Proyecto 2: Extensión del intérprete de un lenguaje funcional

Carlos Martín Flores González, 2015183528
Willard Zamora Cárdenas, 2017239202
Profesor: Ignacio Trejos Zelaya
19 de Junio del 2018

Índice

1. Estrategia de solución

Se empezó el desarrollo del proyecto por medio de la exploración del código proporcionado por el profesor. Se ejecutaron pruebas sobre el mismo con el fin de entender cómo funcionaba y las partes del código que se ejercitaban luego de cada invocación.

Una vez que se tuvo un mejor conocimiento del funcionamiento se estudió el código del intéprete imperativo para incluir el manejo de las declaraciones en este proyecto. Este fue el primer logro. Luego de se empezaron a estudiar los casos de las expresiones `RegExp` y `CampoExp` pero en principio no se logró con una solución por lo que se decidió continuar con `CondExp`. Al estudiar cómo funcionaba el 'cond' de Scheme se logró adaptar en relativamente poco tiempo el código que se venia desarrollando para dicha expresión.

Luego de la experiencia anterior, se podría decir que ya se estaba "entrando en calor" con el código SML y gracias a esto se fue desarrollando, probando y puliendo el código para las expresiones `RegExp` y `CampoExp`.

Aunque en principio se decidió no implementar `IterExp` porque era opcional, se empezaron a probar los casos proporcionados por el profesor pero estos no se lograron entender. En la última clase se consultó al profesor sobre esto y dio pie a que se lograra implementar `IterExp`.

El proceso de desarrollo en general se dio a partir de discusión, desarrollo, pruebas y validación entre los miembros del equipo.

2. Reflexiones sobre la experiencia

2.1. Martín Flores

Esta fue mi primera experiencia con Standar ML. A pesar que ya tenía experiencia en otros lenguajes de programción funcionales, este al ser nuevo siempre presenta retos y más cuando hay que cumplir con alguna entrega. Durante el desarrollo del proyecto, navegué por varios sitios Web y pude comprobar que este es un lenguaje que goza de mucha aceptación dentro de las universidades principalemente en cursos de teoría de lenguajes de programación o bien para introducir conceptos de programación funcional.

En principio, aunque puede resultar algo diferente, conforme uno se va adentrando pude notar que es un lenguaje en el que se pueden lograr mucha expresividad, el concepto del 'datatype' me parece muy para la definición de tipos a un "bajo costo.[en] términos de codificación.

No se pudo encontrar buenas herramientas de *tooling* para desarrollar en SML. El *plugin* de Sublime Text ayuda pero es limitado. El intérprete de *Moscow ML* también es limitado a la hora de introducir código.

2.2. Willard Zamora

A pesar de los contratiempos en el inicio del proyecto, con la comprehension del código y el lenguaje, el proyecto resulta ser gratificante, a medida que se progresa se adapta la forma de pensar y las soluciones se tornan más sencillas, basta con un empujón del profesor para caer en conciencia de lo que debe hacer algún segmento de código. Hago énfasis en el cambio de paradigma pues resulta entretenido, junto con la creación del intérprete, que conforme se agrega al mismo se ve como se incrementa, aunque sea un poco, la capacidad del lenguaje interpretado. Esta fue mi primera experiencia programando un intérprete y no estuvo mal, sin embargo, considero que se pudo sacar más provecho con entregas incrementales, para familiarizarnos con SML desde antes e invertir más tiempo pensando en la solución y no en como implementar la solución en SML, que fue algo que constantemente me ocurrió.

2.3. Plano grupal

Willard y Martín se conocieron durante el curso y nunca habían trabajado juntos en ningún proyecto, a pesar de esto durante el desarrollo se notó que contaban con intereses similares y se lograron complementar bien. Se considera que la comunicación constante fue un factor clave en lograr que el proyecto avanzara y se fuera puliendo paulatinamente. Durante varias de las video-llamadas que se llevaron a cabo para el desarrollo, los puntos de vista del uno y el otro fueron dándole forma al resultado final.

3. Documentación del proyecto

3.1. Representacion utilizada para los registros y cualquier otro valor semántico

Se hace uso de un nuevo tipo de valor Registro, el cual se agrega en Val.sml con el objetivo de implementar `RexExp` y `CampoExp`

```
Registros of (Identificador * Valor) list
```

3.2. La solución dada al manejo de registros (expresiones-registro, accesos a campos de un registro)

En `RexExp` se hace uso del nuevo tipo valor `Registros` y la función existente de `map_ambiente`, aplicando a cada par (Identificador, Expresion) la funcion que evalua la expresion y la asocia al identificador.

```
RegExp registros
=>  let fun map_exp exp' = evalExp ambiente exp'
    in  let val lista = map_ambiente map_exp registros
        in Registros lista
        end
    end
```

Con respecto a `CampoExp` simplemente se genera el valor registro a partir de la expresion y en ella se hace busca del identificador pasando como ambiente la lista tipo `Registros`.

```
CampoExp (exp', ident)
=>  let val Registros lista = evalExp ambiente exp'
    in  busca ident lista
    end
```

3.3. La solución dada a la evaluación de la expresión iterativa.

Se empieza por inicializar una lista con `ini_ambiente` (identificador con expresión de inicializar) y durante este proceso se verifica que el dominio sea disjunto, en caso de que lo sea se procede con la evaluación de condicionExp, si esta es verdadera se actualiza la lista con `act_ambiente` (identificador con expresión actualizar), si es falsa se evalua `trueExp`. Durante el proceso del ciclo se va hace un ambiente temporal que incluye el ambiente original concatenado con la lista actual, en el caso inicial es el ambiente original <+> la lista de valores inicializados y en los demás es ambiente original con la lista de valores actualizados, este ambiente temporal hace uso del ambiente original sin extenderlo innecesariamente, manteniendo siempre los últimos valores de la lista.

```
IterExp (lista, condicionExp, trueExp)
=>  let fun modificar ambiente' exp' = evalExp ambiente' exp'
    in  let val listaAmb = ini_ambiente modificar lista ambiente
        in  let fun ciclo listaAmb'
            = let
                val iterAmb = (ambiente <+> listaAmb')
              in
                case (evalExp iterAmb condicionExp) of
                (ConstBool false)
                => ciclo (act_ambiente modificar lista iterAmb)
                | (ConstBool true)
                => evalExp iterAmb trueExp
              end
              in ciclo listaAmb
              end
        end
    end
```

3.4. La solución dada a la evaluación de la expresión condicional generalizada

Para esta evaluación se crearon dos condiciones:

1. Cuando se pasa una lista vacía y una expresión final (caso base)
2. Cuando se pasa una lista de la forma `Condicion * Expresion` y una expresión final

Cuando se da el caso 1, se verifica primero si la `expresionFinal` es de alguno de los tipos opcionales definidos (`Something` o `Nothing`). En el caso que `expresionFinal` sea de tipo `Something`, quiere decir que la expresión si contiene una expresión final que puede ser evaluada. Para evaluar esta expresión se usa `evalExp`. En el caso que no se incluya una `expresionFinal` se lanza una excepción.

Cuando se pasa una lista de la forma `Condicion * Expresion`, primero se evalua la condición (`cond`) y en el caso que el resultado sea verdadero (`ConstBool true`) se procede entonces a evaluar la expresion (`expresion`). Si el resultado de la evaluación de `cond` sea falso (`ConstBool false`) entonces se evalua la expresión pero esta vez pasando como argumento el resultado de evaluar el resto de la lista (`tail`) con la expresión final. De esta forma se va a ir consumiento la lista de forma recursiva y probando cada uno de los pares `Condicion * Expresion`.

```
| CondExp ([], expresionFinal)
   => let
      in
        case expresionFinal of
        (Something expFinal) => evalExp ambiente expFinal
        | Nothing  => raise NoHayClausulaElse "CondExp: No hay Else"
      end
| CondExp ((cond, expresion)::tail, expresionFinal)
    => let val condicion = evalExp ambiente cond
       in
        case condicion of
             (ConstBool false)
              => evalExp ambiente (CondExp(tail, expresionFinal))
           | (ConstBool true)
              => evalExp ambiente expresion
        end
```

3.5. La solución dada a las extensiones hechas a los patrones (patrones estratificados ['as'], patrones-registro).

Se crearon dos casos para hacer la concordancia:

1. Cuando se pasa una lista vacía de identificadores junto con `Registros` (identificador * valor) (caso base)
2. Cuando se pasa una lista de identificadores con la forma `(id::tail)` junto con `Registros`

Cuando se da el caso 1, se retorna un `ambienteVacio`. Cuando se da el caso 2, se busca el identificador `id` en la lista de registros para que luego por medio del combinador de ambientes `<|>`, se combine el resultado de la operación anterior junto con el resultado de la invocación a `concordar` pasando por parámtro el resto de la lista de identificadores junto con los `Registros` para ir asociando el resto de identicadores.

A continuación un extracto del código de `Concord.sml`:

```
|   concordar (RegPat (id::tail)) (Registros (registros))
    = id |-> (busca id registros) <|> concordar (RegPat (tail)) (Registros (registros))
|   concordar (RegPat []) (Registros (registros))
```

3.6. La solución dada a la combinación de ambientes con dominios disyuntos (función <|>).

Esta solución fue proporcionada por el profesor.

3.7. Otras modificaciones hechas al intérprete

Las sigiuentes funciones fueron agregadas a Ambi.sml, `ini_ambiente` y `act_ambiente` realizan el mismo recorrido que `map_ambiente`, pero sobre una lista de tripletas.

1. `ini_ambiente` aplica la expresión de inicialización al identificador y va verificando que los dominios sean disjuntos conforme recorre la lista.
2. `act_ambiente` aplica la expresión de actualización al identificador.
3. `existe_en_lista` busca en una lista de tripletas `(identificador * _ * _ )` que no existan identificadores repetidos.

```
fun existe_en_lista ident []
    = false
|   existe_en_lista ident ((ident',_,_)::ambiente)
    = if ident = ident'
      then true
      else existe_en_lista ident ambiente

fun ini_ambiente f [] ambiente
    = []
|   ini_ambiente f ((ident, expIni, expAct)::cola) ambiente
    = if existe_en_lista ident cola then
        raise DominiosNoDisyuntos
      else
        (ident,(f ambiente expIni))::(ini_ambiente f cola ambiente)

fun act_ambiente f [] ambiente
    = []
|   act_ambiente f ((ident, expIni, expAct)::cola) ambiente
```

3.8. Casos de prueba y resultados observados

Las siguientes pruebas fueron tomadas del archivo `PruebasFun.sml`.

```
1  (*************************************************************)
2  (**********************REGISTROS******************************)
3
4  - val registro1 = RegExp [("a", ConstExp(Entera 1)),
5                            ("b", ConstExp(Entera 2))];
6  > val registro1 = RegExp [("a", ConstExp(Entera 1)), ("b", ConstExp(Entera 2))]
7      : Expresion
8  - evalProg registro1;
9  > val it = Registros [("a", ConstInt 1), ("b", ConstInt 2)] : Valor
10
11 (* Campos repetidos *)
```

```
- val registro2 = RegExp [("x", ConstExp(Entera 3)),
                          ("x", ConstExp(Entera 4))];
> val registro2 = RegExp [("x", ConstExp(Entera 3)), ("x", ConstExp(Entera 4))]
    : Expresion
- evalProg registro2;
> val it = Registros [("x", ConstInt 3), ("x", ConstInt 4)] : Valor

(* Campo existente *)
- val regA  = CampoExp(registro1, "a");
> val regA =
    CampoExp(RegExp [("a", ConstExp(Entera 1)), ("b", ConstExp(Entera 2))],
             "a") : Expresion
- evalProg regA;
> val it = ConstInt 1 : Valor

(* Campo inexistente *)
- val regC  = CampoExp(registro1, "c");
> val regC =
    CampoExp(RegExp [("a", ConstExp(Entera 1)), ("b", ConstExp(Entera 2))],
             "c") : Expresion
- evalProg regC;
! Uncaught exception:
! NoEstaEnElDominio  "c"


- val pruFun = LetExp( ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 1)),
                    regFun);
> val pruFun =
    LetExp(ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 1)),
           IfExp(ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 1))),
                 RegExp [("a", ConstExp(Entera 1)), ("b", ConstExp(Entera 2))],
                 RegExp [("x", ConstExp(Entera 3)),
                         ("x", ConstExp(Entera 4))])) : Expresion
- evalProg pruFun;
> val it = Registros [("a", ConstInt 1), ("b", ConstInt 2)] : Valor


(* Acceso a campos donde el registro es el resultado de una expresion *)

- val regA1 = CampoExp(pruFun, "a");
> val regA1 =
    CampoExp(LetExp(ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 1)),
                    IfExp(ApExp(IdExp "=",
                                ParExp(IdExp "a", ConstExp(Entera 1))),
                          RegExp [("a", ConstExp(Entera 1)),
                                  ("b", ConstExp(Entera 2))],
                          RegExp [("x", ConstExp(Entera 3)),
                                  ("x", ConstExp(Entera 4))])), "a") :
  Expresion
- evalProg regA1;
> val it = ConstInt 1 : Valor

- val regC1 = CampoExp(pruFun, "c");
> val regC1 =
    CampoExp(LetExp(ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 1)),
                    IfExp(ApExp(IdExp "=",
                                ParExp(IdExp "a", ConstExp(Entera 1))),
                          RegExp [("a", ConstExp(Entera 1)),
                                  ("b", ConstExp(Entera 2))],
                          RegExp [("x", ConstExp(Entera 3)),
                                  ("x", ConstExp(Entera 4))])), "c") :
  Expresion
- evalProg regC1;
! Uncaught exception:
! NoEstaEnElDominio  "c"


(*********************************************************)
```

```
(*********************ITERACION**************************)

(* Iteracion exitosa *)
- val iter1 = LetExp(ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 6)),
               LetExp(ValDecl(NoRecursiva, IdPat "fact",
                 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")));
> val iter1 =
    LetExp(ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 6)),
           LetExp(ValDecl(NoRecursiva, IdPat "fact",
                          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"))) : Expresion
- evalProg iter1;
> val it = ConstInt 720 : Valor

(* Iteracion no exitosa, variable de iteracion repetida *)
- val iter2 = LetExp(ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 6)),
               LetExp(ValDecl(NoRecursiva, IdPat "fact",
                 AbsExp [(IdPat "k",
                    IterExp([(
                       "n", IdExp "k", ApExp(IdExp "-", ParExp(IdExp "n", ConstExp(Entera 1))))
                    ,   ("n", ConstExp(Entera 1), ApExp(IdExp "*", ParExp(IdExp "n", IdExp "n")))
                    ],
                    ApExp(IdExp "=",ParExp(IdExp "n", ConstExp(Entera 0))),
                    IdExp "n"))
                 ])
               , ApExp(IdExp "fact", IdExp "a")));
> val iter2 =
    LetExp(ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 6)),
           LetExp(ValDecl(NoRecursiva, IdPat "fact",
                          AbsExp [(IdPat "k",
                                   IterExp([("n", IdExp "k",
                                             ApExp(IdExp "-",
                                                   ParExp(IdExp "n",
                                                          ConstExp(Entera 1)))),
                                            ("n", ConstExp(Entera 1),
                                             ApExp(IdExp "*",
                                                   ParExp(IdExp "n",
                                                          IdExp "n")))],
                                           ApExp(IdExp "=",
                                                 ParExp(IdExp "n",
                                                        ConstExp(Entera 0))),
                                           IdExp "n"))]),
                  ApExp(IdExp "fact", IdExp "a"))) : Expresion
- evalProg iter2;
! Uncaught exception:
! DominiosNoDisyuntos

```

```
(* Iteracion para que devuelva el n mas recientemente definido *)
- val iter3 = LetExp(ValDecl(NoRecursiva, IdPat "n", ConstExp(Entera 1)),
                  IterExp([(
                            "n", ConstExp(Entera 1), ApExp(IdExp "+", ParExp(IdExp "n", ConstExp(
                                Entera 1))))
                           ],
                           ApExp(IdExp ">",ParExp(IdExp "n", ConstExp(Entera 10))),
                           IdExp "n"
                          ));
> val iter3 =
    LetExp(ValDecl(NoRecursiva, IdPat "n", ConstExp(Entera 1)),
           IterExp([("n", ConstExp(Entera 1),
                     ApExp(IdExp "+", ParExp(IdExp "n", ConstExp(Entera 1))))],
                   ApExp(IdExp ">", ParExp(IdExp "n", ConstExp(Entera 10))),
                   IdExp "n")) : Expresion
- evalProg iter3;
> val it = ConstInt 11 : Valor

(* Iteracion para que devuelva error de no reconocer a n, en la inicializacion de m *)
- val iter4 = IterExp([(
                       "n", ConstExp(Entera 1), ApExp(IdExp "+", ParExp(IdExp "n", ConstExp(Entera 1))))
                   ,   ("m", IdExp "n", ApExp(IdExp "+", ParExp(IdExp "n", IdExp "m")))
                    ],
                    ApExp(IdExp ">",ParExp(IdExp "n", ConstExp(Entera 10))),
                    IdExp "n");
> val iter4 =
    IterExp([("n", ConstExp(Entera 1),
              ApExp(IdExp "+", ParExp(IdExp "n", ConstExp(Entera 1)))),
             ("m", IdExp "n", ApExp(IdExp "+", ParExp(IdExp "n", IdExp "m")))],
            ApExp(IdExp ">", ParExp(IdExp "n", ConstExp(Entera 10))),
            IdExp "n") : Expresion
- evalProg iter4;
! Uncaught exception:
! NoEstaEnElDominio  "n"


(* Iteracion que devuelve un registro *)
- val iter5 = LetExp(ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 3)),
                  IterExp([(
                            "n", ConstExp(Entera 1), ApExp(IdExp "+", ParExp(IdExp "n", ConstExp(
                                Entera 1))))
                           ],
                           ApExp(IdExp ">", ParExp(IdExp "n", ConstExp(Entera 10))),
                           RegExp[("a", IdExp "a"), ("n", IdExp "n")]
                          ));
> val iter5 =
    LetExp(ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 3)),
           IterExp([("n", ConstExp(Entera 1),
                     ApExp(IdExp "+", ParExp(IdExp "n", ConstExp(Entera 1))))],
                   ApExp(IdExp ">", ParExp(IdExp "n", ConstExp(Entera 10))),
                   RegExp [("a", IdExp "a"), ("n", IdExp "n")])) : Expresion
- evalProg iter5;
> val it = Registros [("a", ConstInt 3), ("n", ConstInt 11)] : Valor


(*********************************************************)
(********************CONDICIONAL**************************)

(* Condicional exitoso *)
- val cond1 = LetExp(ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 1)),
                  CondExp([
                           (ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 1))),
                              ConstExp(Entera 1)),
                           (ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 2))),
                              ConstExp(Entera 2))
                          ],
                          Nothing));
> val cond1 =
```

```
    LetExp(ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 1)),
           CondExp([(ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 1))),
                     ConstExp(Entera 1)),
                    (ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 2))),
                     ConstExp(Entera 2))], Nothing)) : Expresion
- evalProg cond1;
> val it = ConstInt 1 : Valor

(* Condicional exitoso, se evalua que se ejecute la primera que encuentra *)
- val cond2 = LetExp(ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 1)),
                  CondExp([
                           (ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 1))),
                              ConstExp(Entera 1)),
                           (ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 2))),
                              ConstExp(Entera 2)),
                           (ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 1))),
                              ConstExp(Entera 3))
                          ],
                          Nothing));
> val cond2 =
    LetExp(ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 1)),
           CondExp([(ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 1))),
                     ConstExp(Entera 1)),
                    (ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 2))),
                     ConstExp(Entera 2)),
                    (ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 1))),
                     ConstExp(Entera 3))], Nothing)) : Expresion
- evalProg cond2;
> val it = ConstInt 1 : Valor

(* Se evalua la ejecucion del else *)
- val cond3 = LetExp(ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 3)),
                  CondExp([
                           (ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 1))),
                              ConstExp(Entera 1)),
                           (ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 2))),
                              ConstExp(Entera 2))
                          ],
                          Something (ConstExp(Entera 3))));
> val cond3 =
    LetExp(ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 3)),
           CondExp([(ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 1))),
                     ConstExp(Entera 1)),
                    (ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 2))),
                     ConstExp(Entera 2))], Something(ConstExp(Entera 3)))) :
  Expresion
- evalProg cond3;
> val it = ConstInt 3 : Valor

(* Condicional no exitoso sin else *)
- val cond4 = LetExp(ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 4)),
                  CondExp([
                           (ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 1))),
                              ConstExp(Entera 1)),
                           (ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 2))),
                              ConstExp(Entera 2))
                          ],
                          Nothing));
> val cond4 =
    LetExp(ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 4)),
           CondExp([(ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 1))),
                     ConstExp(Entera 1)),
                    (ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 2))),
                     ConstExp(Entera 2))], Nothing)) : Expresion
- evalProg cond4;
! Uncaught exception:
! NoHayClausulaElse  "CondExp: No hay Else"

```

```
(*********************************************************)
(*********************CONCORDANCIA************************)

(* Concordancia de patrones con dominios no disyuntos *)
- val conpat1 =  ParPat(IdPat "a", ParPat(IdPat "b", IdPat "a"));
> val conpat1 = ParPat(IdPat "a", ParPat(IdPat "b", IdPat "a")) : Patron
- val conpat2 = (ParExp(ConstExp(Entera 1), ParExp(ConstExp(Entera 2), ConstExp(Entera 3))));
> val conpat2 =
    ParExp(ConstExp(Entera 1), ParExp(ConstExp(Entera 2), ConstExp(Entera 3)))
     : Expresion
- val concord1 = LetExp(ValDecl(NoRecursiva, conpat1, conpat2),
                        IdExp "a");
> val concord1 =
    LetExp(ValDecl(NoRecursiva,
                   ParPat(IdPat "a", ParPat(IdPat "b", IdPat "a")),
                   ParExp(ConstExp(Entera 1),
                          ParExp(ConstExp(Entera 2), ConstExp(Entera 3)))),
           IdExp "a") : Expresion
- evalProg concord1;
! Uncaught exception:
! DominiosNoDisyuntos

(* Concordancia de regitros con dominios no disyuntos *)
- val conreg1 = RegPat ["a", "b", "a"];
> val conreg1 = RegPat ["a", "b", "a"] : Patron
- val conreg2 = RegExp [("a", ConstExp(Entera 1)),
                        ("b", ConstExp(Entera 2))
                         ];
> val conreg2 = RegExp [("a", ConstExp(Entera 1)), ("b", ConstExp(Entera 2))] :
  Expresion
- val concord2 = LetExp(ValDecl(NoRecursiva, conreg1, conreg2),
                        IdExp "a");
> val concord2 =
    LetExp(ValDecl(NoRecursiva, RegPat ["a", "b", "a"],
                   RegExp [("a", ConstExp(Entera 1)),
                           ("b", ConstExp(Entera 2))]), IdExp "a") : Expresion
- evalProg concord2;
! Uncaught exception:
! DominiosNoDisyuntos

(* Concordancia de registros de tamannos diferentes  *)
- val conreg3 = RegPat ["a", "b"];
> val conreg3 = RegPat ["a", "b"] : Patron
- val conreg4 = RegExp [("a", ConstExp(Entera 1)),
                        ("b", ConstExp(Entera 2)),
                        ("c", ConstExp(Entera 3))
                       ];
> val conreg4 =
    RegExp [("a", ConstExp(Entera 1)), ("b", ConstExp(Entera 2)),
            ("c", ConstExp(Entera 3))] : Expresion
- val concord3 = LetExp(ValDecl(NoRecursiva, conreg3, conreg4),
                        ApExp(IdExp "+", ParExp(IdExp "a", IdExp "b")));
> val concord3 =
    LetExp(ValDecl(NoRecursiva, RegPat ["a", "b"],
                   RegExp [("a", ConstExp(Entera 1)),
                           ("b", ConstExp(Entera 2)),
                           ("c", ConstExp(Entera 3))]),
           ApExp(IdExp "+", ParExp(IdExp "a", IdExp "b"))) : Expresion
- evalProg concord3;
> val it = ConstInt 3 : Valor

(* Concordancia de registros no exitosa *)
- val conreg5 = RegPat ["a", "d"];
> val conreg5 = RegPat ["a", "d"] : Patron
- val conreg6 = RegExp [("a", ConstExp(Entera 1)),
                        ("b", ConstExp(Entera 2)),
                        ("c", ConstExp(Entera 3))
                       ];
```

```
> val conreg6 =
    RegExp [("a", ConstExp(Entera 1)), ("b", ConstExp(Entera 2)),
            ("c", ConstExp(Entera 3))] : Expresion
- val concord4 = LetExp(ValDecl(NoRecursiva, conreg5, conreg6),
                     ApExp(IdExp "+", ParExp(IdExp "a", IdExp "d")));
> val concord4 =
    LetExp(ValDecl(NoRecursiva, RegPat ["a", "d"],
                   RegExp [("a", ConstExp(Entera 1)),
                           ("b", ConstExp(Entera 2)),
                           ("c", ConstExp(Entera 3))]),
          ApExp(IdExp "+", ParExp(IdExp "a", IdExp "d"))) : Expresion
- evalProg concord4;
! Uncaught exception:
! NoEstaEnElDominio  "d"


(* Concordancia de un registro con una constante  *)
- val conreg7 = RegPat ["a", "b"];
> val conreg7 = RegPat ["a", "b"] : Patron
- val concord5 = LetExp(ValDecl(NoRecursiva, conreg7, ConstExp(Entera 8)),
                     IdExp "a");
> val concord5 =
    LetExp(ValDecl(NoRecursiva, RegPat ["a", "b"], ConstExp(Entera 8)),
           IdExp "a") : Expresion
- evalProg concord5;
! Uncaught exception:
! PatronesNoConcuerdan

(*************************************************************)
(************************COMOPAT******************************)

(* Concordancia exitosa  *)
- val comop1 = ComoPat("x", ParPat(IdPat "y", IdPat "z"));
> val comop1 = ComoPat("x", ParPat(IdPat "y", IdPat "z")) : Patron
- val comop2 = ParExp(ConstExp(Entera 2), ConstExp(Entera 3));
> val comop2 = ParExp(ConstExp(Entera 2), ConstExp(Entera 3)) : Expresion
- val compat1 = LetExp(ValDecl(NoRecursiva, comop1, comop2),
                     ParExp( ApExp(IdExp "+", ParExp(IdExp "y", IdExp "z")),
                             IdExp "x"));
> val compat1 =
    LetExp(ValDecl(NoRecursiva, ComoPat("x", ParPat(IdPat "y", IdPat "z")),
                   ParExp(ConstExp(Entera 2), ConstExp(Entera 3))),
           ParExp(ApExp(IdExp "+", ParExp(IdExp "y", IdExp "z")), IdExp "x")) :
  Expresion
- evalProg compat1;
> val it = Par(ConstInt 5, Par(ConstInt 2, ConstInt 3)) : Valor

(* Concordancia donde el se tienen nombres repetidos, error  *)
- val comop3 = ComoPat("x", ParPat(IdPat "x", IdPat "z"));
> val comop3 = ComoPat("x", ParPat(IdPat "x", IdPat "z")) : Patron
- val compat2 = LetExp(ValDecl(NoRecursiva, comop3, comop2),
                     IdExp "y");
> val compat2 =
    LetExp(ValDecl(NoRecursiva, ComoPat("x", ParPat(IdPat "x", IdPat "z")),
                   ParExp(ConstExp(Entera 2), ConstExp(Entera 3))), IdExp "y")
     : Expresion
- evalProg compat2;
! Uncaught exception:
! DominiosNoDisyuntos

(* Concordancia donde no se tiene exito *)
- val comop4 = ComoPat("x", ParPat(IdPat "y", ConstPat(Entera 1)));
> val comop4 = ComoPat("x", ParPat(IdPat "y", ConstPat(Entera 1))) : Patron
- val compat3 = LetExp(ValDecl(NoRecursiva, comop4, comop2),
                     IdExp "y");
> val compat3 =
    LetExp(ValDecl(NoRecursiva,
                   ComoPat("x", ParPat(IdPat "y", ConstPat(Entera 1))),
```

```
                  ParExp(ConstExp(Entera 2), ConstExp(Entera 3))), IdExp "y")
     : Expresion
- evalProg compat3;
! Uncaught exception:
! PatronesNoConcuerdan

(*********************************************************)
(*****DECLARACIONES COLATERALES, SECUENCIALES y LOCALES******)

- val val1 = ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 1));
> val val1 = ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 1)) : Declaracion
- val val2 = ValDecl(NoRecursiva, IdPat "b", ConstExp(Entera 2));
> val val2 = ValDecl(NoRecursiva, IdPat "b", ConstExp(Entera 2)) : Declaracion
- val val3 = ValDecl(NoRecursiva, IdPat "c", ApExp(IdExp "+", ParExp(IdExp "a", ConstExp(Entera 2))));
> val val3 =
    ValDecl(NoRecursiva, IdPat "c",
            ApExp(IdExp "+", ParExp(IdExp "a", ConstExp(Entera 2)))) :
  Declaracion
- val val4 = ValDecl(NoRecursiva, IdPat "d", ApExp(IdExp "+", ParExp(IdExp "f", ConstExp(Entera 3))));
> val val4 =
    ValDecl(NoRecursiva, IdPat "d",
            ApExp(IdExp "+", ParExp(IdExp "f", ConstExp(Entera 3)))) : Declaracion

(***** Declaraciones colaterales *****)

(* Declaracion colateral exitosa *)
- val colat1 = LetExp(AndDecl(val1, val2),
                  ApExp(IdExp "+", ParExp(IdExp "a", IdExp "b")));
> val colat1 =
    LetExp(AndDecl(ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 1)),
                   ValDecl(NoRecursiva, IdPat "b", ConstExp(Entera 2))),
           ApExp(IdExp "+", ParExp(IdExp "a", IdExp "b"))) : Expresion
- evalProg colat1;
> val it = ConstInt 3 : Valor

(* Declaracion colateral exitosa  *)
- val colat2 = LetExp(AndDecl(val1, val2),
                   IdExp "a");
> val colat2 =
    LetExp(AndDecl(ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 1)),
                   ValDecl(NoRecursiva, IdPat "b", ConstExp(Entera 2))),
           IdExp "a") : Expresion
- evalProg colat2;
> val it = ConstInt 1 : Valor

(* Declaracion colateral no exitosa  *)
- val colat3 = LetExp(AndDecl(val1, val3),
                   IdExp "c");
> val colat3 =
    LetExp(AndDecl(ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 1)),
                   ValDecl(NoRecursiva, IdPat "c",
                           ApExp(IdExp "+",
                                 ParExp(IdExp "a", ConstExp(Entera 2))))),
           IdExp "c") : Expresion
- evalProg colat3;
! Uncaught exception:
! NoEstaEnElDominio  "a"

(* Declaracion colateral no exitosa  *)
- val colat4 = LetExp(AndDecl(val1, val1),
                   IdExp "a");
> val colat4 =
    LetExp(AndDecl(ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 1)),
                   ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 1))),
           IdExp "a") : Expresion
- evalProg colat4;
! Uncaught exception:
! DominiosNoDisyuntos
```

```
485
486 (****** Declaraciones secuenciales *****)
487
488 (* Declaracion secuencial exitosa *)
489 - val sec1 = LetExp(SecDecl(val1, val2),
490                  ApExp(IdExp "+", ParExp(IdExp "a", IdExp "b")));
491 > val sec1 =
492     LetExp(SecDecl(ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 1)),
493                    ValDecl(NoRecursiva, IdPat "b", ConstExp(Entera 2))),
494            ApExp(IdExp "+", ParExp(IdExp "a", IdExp "b"))) : Expresion
495 - evalProg sec1;
496 > val it = ConstInt 3 : Valor
497
498 (* Declaracion secuencial exitosa *)
499 - val sec2 = LetExp(SecDecl(val1, val3),
500                  IdExp "c");
501 > val sec2 =
502     LetExp(SecDecl(ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 1)),
503                    ValDecl(NoRecursiva, IdPat "c",
504                            ApExp(IdExp "+",
505                                  ParExp(IdExp "a", ConstExp(Entera 2))))),
506            IdExp "c") : Expresion
507 - evalProg sec2;
508 > val it = ConstInt 3 : Valor
509
510
511 (***** Declaraciones locales *****)
512
513 (* Declaracion local exitosa *)
514 - val loc1 = LetExp(LocalDecl(val1, val3),
515                  IdExp "c");
516 > val loc1 =
517     LetExp(LocalDecl(ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 1)),
518                      ValDecl(NoRecursiva, IdPat "c",
519                              ApExp(IdExp "+",
520                                    ParExp(IdExp "a", ConstExp(Entera 2))))),
521            IdExp "c") : Expresion
522
523 (* Declaracion local no exitosa *)
524 - val loc2 = LetExp(LocalDecl(val1, val4),
525                  IdExp "d");
526 > val loc2 =
527     LetExp(LocalDecl(ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 1)),
528                      ValDecl(NoRecursiva, IdPat "d",
529                              ApExp(IdExp "+",
530                                    ParExp(IdExp "f", ConstExp(Entera 3))))),
531            IdExp "d") : Expresion
532 - evalProg loc2;
533 ! Uncaught exception:
534 ! NoEstaEnElDominio  "f"
535
536 (* Declaracion local no exitosa *)
537 - val loc3 = LetExp(LocalDecl(val1, val3),
538                  IdExp "a");
539 > val loc3 =
540     LetExp(LocalDecl(ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 1)),
541                      ValDecl(NoRecursiva, IdPat "c",
542                              ApExp(IdExp "+",
543                                    ParExp(IdExp "a", ConstExp(Entera 2))))),
544            IdExp "a") : Expresion
545 - evalProg loc3;
546 ! Uncaught exception:
547 ! NoEstaEnElDominio  "a"
548
549 (*********************************************************)
550 (****************DECLARACIONES RECURSIVAS*****************)
551
552 (* Factorial recursivo exitoso *)
```

```
- val fact1 = LetExp(ValDecl(Recursiva, IdPat "fact",
                    AbsExp[(IdPat "f",
                        CondExp([
                                 (ApExp(IdExp "=", ParExp(IdExp "f", ConstExp(Entera 0))),
                                   ConstExp(Entera 1)),
                                 (ApExp(IdExp "=", ParExp(IdExp "f", ConstExp(Entera 1))),
                                   ConstExp(Entera 1))
                                ],
                                Something
                                 (ApExp(IdExp "*", ParExp(IdExp "f", ApExp(IdExp "fact",
                                                                           ApExp(IdExp "-", ParExp(
                                                                               IdExp "f", ConstExp(
                                                                               Entera 1))))
                                                                    )))

                                  ))
                            ]),
                  ApExp(IdExp "fact", ConstExp(Entera 5)));
> val fact1 =
    LetExp(ValDecl(Recursiva, IdPat "fact",
                   AbsExp [(IdPat "f",
                            CondExp([(ApExp(IdExp "=",
                                            ParExp(IdExp "f",
                                                   ConstExp(Entera 0))),
                                      ConstExp(Entera 1)),
                                     (ApExp(IdExp "=",
                                            ParExp(IdExp "f",
                                                   ConstExp(Entera 1))),
                                      ConstExp(Entera 1))],
                                    Something(ApExp(IdExp "*",
                                                    ParExp(IdExp "f",
                                                           ApExp(IdExp "fact",
                                                                 ApExp(IdExp "-",
                                                                       ParExp(#,
                                                                              #)))))))]),
           ApExp(IdExp "fact", ConstExp(Entera 5))) : Expresion
- evalProg fact1;
> val it = ConstInt 120 : Valor

(* Factorial recursivo no exitoso *)
- val fact2 = LetExp(ValDecl(NoRecursiva, IdPat "fact",
                    AbsExp[(IdPat "f",
                        CondExp([
                                 (ApExp(IdExp "=", ParExp(IdExp "f", ConstExp(Entera 0))),
                                   ConstExp(Entera 1)),
                                 (ApExp(IdExp "=", ParExp(IdExp "f", ConstExp(Entera 1))),
                                   ConstExp(Entera 1))
                                ],
                                Something
                                 (ApExp(IdExp "*", ParExp(IdExp "f", ApExp(IdExp "fact",
                                                                           ApExp(IdExp "-", ParExp(
                                                                               IdExp "f", ConstExp(
                                                                               Entera 1))))
                                                                    )))

                                  ))
                            ]),
                  ApExp(IdExp "fact", ConstExp(Entera 5)));
> val fact2 =
    LetExp(ValDecl(NoRecursiva, IdPat "fact",
                   AbsExp [(IdPat "f",
                            CondExp([(ApExp(IdExp "=",
                                            ParExp(IdExp "f",
                                                   ConstExp(Entera 0))),
                                      ConstExp(Entera 1)),
                                     (ApExp(IdExp "=",
                                            ParExp(IdExp "f",
                                                   ConstExp(Entera 1))),
```

```
                                        ConstExp(Entera 1))],
                                    Something(ApExp(IdExp "*",
                                                    ParExp(IdExp "f",
                                                           ApExp(IdExp "fact",
                                                                 ApExp(IdExp "-",
                                                                       ParExp(#,
                                                                              #)))))))])),
          ApExp(IdExp "fact", ConstExp(Entera 5))) : Expresion
- evalProg fact2;
! Uncaught exception:
! NoEstaEnElDominio  "fact"

(* Funciones mutuamente recursivas exitosas *)
- val mut1 = LetExp(ValDecl(Recursiva, ParPat(IdPat "f", IdPat "g"),
                   (ParExp(
                    AbsExp[(IdPat "x",
                            ApExp(IdExp "+", ParExp(ConstExp(Entera 1),
                                                    ApExp(IdExp "g",
                                                          ApExp(IdExp "-", ParExp(IdExp "x",
                                                                                  ConstExp(Entera 1))))
                                                                                      )
                           ))],
                    AbsExp[(IdPat "y",
                            IfExp( ApExp(IdExp "<", ParExp(IdExp "y",
                                                           ConstExp(Entera 0))),
                                   IdExp "y",
                                   ApExp(IdExp "+", ParExp(ConstExp(Entera 1),
                                                           ApExp(IdExp "f",
                                                                 ApExp(IdExp "-",ParExp(IdExp "y"
                                                                                        ,
                                                                                                ConstExp(
                                                                                                    Entera
                                                                                                     1)
                                                                                                )
                                                                                            )
                                                                                        )
                                                                                    )
                                                                                )
                                                                            )
                                                                        )
                                                                    )
                                                                )
                                                            )
                                                        )

                            )]
                  ))),
                  ApExp(IdExp "f", ConstExp(Entera 10)));
> val mut1 =
    LetExp(ValDecl(Recursiva, ParPat(IdPat "f", IdPat "g"),
                   ParExp(AbsExp [(IdPat "x",
                                   ApExp(IdExp "+",
                                         ParExp(ConstExp(Entera 1),
                                                ApExp(IdExp "g",
                                                      ApExp(IdExp "-",
                                                            ParExp(IdExp "x",
                                                                   ConstExp#)))))))],
                           AbsExp [(IdPat "y",
                                   IfExp(ApExp(IdExp "<",
                                               ParExp(IdExp "y",
                                                      ConstExp(Entera 0))),
                                         IdExp "y",
                                         ApExp(IdExp "+",
                                               ParExp(ConstExp(Entera 1),
                                                      ApExp(IdExp "f",
                                                            ApExp(IdExp "-",
                                                                  ParExp#)))))))])),
          ApExp(IdExp "f", ConstExp(Entera 10))) : Expresion
- evalProg mut1;
> val it = ConstInt 10 : Valor

(* Funciones mutuamente recursivas no exitosas *)
- val mut2 = LetExp(ValDecl(NoRecursiva, ParPat(IdPat "f", IdPat "g"),
```

```
                    (ParExp(
                     AbsExp[(IdPat "x",
                             ApExp(IdExp "+", ParExp(ConstExp(Entera 1),
                                                     ApExp(IdExp "g",
                                                           ApExp(IdExp "-", ParExp(IdExp "x",
                                                                                   ConstExp(Entera 1))))
                                                                                       )
                            ))],
                     AbsExp[(IdPat "y",
                             IfExp( ApExp(IdExp "<", ParExp(IdExp "y",
                                                            ConstExp(Entera 0))),
                                    IdExp "y",
                                    ApExp(IdExp "+", ParExp(ConstExp(Entera 1),
                                                            ApExp(IdExp "f",
                                                                  ApExp(IdExp "-",ParExp(IdExp "y"
                                                                                         ,
                                                                                                 ConstExp(
                                                                                                     Entera
                                                                                                      1)
                                                                                                 )
                                                                                             )
                                                                                         )
                                                                                     )
                                                                                 )
                                                                             )
                                                                         )
                                                                     )
                                                                 )
                                                             )
                                                         )
                                                     )
                                                 )
                                             )

                             )]
                  ))),
                  ApExp(IdExp "f", ConstExp(Entera 10)));
> val mut2 =
    LetExp(ValDecl(NoRecursiva, ParPat(IdPat "f", IdPat "g"),
                   ParExp(AbsExp [(IdPat "x",
                                   ApExp(IdExp "+",
                                         ParExp(ConstExp(Entera 1),
                                                ApExp(IdExp "g",
                                                      ApExp(IdExp "-",
                                                            ParExp(IdExp "x",
                                                                   ConstExp#))))))],
                           AbsExp [(IdPat "y",
                                    IfExp(ApExp(IdExp "<",
                                                ParExp(IdExp "y",
                                                       ConstExp(Entera 0))),
                                          IdExp "y",
                                          ApExp(IdExp "+",
                                                ParExp(ConstExp(Entera 1),
                                                       ApExp(IdExp "f",
                                                             ApExp(IdExp "-",
                                                                   ParExp#))))))])),
           ApExp(IdExp "f", ConstExp(Entera 10))) : Expresion
- evalProg mut2;
! Uncaught exception:
! NoEstaEnElDominio  "g"
```