Instituto Tecnológico de Costa Rica
Escuela de Ingeniería en Computación
Lenguajes de Programación
Proyecto 3

Functional Programming Reloaded!

Ustedes van a definir varias funciones **SML**. Algunas de ellas van a ser bastante cortas porque usarán funciones *high-order*. También en algunos casos se utilizará la biblioteca de **SML**.

1. Escriba una función *only_capitals* que toma una lista de *string* y retorna una lista de *string* que contiene solamente *strings* en el argumento que inician con una letra mayúscula. Se asume que todos los *string* contienen al menos 1 caracter. Utilice *List.filter*, *Char.isUpper* y *String.sub* de la biblioteca de **SML**.

2. Escriba una función *longest_string1* que toma una lista de *string* y retorna el *string* más grande en la lista. Si la lista está vacía retorna "". En el caso de empates, retorna el *string* más cercano al inicio de la lista. Use *foldl*, *String.size* y sin recursividad (obviamente la recursividad está dada solamente en *foldl*).

3. Escriba una función *longest_string2* que es exactamente como *longest_string1* excepto que en el caso de empates retorna el *string* más cercano al final de la lista. Use *foldl* y *String.size*.

4. Escriba las funciones *longest_string_helper*, *longest_string3* y *longest_string4*, tal que:
   - *longest_string3* tiene el mismo comportamiento que *longest_string1* y *longest_string4* tiene el mismo comportamiento que *longes_string2*.
   - *longest_string_helper* es de tipo **(int * int -> bool) -> string list -> string** (cabe resaltar el uso de *currying*). Esta función se ve como *longest_string1* y *longest_string2* pero es más general porque toma una función como argumento.
   - Si *longest_string_helper* se le es pasada a una función que se comporta como **>** (entonces retorna **true** cuando el primer argumento es estrictamente mayor que el segundo), entonces la función retornada tiene el mismo comportamiento que *longest_string1*.
   - *longest_ string3* y *longest_string4* son definidas con *val-bindings* y *partial applications* de *longest_string_helper*.

5. Escriba una función *longest_capitalized* que toma una lista de *string* y retorna el *string* más grande de la lista que inicie con mayúscula, o "" si no hay *strings* que cumplan. Asuma que todos los *strings* tiene al menos 1 caracter. Use *val-bindings* y el operador **o** de la biblioteca de **SML** para composición de funciones. Resuelva los problemas de empates como se hizo en el ejercicio 2.

6. Escriba una función *rev_string* que toma un *string* y retorna ese mismo *string* pero en orden inverso. Use el operador **o**, la función de la biblioteca de **SML** *rev* para invertir listas y funciones del módulo *String* de la biblioteca de **SML.**

Estas funciones serán utilizadas en los ejercicios finales de este proyecto.

7. Escriba una función *first_answer* de tipo **('a -> 'b option) -> 'a list -> 'b** (los 2 argumentos son *currying*). El primer argumento debe ser aplicado a elementos del segundo argumento en orden hasta la primera vez que retorne *SOME* ***v*** para algún ***v***, y entonces ***v*** es el resultado de llamar a *first_answer*. Si el primer argumento retorna *NONE* para todos los elementos de la lista, entonces *first_answer* debe lanzar una excepción *NoAnswer*.

8. Escriba la función *all_answers* de tipo **('a -> 'b list option) -> 'a list -> 'b list option** (note que los 2 argumentos son *currying*). El primer argumento debe ser aplicado a los elementos del segundo argumento. Si retorna *NONE* por cualquier elemento, entonces el resultado de *all_answers* es *NONE*. Si no, la llamada del primer argumento va a producir *SOME lst1*, *SOME lst2*, ..., *SOME lstn* y el resultado de *all_answers* es *SOME lst* donde *lst* es *lst1*, *lst2*, ..., *lstn* anexados (*appended*), en este caso el orden no importa. Use el operador **@** y una nota final: si se llama a la función *all_answers f []*, debe retornar *SOME []*.

Los siguientes problemas utilizan las siguientes definiciones:

```
datatype pattern = Wildcard
         | Variable of string
         | UnitP
         | ConstP of int
         | TupleP of pattern list
         | ConstructorP of string * pattern

datatype valu = Const of int
          | Unit
          | Tuple of valu list
          | Constructor of string * valu

fun g f1 f2 p =
    let
        val r = g f1 f2
    in
        case p of
            Wildcard          => f1 ()
          | Variable x        => f2 x
          | TupleP ps         => List.foldl (fn (p,i) => (r p) + i) 0 ps
          | ConstructorP(_,p) => r p
          | _                 => 0
    end
```

Dado un ***valu v*** y un ***pattern p***, ***p*** hace un *match* con ***v*** o no. Si se hace un *match* se produce una lista de parejas ***string*** * ***valu***; en este caso el orden no interesa. Las reglas para el *matching* son las siguientes:

- **Wildcard**: hace un *matching* con cualquier cosa y produce una lista de *bindings empty*.
- **Variable s**: hace un *matching* con cualquier valor ***v*** y produce una lista de un elemento que contiene ***(s, v)***.
- **UnitP**: hace *matching* solamente con **Unit** y produce una lista vacía de *bindings*.
- **ConstP 51**: hace *matching* solamente con **Const 51** y produce una lista vacía de *bindings* (y de manera similar para cualquier otro entero).
- **TupleP ps**: hace *matching* con un valor de la forma **Tuple vs** si **ps** y **vs** tienen la misma longitud (*length*). Y por cada i, el i-ésimo elemento de **ps** hace *matching* con el i-ésimo elemento de **vs**. La lista de *bindings* a retornar, es el resultado de todas las listas de los *pattern matching* anidados anexados (*appended*).
- **ConstructorP(s1, p)**: hace *matching* con **Constructor(s2, v)** si **s1** y **s2** son el mismo *string* y **p** hace matching con **v**. La lista de *bindings* a retornar es la producida por el *pattern matching* anidado. En el caso de **s1** y **s2** les llamaremos los nombres de los constructores (*constructors names*).
- No hay matching.

Para el ejercicio 9 necesitará de la definición de *pattern* y la función **g**.

9. (**a**) Use **g** para definir una función *count_wildcards* que toma un *pattern* y retorna cuantos *Wildcards* ese *pattern* contiene.

   (**b**) Use **g** para definir una función *count_wild_and_variable_lengths* que toma un *pattern* y retorna el número de *Wildcards* que están en *pattern* mas la suma de las longitutes (*lengths*) de todas las variables que el *pattern* contenga. Utilice **String.size**, lo que interesan son los nombres de las variables, el nombre de los constructores es irrelevante.

   (**c**) Use **g** para definir una función *count_some_var* que toma un *string* y un *pattern* (como un *pair*) y retorna el número de veces que el *string* aparece como una variable en el *pattern*. Nos interesan solamente los nombres de las variables, el nombre de los constructores es irrelevante.

Para el ejercicio 10 necesitará solamente de la definición de *pattern*.

10. Escriba una función *check_pat* que toma un *pattern* y retorna **true** si todas las variables que aparecen en el *pattern* son distintas unas de otras (tiene diferentes *strings*). Los nombres de los constructores son irrelevantes. Programe esta función con dos *helper functions*. La primera toma un *pattern* y retorna una lista de *strings* contenidos en las variables. En este caso use **foldl** y el operador **@**. La segunda función toma la lista de *strings* y verifica que no haya repetidos. En este caso use **List.exists**.

Para el ejercicio 11 necesitará de las definiciones de *pattern*, *valu* y las reglas para *matching* definidas anteriormente.

11. Escriba una función *match* que toma un **valu * pattern** y retorna un **(string * valu) list option**, que será **NONE** si el *pattern* no hace *matching* y **SOME lst** donde **lst** es la lista de *bindings* si se hizo *matching*. En este caso el *pattern matching* debe tener 7 casos (*branches*). El *branch* para tuplas usará **all_answers** del ejercicio 8 y **ListPair.zip** de la biblioteca de **SML**.

El ejercicio 12 necesitará las definiciones de *pattern* y *valu*.

12. Escriba una función *first_match* que toma un valor y una lista de *patterns* y retorna **(string * valu) list option**, que sería **NONE** si ningún *pattern* hace *matching* o **SOME lst** donde **lst** es la lista de *bindings* para el primer *pattern* en la lista que hace *matching*. Use la función *first_answer* del ejercicio 7.

Resumen de los tipos de las funciones:

```
val g = fn : (unit -> int) -> (string -> int) -> pattern -> int
val only_capitals = fn : string list -> string list
val longest_string1 = fn : string list -> string
val longest_string2 = fn : string list -> string
val longest_string_helper = fn : (int * int -> bool) -> string list -> string
val longest_string3 = fn : string list -> string
val longest_string4 = fn : string list -> string
val longest_capitalized = fn : string list -> string
val rev_string = fn : string -> string
val first_answer = fn : (’a -> ’b option) -> ’a list -> ’b
val all_answers = fn : (’a -> ’b list option) -> ’a list -> ’b list option
val count_wildcards = fn : pattern -> int
val count_wild_and_variable_lengths = fn : pattern -> int
val count_some_var = fn : string * pattern -> int
val check_pat = fn : pattern -> bool
val match = fn : valu * pattern -> (string * valu) list option
val first_match = fn : valu -> pattern list -> (string * valu) list option
```

Test cases

```
val test1 = only_capitals ["A","B","C", "lower"] = ["A","B","C"]

val test2 = longest_string1 ["A","bc","C"] = "bc"

val test3 = longest_string2 ["A","bc","C"] = "bc"

val test4a = longest_string3 ["A","bc","C"] = "bc"

val test4b = longest_string4 ["A","B","C"] = "C"

val test5 = longest_capitalized ["A","bc","C"] = "A"
```

```
val test6 = rev_string "abc" = "cba"

val test7 = first_answer (fn x => if x > 3 then SOME x else NONE) [1,2,3,4,5] = 4

val test8 = all_answers (fn x => if x <> 2 then SOME [x] else NONE) [2,3,4] = NONE

val test9a = count_wildcards Wildcard = 1


val pattern1 = TupleP([Variable "var", Wildcard, TupleP([Variable "var", Wildcard,
TupleP([Variable "var", Wildcard])])]);
val test9a_2 = count_wildcards pattern1 = 3;

val test9b = count_wild_and_variable_lengths (Variable("a")) = 1

val pattern2 = TupleP([Wildcard, Variable("abc")])
val test9b_2 = count_wild_and_variable_lengths pattern2 = 4

val pattern3 = TupleP([Variable("x"), Wildcard, Variable("x")])
val test9c = count_some_var ("x", pattern3) = 2

val test10 = check_pat (Variable("x")) = true

val pattern4 = TupleP([ConstP 12, Variable "var1", Variable "var2",
ConstructorP("constr1", Wildcard)]);
val test10_2 = check_pat pattern4;

val test11 = match (Const(1), UnitP) = NONE

val test11_2 = match(Unit, UnitP) = SOME [];

val pattern_t = Tuple([Const 12, Constructor("blah", Unit), Constructor("constr1",
Tuple([]))]);
val pattern_tp = TupleP([ConstP 12, Variable "var1", ConstructorP("constr1", Wildcard)]);
val test11_3 = match(pattern_t, pattern_tp) = SOME [("var1", Constructor("blah", Unit))];

val test12 = first_match Unit [UnitP] = SOME []
```