CC41A Lenguajes de Programación

Examen - Otoño 2005 Profesor: Luis Mateu

Pregunta 1

El problema de hacer calzar un patrón *p* con una lista de números *l* (match) consiste en calcular la lista de substituciones que se necesita hacer en *p* para que sea igual a *l*. El patrón *p* es una lista que contiene números o símbolos. Los símbolos denotan las variables del patrón y los números corresponden a información que debe aparecer literalmente en la lista *l*, la cual contiene solo números. Por ejemplo:

```
(match '(a) '(1)) => ((a 1))                          una sola substitución
(match '(1) '(1)) => ()                               no hay substituciones
(match '(1) '(2)) => #f                               no calzan
(match '(1 b c) '(1 2 3))   => ((b 2) (c 3))          dos substituciones
(match '(a 1 2) '(0 1 2 3)) => #f                     no calzan debido al largo
(match '(a 1 2) '(0 1)) => #f                         no calzan debido al largo
(match '(a 4 2 b 7 c) '(1 4 2 1 7 7)) => ((a 1) (b 1) (c 7))
(match '() '())   => ()                               no hay substituciones
```

a.- Escriba un programa en Scheme *estrictamente funcional* que calcule la función `(match p l)`. Preocúpese especialmente de que funcione para los ejemplos dados más arriba, notando que todos los símbolos que contiene el patrón son *distintos*. Recuerde que en Scheme Ud. puede determinar si un valor **`v`** es un símbolo utilizando la función **`(symbol? v)`** y si es un número con **`(number? v)`**.

b.- Modifique la función de la parte a.- de modo que funcione para los ejemplos anteriores, y además para los siguientes ejemplos:

```
(match '(a 2 a b) '(1 2 1 3)) => ((a 1) (b 3))
(match '(a a) '(1 2)) => #f
(match '(a a a) '(1 1 2)) => #f
```

Es decir que en esta parte el patrón puede contener símbolos repetidos. En tal caso considere un símbolo que aparece por primera vez en el patrón y que se hace coincidir en la lista con un número N. Si el mismo símbolo aparece nuevamente en el patrón, debe ser hecho coincidir en la lista con el mismo valor N.

Utilice el procedimiento predefinido assoc, el que funciona de la siguiente manera:

```
(assoc 'b '((a 1) (b 2) (c 3))) => (b 2)
(assoc 'x '((a 1) (b 2) (c 3))) => #f
```

(Ud. no puede agregar complejidad innecesaria a su solución de la parte a.-. En particular no puede dar una solución que considere símbolos repedidos. Esto significa que si sólo resuelve la parte b.- aludiendo que también satisface los requerimientos de la parte a.- tendrá solo la mitad del puntaje.)

Pregunta 2

a.- Dada la siguiente base de reglas en Prolog:

```
term([X], X) :- number(X).
term(E, Z) :- append(L, [/, Y], E), number(Y), term(L, X),
              Z is X/Y.

exp(E, Z) :- term(E, Z).
exp(E, Z) :- append(L, [- | R], E), term(L, X), exp(R, Y),
              Z is X-Y.
```

Encuentre la o las soluciones para X en las siguientes consultas:

```
?- exp([4, -, 3, -, 1], X).
?- exp([4, /, 2, /, 2], X).
```

Observaciones: El predicado number(X) tiene éxito cuando X es un número. Tenga cuidado porque el resultado podría no ser el esperado. Explique.

b.- Escriba una base de reglas en Prolog que permita evaluar una lista de tokens que representa una expresión aritmética. Ejemplos:

consulta	*soluciones*
`?- exp([1], X).`	`X= 1`
`?- exp([8,/,4,/,2],X).`	`X= 1`
`?- exp([4,+,3,*,2],X).`	`X= 10`
`?- exp([[4,+,3],*,2],X).`	`X= 14`
`?- exp([3,*,[1,+,2]],X).`	`X= 9`

Observaciones: Utilice el predicado is_list(L) para determinar si L es una lista. Recuerde que en el análisis de expresiones por descenso recursivo visto en clases se utilizaba una función de análisis adicional para los *factores* además de las funciones para *expresiones* y *términos*.

c.- Considere el caso en que no hay reglas de precedencia o asociación para las expresiones. Escriba una base de reglas en Prolog que dada una lista de tokens, que representa una expresión aritmética, permita consultar cuáles son todos los árboles de sintaxis abstracta que se pueden construir para esa expresión. Ejemplos:

consulta	*soluciones*
`?- parse([1,-,2,/,3],X).`	`X=sub(1,div(2,3)); X=div(sub(1,2),3)`
`?- parse([4,/,2,/,2],X).`	`X=div(4,div(2,2)); X=div(div(4,2),2)`

`?- parse([1,-,1,-,1,`
`-,1],X).`
	`X=sub(1,sub(1,sub(1,1)));`
	`X=sub(1,sub(sub(1,1),1));`
	`X=sub(sub(1,1),sub(1,1));`
	`X=sub(sub(1,sub(1,1)),1);`
	`X=sub(sub(sub(1,1),1),1);`

Observaciones: Sólo considere los operadores - y /.