lambda x.x 2) ((lambda y.y) 1)

c) Indicar a cuáles de las líneas de código enunciadas en el inciso b) corresponden las siguientes líneas de código en lenguaje Python e identifique los siguientes elementos: parámetro formal, parámetro real, cuerpo de la función aplicada.

```
p1: print ((lambda x: x+2) (1))
p2: print ((lambda x: x(2)+1) (lambda y: y))
p3: print ((lambda x: (lambda y : x + y) (1) ) (2))
```

Ejercicios adicionales del TP para la materia “Lenguajes de Programación I” (plan 2011)

Ejercicio 2011_1

Considerando el fragmento de la gramática del lenguaje Go:

1. Assignment = ExpressionList assign_op ExpressionList .
2. assign_op = [add_op | mul_op] "=" .
3. ExpressionList = Expression { "," Expression } .
4. Expression = UnaryExpr | Expression binary_op Expression .
5. UnaryExpr = PrimaryExpr | unary_op UnaryExpr .
6. binary_op = "||" | "&&" | rel_op | add_op | mul_op .
7. rel_op = "==" | "!=" | "<" | "<=" | ">" | ">=" .
8. add_op = "+" | "-" | "|" | "^" .
9. mul_op = "*" | "/" | "%" | "<<" | ">>" | "&" | "&^" .
10. unary_op = "+" | "-" | "!" | "^" | "*" | "&" | "<-" .
11. PrimaryExpr = Operand.
12. Operand = Literal | OperandName | "(" Expression ")" .
13. OperandName = Identifier

donde Literal e Identifier hacen referencia a los no terminales de constante e identificador de la teoría.

Para cada una de las siguientes instrucciones, si es posible, construir el árbol de parsing. Si no es posible, indicar cuáles de las reglas se podrían aplicar.

El árbol de parsing puede construirse gráficamente o como una lista de duplas consistentes en:

[número de regla, parte del código sobre la cual se aplica la regla]

Por ejemplo para el inciso a): [13, “x”], [12,“3”], [2,“=”],

a) x = 3
b) x1, x2 = 3, 4
c) x +=3
d) 1 = 2
e) x = 1 + (y = 2)

Ejercicio 2011_2

Considerando la gramática completa del lenguaje Go (https://golang.org/ref/spec) que además es posible descargarla de aquí.
Construir el árbol de parsing como una lista de duplas consistentes en:

[nombre de regla, elementos reconocidos/producidos por la regla]

para las instrucciones del siguiente programa:

1. var a [10] int
2. var b [10][20] int
3. var i,j = 3,4
4. a[3] = 5 + i
5. b[i][j] = 7
6. b[3][a[i]] = 7

Ejercicio 2011_3

Observando las reglas BNF del lenguaje Java desde Java BNF. Para lo solicitado en cada inciso, si es posible o necesario *solucionarlo sintácticamente* proponga eliminaciones, modificaciones o agregados de reglas.
Si la solución debe ser *indefectiblemente semántica o pragmática*, indicar el motivo.

a) Se desea modificar el lenguaje para que soporte variables de tipo void. ¿Qué ocurriría si los arreglos no pudiesen ser void?
b) Se desea incorporar el mecanismo de inferencia de tipos por el cual se permite omitir el tipo de resultado retornado por un método.
c) Se desea que el lenguaje permita la inferencia de tipos por la cual se admite omitir el tipo de la variable declarada siempre y cuando se realice junto con una inicialización.
d) Se debe evitar la combinación de los modificadores public y private en la misma declaración de atributos (fields) definidos en la clase para evitar que se produzca una inconsistencia semántica.