MC-8812 Teoría de los lenguajes de programación
Proyecto 2: **Extensión del intérprete de un lenguaje funcional**

Objetivo
Al concluir este proyecto, ustedes habrán extendido el intérprete (dado) de un lenguaje funcional con pares y patrones. La extensión consiste de varias formas de declaración, nuevas formas de expresión (registros, iteraciones y condicionales generalizados), así como patrones-registro y patrones estratificados.

Base
Se les ha facilitado el intérprete de un lenguaje de programación funcional que posee muchas de las características básicas de lenguajes como Standard ML. Ese intérprete les sirve como referencia para resolver este proyecto. Deberá usarse alguna de las implementaciones del lenguaje Standard ML[1], en que deberá escribirse el intérprete del lenguaje funcional extendido. Les serán también útiles los apuntes y discusiones hechos en clases.

Entradas
Los programas de entrada serán suministrados por ustedes mismos, mediante la representación de sintaxis abstracta facilitada por el profesor[2]. El programa por evaluar deberá pasarse a la función `evalProg`. Esa función evalúa una expresión en el ambiente de las funciones pre-definidas.

Lenguaje fuente

Sintaxis y semántica
El lenguaje fuente tendrá una sintaxis abstracta que extiende la dada para el intérprete del lenguaje funcional con pares y patrones.

Expresiones
La expresión `LetExp` deberá procesarse de manera distinta a la que se muestra en el intérprete original, para tomar en cuenta la manera más rica en que pueden hacerse declaraciones en el lenguaje extendido. El intérprete del lenguaje *imperativo* estudiado en clases da una idea general respecto de cómo resolver este problema.

La expresión `RegExp` debe evaluarse de manera que para cada para par (identificador_de_campo, expresión), la expresión se evalúe en el ambiente vigente y su valor quede asociado al identificador_de_campo. Los pares resultantes (identificador_de_campo, valor) conformarán el valor-registro correspondiente a la expresión-registro.

La expresión `CampoExp` accede el campo correspondiente al identificador, dentro del valor-registro que resultó de evaluar la expresión suministrada (que puede ser *cualquier* expresión, siempre que dé como resultado un valor-registro). Si se intenta acceder un campo de un valor que no es un registro, deberá levantarse la excepción `ErrorDeTipo` con un mensaje apropiado.

La expresión `CondExp` procede de manera semejante al **cond** de Scheme: se van probando los pares (condición, expresión) en el orden dado; si la condición es verdadera, se evalúa la expresión adyacente; si la condición es falsa, se continúa con el siguiente par. Si ninguna de las condiciones es verdadera, se procede a evaluar la expresión final (**else**); cuando ésta está ausente, se levanta la excepción `NoHayClausulaElse`.

La expresión `IterExp` procede de manera semejante al **do** de Scheme[3]: consta de la declaración *colateral* de un conjunto de variables *locales a la expresión iterativa*, cada una de las cuales toma un valor inicial y es afectada – en cada iteración subsiguiente – por el valor que se obtiene al evaluar su correspondiente expresión de actualización. Inmediatamente después de dar valor inicial a las variables locales de la repetición, y antes de hacer las actualizaciones que llevan hacia la siguiente iteración, se evalúa la condición. Si la condición es verdadera, se procede a evaluar la expresión de finalización. Si la condición es falsa, se actualizan *en paralelo* todas las variables de la expresión iterativa y se vuelve a evaluar la condición. Decimos que se procede 'en paralelo' porque las expresiones de actualización solamente pueden depender de los valores de las variables no-locales (obtenidas del

[1] Por su simplicidad, se recomienda Moscow ML (mosml).
[2] Ver el apéndice.
[3] En su variante sin efectos colaterales ni asignaciones.

ambiente vigente) o de los valores *previos* de las variables locales (declaradas por la expresión iterativa). Ver ejemplo en el apéndice.

Declaraciones[4]
Se les ha facilitado un **datatype** que especifica la sintaxis abstracta del lenguaje funcional extendido, que incluye a las declaraciones:

- De valores. Estos son determinados por un patrón y una expresión. La expresión evalúa a un valor que es filtrado por un patrón[5]. El resultado es un ambiente en el cual se asocian las variables incluidas en el patrón con los componentes correspondientes en el valor (si no hay errores de concordancia).
- La declaración de valores puede incluir un modificador que indica si los identificadores deben ser tratados de manera recursiva (esto es útil para las funciones). Si se quiere declarar funciones mutuamente recursivas, éstas deberán declararse en una declaración de valor que incluya sus identificadores en un patrón compuesto (par o registro), con sus correspondientes expresión-función dentro de una estructura de datos isomorfa (expresión-par o expresión-registro)[6].
- Las declaraciones compuestas son: *colaterales* (`AndDecl`), *locales* (`LocalDecl`) y *secuenciales* (`SecDecl`). El comportamiento corresponderá al explicado en clase para el lenguaje imperativo. Note que el intérprete del lenguaje imperativo, facilitado por el profesor, no verifica que los identificadores sean únicos; esto deberá ser corregido en su intérprete para el procesamiento de las declaraciones colaterales.

Patrones
Además de los patrones que aparecen en el lenguaje original, es posible tener patrones *estratificados* y patrones-*registro*.

En un patrón estratificado, un identificador puede asociarse a *todo* un valor compuesto (patrón **as** en Standard ML, `ComoPat` en nuestra sintaxis abstracta). Al hacer la concordancia, el identificador quedará asociado a *todo* el valor compuesto, que será filtrado de manera estructural por un patrón subordinado. Si la concordancia del patrón subordinado falla, también fallará la concordancia del patrón estratificado.

Los patrones-registro consisten de una lista de los identificadores que deberán hacerse concordar con un valor-registro. Un patrón-registro concordará con cualquier valor-registro que contenga *al menos* los mismos identificadores de campo contenidos en el patrón-registro. El resultado de tal concordancia será un *ambiente* en el cual aparezcan ligados los identificadores de campo a los valores correspondientes.

Nótese que en `concord.sml` del intérprete base se usa **<+>** donde debió usarse **<|>** para combinar los ambientes resultantes de concordar los patrones izquierdo y derecho de un patrón-par contra un valor-par. Ud. debe usar **<|>**, para que se lea (en `concord.sml`):

```
|   concordar (ParPat (pati,patd)) (Par (vali,vald))
    = (concordar pati vali)
      <|>                          (* combina ambientes disyuntos *)
      (concordar patd vald)
```

Alcance
El lenguaje tiene *alcance léxico*. Los identificadores usados por la invocación de una función se resuelven con referencia al ambiente de *definición* de ésta[7], extendido por el ambiente que resulta de concordar el valor del argumento con el patrón de alguna regla de la función.

[4] Ver el intérprete imperativo y los apuntes de clase referentes al manejo de declaraciones compuestas.
[5] Note que tanto el patrón como el valor pueden ser compuestos (pares o registros).
[6] Esto fue ilustrado en clase, al definir un par de funciones `val rec (f,g) = (fn x => … g … , fn y => … f …)`.
[7] Atención: esto incluye los identificadores no-locales vigentes cuando se definió la función, así como cualesquiera identificadores que se introdujeron al declarar *recursivamente* la función de marras.

Paso de parámetros
En este lenguaje los parámetros se pasan *por constante*, no hay ningún otro modo de paso de parámetros. La semántica del paso de parámetros es por concordancia de patrones y está explicada en el intérprete original. Las extensiones a la concordancia de patrones explicadas arriba afectan, consecuentemente, el paso de parámetros.

Programas
En nuestro lenguaje funcional (extendido), un *programa* corresponde a una *expresión*[8]. Esta expresión será evaluada en el contexto de un ambiente de operaciones (funciones) primitivas.

Validaciones
El lenguaje no es estricto en cuanto a los tipos; a ustedes no se les pide que implementen restricciones para comprobar la validez de los tipos en las expresiones *antes* de ejecutar el programa. Debe evaluarse que las operaciones primitivas se apliquen sobre valores de los tipos apropiados (como se ilustra en el intérprete original).

Tanto las declaraciones colaterales (`AndDecl`) como la concordancia de patrones compuestos (`ParPat`, `RegPat`, `ComoPat`) deben levantar una excepción cuando se esté declarando un identificador repetido. Asimismo, en `IterExp` debe validarse que no haya declaración duplicada de variables dentro de una misma expresión iterativa[9]. *Recomendación*: esto es fácil y elegante si construyen una función (llamémosla **<|>**), semejante a **<+>**, que combine ambientes si estos tienen dominios *disyuntos* o que levante una excepción (`DominiosNoDisyuntos`) en caso de haber identificadores repetidos en los dominios.

Nuevas primitivas
No se pide la definición de nuevas primitivas.

Proceso y salidas
El intérprete debe ser capaz de procesar todo el lenguaje. Ud. debe usar la sintaxis abstracta dada en el apéndice. Procure aprovechar al máximo lo que está ya implementado en el intérprete de base.

Ud. deberá construir casos de prueba que ejerciten *todas* las extensiones hechas al lenguaje funcional original. Debe existir al menos un caso de prueba por cada validación solicitada. Cada caso de prueba deberá ir documentado aparte, indicando el objetivo de la prueba, los resultados esperados y los resultados observados. En particular, deberá presentar ejemplos de programas que hagan uso de la combinación de construcciones sintácticas introducidas para efectos de esta tarea.

La salida será la que dé el ambiente de Standard ML. Ud. la guardará en archivos de texto, que incluirá como un apéndice de la documentación por entregar.

Documentación[10]
Debe documentar clara y *concisamente* los siguientes puntos:
- La representación utilizada para los registros y cualquier otro valor semántico.
- La solución dada al manejo de registros (expresiones-registro, acceso a campos de un registro).
- La solución dada a la evaluación de la expresión iterativa.
- La solución dada a la evaluación de la expresión condicional generalizada.
- La solución dada a las extensiones hechas a los patrones (patrones estratificados [‘**as’**], patrones-registro).
- La solución dada a la combinación de ambientes con dominios disyuntos (función **<|>**).
- Otras modificaciones hechas al intérprete.
- Texto fuente del intérprete completo.
- Casos de prueba y resultados observados.
- La elegancia de su programación es sujeta a evaluación (se considera parte de la documentación).

[8] En Standard ML, un programa es una declaración.
[9] Recuerde que la expresión iterativa *declara* variables *locales* a la expresión.
[10] Ponga atención a lo que se solicita como documentación. Esta es *requerida* y es evaluada *rigurosamente*.

Entrega
Domingo 2018.06.10. La documentación debe ser entregada, junto con el intérprete y las pruebas, en una carpeta comprimida en formato .zip. Deben proveer una portada en la cual se identifique, con nombres completos y carnets a los miembros del grupo. La entrega debe ser vía correo-e a itrejos@itcr.ac.cr, con el asunto "MC-8812: Proyecto 2 Intérprete funcional" seguido por los números de carnet de los miembros del grupo, separados por guiones.

Tamaño del grupo
Dos personas; excepcionalmente tres (con extensión de la sintaxis y semántica del lenguaje).

Evaluación
La documentación es muy importante. El código fuente debe tener comentarios que identifiquen claramente las extensiones o modificaciones hechas por el grupo al intérprete original.

Apéndice: sintaxis abstracta[11]

```
(* Definimos un tipo para las cosas que son opcionales *)

datatype 'a option = Nothing
                   | Something of 'a

(* Lenguaje funcional con pares y patrones, extendido para el proyecto *)

(* Identificadores, variables, etc.
   Los identificadores son representados mediante hileras. *)

type Identificador = string
type Variable      = Identificador

(* Las literales del lenguaje son enteras nada más *)

datatype Literal = Booleana of bool
                 | Entera   of int

(* Este es un lenguaje de expresiones, con las siguientes opciones:
   - Literal (entera o booleana)
   - Variable
   - Condicional simple (if)
   - Condicional generalizado (cond)
   - Par
   - Bloque (let)
   - Aplicación de función
   - Abstracción de función
   - Agregación de registro
   - Acceso calificado a campos de un registro
   - Iteración (sin efectos colaterales)

   Además de las declaraciones de valor, que pueden ser recursivas
   o no-recursivas, el lenguaje permitirá las siguientes formas de
   declaración compuesta:
   - colateral (and)
   - secuencial (;)
   - bloque (local ... in ... end)

*)

(* una declaración (de función) puede ser recursiva o no *)

datatype Recurrencia =
           Recursiva
         | NoRecursiva

datatype Expresion =
           ConstExp   of Literal
         | IdExp      of Identificador
         | IfExp      of Expresion * Expresion * Expresion
```

[11] **Debe usar esta sintaxis abstracta**. Este texto está en Standard ML, salvo unas notas al pie que se incluyen como aclaración.

```
          | ParExp     of Expresion * Expresion
          | LetExp     of Declaracion * Expresion
          | ApExp      of Expresion * Expresion
          | AbsExp     of Reglas
          | RegExp     of (Identificador * Expresion) list
          | CampoExp   of Expresion * Identificador
          | IterExp    of (Identificador * Expresion * Expresion) list * Expresion[12] * Expresion[13]
          | CondExp    of (Expresion * Expresion) list * Expresion option[14]

and      Declaracion =
           ValDecl    of Recurrencia[15] * Patron * Expresion
          | AndDecl    of Declaracion * Declaracion
          | SecDecl    of Declaracion * Declaracion
          | LocalDecl  of Declaracion * Declaracion

and       Patron =
            ConstPat   of Literal
          | IdPat      of Identificador
          | ParPat     of Patron * Patron
          | RegPat     of Identificador list
          | ComoPat    of Identificador * Patron
          | Comodin

  (* el tipo que sigue está subordinado a los datatypes anteriores *)

withtype Reglas =
            (Patron * Expresion) list

;

(* un programa es una expresión *)

type Programa = Expresion

(* Hay varias cosas en el intérprete que no están implementadas.
   Ud. deberá implementarlas. Los componentes no implementados
   levantan esta excepción cuando se trata de evaluarlos. *)

exception NoImplementada of string
```

Apéndice: ejemplo de expresión iterativa

Una expresión iterativa para calcular el factorial (en sintaxis de Scheme) podría ser:

```
(define fact
  (lambda (k)
    (do ((n       k ( - n 1))
         (product 1 ( * product n)))
        ((zero? n) product))))
```

o, en una sintaxis parecida a la utilizada en clase:

```
val fact = fn k =>
  iter val n       from k next n - 1
       val product from 1 next product * n
    when  n = 0
    yield product
  end
```

La declaración aparecería incrustada en una frase como la siguiente (donde se calcula el factorial de 6):

```
LetExp(ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 6)),
  LetExp(ValDecl(NoRecursiva, IdPat "fact",
```

[12] Esta es la condición que determina si se continúa haciendo repeticiones.
[13] Esta es la expresión de finalización: se evalúa cuando la condición resulta verdadera.
[14] Esta es la expresión que corresponde a la cláusula **else**, la cual es opcional.
[15] Observe que la recurrencia afecta las declaraciones de valor.

```
    AbsExp [(IdPat "k",
      IterExp([(
        "n", IdExp "k", ApExp(IdExp "-", ParExp(IdExp "n", ConstExp(Entera 1))))
      , ("product", ConstExp(Entera 1), ApExp(IdExp "*", ParExp(IdExp "product", IdExp "n")))
      ],
      ApExp(IdExp "=",ParExp(IdExp "n", ConstExp(Entera 0))),
      IdExp "product"))
    ])
  , ApExp(IdExp "fact", IdExp "a")))
```

Apéndice: extensiones al intérprete

En `ambi.sml`, añadir:

```
exception DominiosNoDisyuntos
```

Esta excepción es levantada por `<|>` cuando trata de combinar dos ambientes que tienen algún identificador en común.

```
infix <|>

fun amb1 <|> amb2 = ... UD. LA IMPLEMENTA ...
```

En `concord.sml`, recuerde que `<+>` no revisa si hay repetición de variables en sub-patrones de un patrón compuesto (par, registro o estratificado).

En `eval.sml` deben añadirse las excepciones necesarias:

```
exception NoEsUnaFuncion of string
and       NoSeAplicanReglas
and       NoHayClausulaElse
```

Recuerde modificar `eval.sml` para que procese todas las nuevas formas de expresión. Póngale cuidado a la expresión iterativa, porque es delicada.