ESCUELA SUPERIOR POLITÉCNICA DEL LITORAL

FACULTAD DE INGENIERÍA EN ELECTRICIDAD Y COMPUTACIÓN
LENGUAJES DE PROGRAMACIÓN
TERCERA EVALUACIÓN - I TÉRMINO 2015

Nombre: _____ ***Matrícula:*** _____

Sección A

1. ¿Cuál de las siguientes sentencias es verdadera para l-values y r-values? **[7%]**
   a. Un l-value es un valor lógico y un r-value es un valor real.
   b. Los l-values están siempre a la izquierda de los r-values.
   c. Un l-value se refiere a una ubicación de una variable mientras que un r-value a su valor actual.
   d. Los l-values son locales y los r-values son relativos.
   e. Los l-values son estáticos y finales mientras que los r-values son dinámicos y extensibles.
2. ¿Qué distingue un lenguaje puramente funcional de uno imperativo? **[7%]**
   a. En un lenguaje puramente funcional no hay variables y por lo tanto no hay operaciones de asignación.
   b. Un lenguaje puramente funcional carece de sentencias "go to", pero un lenguaje imperativo siempre tiene un comando.
   c. Todos los subprogramas deben ser declarados con la función keyword en un lenguaje puramente funcional.
   d. No hay una diferencia real, solo una diferencia en el estilo de codificación recomendado.
3. ¿Qué sucede si una asignación tal como **x:=y**? **[7%]**
   a. La dirección de **x** es modificada a la dirección de **y**
   b. La dirección de **y** es modificada a la dirección de **x**
   c. El objeto enlazado a **y** es copiado y enlazado a **x** y cualquier enlace previo de **x** a un objeto es perdido.
   d. **x** y **y** se convierten en *aliases*.
   e. Los contenidos de **y** son copiados en el almacenamiento asignado a **x**

Sección B

4. Usted ha sido contratado por una nueva compañía bioinformática que hace todo su desarrollo en Lisp. La compañía ha escogido Lisp porque una lista en Lisp es una manera natural de representar un ADN y secuencias de aminoácidos. Una forma de determinar una larga secuencia de ADN es romperla en piezas pequeñas, buscar la secuencia de piezas y luego re-ensamblar las piezas basadas en áreas en donde se sobreponen con los mismos códigos. El problema a continuación le pide a usted escribir unas funciones sencillas para hacerlo: **[18%]**
   - Complete la definición para la función PREFIX. Toma dos argumentos que se asumen que son listas de átomos y retorna verdadero si la primera es un prefijo de la segunda. Por ejemplo:

```
(prefix '(a b) '(a b c d)) ==>  T
(prefix NIL '(x y z)) => T
(prefix '(a b) '(a a b c d)) => NIL
```

```
(defun prefix (one two)  (cond ...))
```

   - Usando prefix como una subrutina, escriba una función recursiva overlap que tome dos secuencias y retorne la longitud de la región superpuesta común (es decir, una cola de la primera secuencia que es igual a la parte inicial de la segunda). Por ejemplo:

```
(overlap '(c a t g) '(t g c a c)) => 2
(overlap '(t c t a c t) '(a c g t)) => 0
(overlap '(c a c t g) '(c a c t g)) => 5
```

```
(defun overlap (one two) (cond ... ))
```


MSc. Carlos Mera Gómez

- Usando prefix como una subrutina, escriba una función ADNSLICE que tome dos listas representando secuencias de ADN y las corte, removiendo cualquier secuencia de superposición común. Esto es, que si alguna cola de la primera lista es igual a algún prefijo de la segunda lista, remueve esa cola de la concatenación. Por ejemplo:

```
(dnasplice '(a t a) '(c t g)) => '(a t a c t g)
(dnasplice '(c a t a) '(t a c g)) => (c a t a c g)
(dnasplice '(a a a g t g c) '(t g c g t g)) => (a a a g t g c g t g)
(dnasplice NIL '(a t)) => (a t)
```

5. Asuma que en el Lisp listener se ha escrito lo siguiente: **[22%]**

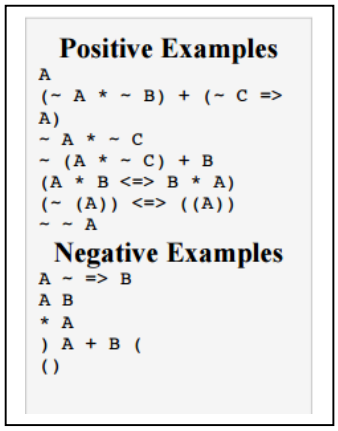
```
> (setf L1 `(a b)  L2 `(c d)  L3 `(e f))
(e f)
> (defun foo (l)  (if (atom l) nil (append (foo (cdr l) (cons (car l) nil)))))
foo
> (defun bar (one two)
    (setf (car one) (car two))
    (setf (cdr one) (cdr two))
    one)
bar
```

Muestre cuál sería la salida para los valores de cada una de las **11** expresiones:

```
> (cons l1 l2)

> (append l1 l2)

> (foo L1)

> (cons (car l1)
        (cons (car (cdr l2))
              (cdr (cdr l3))))

> (list (car l1) (car l2))

> (cons (car l1) (car l2))

> (setf L4 (list L1 (cdr L2) (foo L3)))

> (list l1 l2 l3)

> (bar l1 l2)

> (list l1 l2)

> (setf (cdr l3) l3)
```

6. Asumiendo que las expresiones Booleanas son construidas con los siguientes símbolos:
   - Operadores infix binarios: *, +, =>, <=>
   - Operadores prefix unarios: ~
   - Variables: A, B, C
   - Paréntesis: (,)

   El operador ~ tiene la precedencia más alta, seguido de * y +, los cuales tienen igual precedencia. Los operadores => y <=> tienen la más baja precedencia. Todos los operadores tienen asociación izquierda. Los paréntesis son usados, usualmente, para agrupar expresiones. Ejemplos positivos y negativos se muestran a continuación:



a. Escriba una gramática BNF para este lenguaje: Asegúrese de generar todas las expresiones Booleanas dadas. *Pista: Usted probablemente desee definir cuatro símbolos no terminales: <exp>, <term>, <factor> y <var>* **[15%]**

b. Usando su gramática elabore un árbol de análisis sintáctico para la expresión A*B => A **[15%]**

7. La tabla que se adjunta proporciona los símbolos que Python utiliza en expresiones regulares.
   a. Un nombre de variable en Python debe empezar con una letra o un guion bajo y puede ser seguido de cualquier número de letras, dígitos y guiones bajos. Proporcione una expresión regular que coincida un nombre de variable de Python legal. **[9%]**

Python RE symbols

^	the beginning of the line
$	the end of the line
+	one or more times
?	at most one time
*	zero or more time
(...)	a group
(?:...)	a noncapturing group
\t	a tab
\n	a newline character
{n}	n times
{n, m}	a range at least n and at most m
[...]	a character class
.	any character
\s	whitespace
\d	a number
\b	a word boundary