

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	2
2. Reflexiones sobre la experiencia	2
2.1. Martín Flores	2
2.2. Willard Zamora	3
2.3. Plano grupal	3
3. Documentación del proyecto	3
3.1. Representación utilizada para los registros y cualquier otro valor semántico	3
3.2. La solución dada al manejo de registros (expresiones-registro, accesos a campos de un registro)	3
3.3. La solución dada a la evaluación de la expresión iterativa.	3
3.4. La solución dada a la evaluación de la expresión condicional generalizada	4
3.5. La solución dada a las extensiones hechas a los patrones (patrones estratificados ['as'], patrones-registro).	4
3.6. La solución dada a la combinación de ambientes con dominios disyuntos (función < >).	5
3.7. Otras modificaciones hechas al intérprete	5
3.8. Casos de prueba y resultados observados	6

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 intérprete 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 venía 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 programación 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 principalmente 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 comprensión 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. Representación 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 Rexp 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 Rexp se hace uso del nuevo tipo valor Registros y la función existente de map_ambiente, aplicando a cada par (Identificador, Expresion) la función que evalúa la expresión 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 expresión 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 evalúa trueExp. Durante el proceso del ciclo se va haciendo 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 evalúa la condición (`cond`) y en el caso que el resultado sea verdadero (`ConstBool true`) se procede entonces a evaluar la expresión (`expresion`). Si el resultado de la evaluación de `cond` sea falso (`ConstBool false`) entonces se evalúa 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 consumiendo 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
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 parámetro el resto de la lista de identificadores.

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 siguientes 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 disyuntos 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  (*****REGISTROS*****)
2
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      : Expression
8  - evalProg registro1;
9  > val it = Registros [("a", ConstInt 1), ("b", ConstInt 2)] : Valor
10
11  (* Campos repetidos *)
12  - val registro2 = RegExp [("x", ConstExp(Entera 3)),
13                        ("x", ConstExp(Entera 4))];
14  > val registro2 = RegExp [("x", ConstExp(Entera 3)), ("x", ConstExp(Entera 4))]
15      : Expression
16  - evalProg registro2;
17  > val it = Registros [("x", ConstInt 3), ("x", ConstInt 4)] : Valor
18
19  (* Campo existente *)
20  - val regA = CampoExp(registro1, "a");
21  > val regA =
22      CampoExp(RegExp [("a", ConstExp(Entera 1)), ("b", ConstExp(Entera 2))],
23              "a") : Expression
24  - evalProg regA;
25  > val it = ConstInt 1 : Valor
26
27  (* Campo inexistente *)
28  - val regC = CampoExp(registro1, "c");
29  > val regC =
30      CampoExp(RegExp [("a", ConstExp(Entera 1)), ("b", ConstExp(Entera 2))],
31              "c") : Expression
32  - evalProg regC;
33  ! Uncaught exception:
34  ! NoEstaEnElDominio "c"
35
36
37  - val pruFun = LetExp(ValDecl(NoRekursiva, IdPat "a", ConstExp(Entera 1)),
38                      regFun);
39  > val pruFun =
40      LetExp(ValDecl(NoRekursiva, IdPat "a", ConstExp(Entera 1)),
41            IfExp(ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 1))),
42                RegExp [("a", ConstExp(Entera 1)), ("b", ConstExp(Entera 2))],
43                RegExp [("x", ConstExp(Entera 3)),
44                        ("x", ConstExp(Entera 4))])) : Expression
45  - evalProg pruFun;
46  > val it = Registros [("a", ConstInt 1), ("b", ConstInt 2)] : Valor
47
48
49  (* Acceso a campos donde el registro es el resultado de una expresion *)
50
51  - val regA1 = CampoExp(pruFun, "a");
52  > val regA1 =
53      CampoExp(LetExp(ValDecl(NoRekursiva, IdPat "a", ConstExp(Entera 1)),
54                    IfExp(ApExp(IdExp "=",
55                                ParExp(IdExp "a", ConstExp(Entera 1))),
56                        RegExp [("a", ConstExp(Entera 1)),
57                                ("b", ConstExp(Entera 2))],
58                        RegExp [("x", ConstExp(Entera 3)),
59                                ("x", ConstExp(Entera 4))])), "a") :
60      Expression
61  - evalProg regA1;
62  > val it = ConstInt 1 : Valor
63
```

```

64 - val regC1 = CampoExp(pruFun, "c");
65 > val regC1 =
66     CampoExp(LetExp(ValDecl(NoRekursiva, IdPat "a", ConstExp(Entera 1)),
67         IfExp(ApExp(IdExp "=",
68             ParExp(IdExp "a", ConstExp(Entera 1))),
69             RegExp [( "a", ConstExp(Entera 1)),
70                 ("b", ConstExp(Entera 2))],
71             RegExp [( "x", ConstExp(Entera 3)),
72                 ("x", ConstExp(Entera 4))])), "c") :
73     Expresion
74 - evalProg regC1;
75 ! Uncaught exception:
76 ! NoEstaEnElDominio "c"
77
78
79 (*****
80 (*****ITERACION*****
81
82 (* Iteracion exitosa *)
83 - val iter1 = LetExp(ValDecl(NoRekursiva, IdPat "a", ConstExp(Entera 6)),
84     LetExp(ValDecl(NoRekursiva, IdPat "fact",
85         AbsExp [(IdPat "k",
86             IterExp([
87                 ("n", IdExp "k", ApExp(IdExp "-", ParExp(IdExp "n", ConstExp(Entera 1))))
88                 , ("product", ConstExp(Entera 1), ApExp(IdExp "*", ParExp(IdExp "product", IdExp "
89                     n"))))
90             ],
91             ApExp(IdExp "=", ParExp(IdExp "n", ConstExp(Entera 0))),
92             IdExp "product"))
93         ], ApExp(IdExp "fact", IdExp "a")));
94 > val iter1 =
95     LetExp(ValDecl(NoRekursiva, IdPat "a", ConstExp(Entera 6)),
96         LetExp(ValDecl(NoRekursiva, IdPat "fact",
97             AbsExp [(IdPat "k",
98                 IterExp([("n", IdExp "k",
99                     ApExp(IdExp "-",
100                         ParExp(IdExp "n",
101                             ConstExp(Entera 1)))))
102                     ("product", ConstExp(Entera 1),
103                         ApExp(IdExp "*",
104                             ParExp(IdExp "product",
105                                 IdExp "n")))]),
106                     ApExp(IdExp "=",
107                         ParExp(IdExp "n",
108                             ConstExp(Entera 0))),
109                     IdExp "product"))]),
110             ApExp(IdExp "fact", IdExp "a")) : Expresion
111 - evalProg iter1;
112 > val it = ConstInt 720 : Valor
113
114 (* Iteracion no exitosa, variable de iteracion repetida *)
115 - val iter2 = LetExp(ValDecl(NoRekursiva, IdPat "a", ConstExp(Entera 6)),
116     LetExp(ValDecl(NoRekursiva, IdPat "fact",
117         AbsExp [(IdPat "k",
118             IterExp([
119                 "n", IdExp "k", ApExp(IdExp "-", ParExp(IdExp "n", ConstExp(Entera 1))))
120                 , ("n", ConstExp(Entera 1), ApExp(IdExp "*", ParExp(IdExp "n", IdExp "n"))))
121             ],
122             ApExp(IdExp "=", ParExp(IdExp "n", ConstExp(Entera 0))),
123             IdExp "n"))
124         ], ApExp(IdExp "fact", IdExp "a")));
125 > val iter2 =
126     LetExp(ValDecl(NoRekursiva, IdPat "a", ConstExp(Entera 6)),
127         LetExp(ValDecl(NoRekursiva, IdPat "fact",
128             AbsExp [(IdPat "k",
129                 IterExp([("n", IdExp "k",

```

```
131         ApExp(IdExp "-",
132               ParExp(IdExp "n",
133                     ConstExp(Entera 1))))),
134       ("n", ConstExp(Entera 1),
135        ApExp(IdExp "*",
136              ParExp(IdExp "n",
137                    IdExp "n")))),
138       ApExp(IdExp "=",
139             ParExp(IdExp "n",
140                   ConstExp(Entera 0))),
141             IdExp "n")))),
142       ApExp(IdExp "fact", IdExp "a"))) : Expression
143 - evalProg iter2;
144 ! Uncaught exception:
145 ! DominiosNoDisyuntos
146
147 (* Iteracion para que devuelva el n mas recientemente definido *)
148 - val iter3 = LetExp(ValDecl(NoRekursiva, IdPat "n", ConstExp(Entera 1)),
149                     IterExp([(
150                         "n", ConstExp(Entera 1), ApExp(IdExp "+", ParExp(IdExp "n", ConstExp(
151                             Entera 1))))
152                         ],
153                         ApExp(IdExp ">", ParExp(IdExp "n", ConstExp(Entera 10))),
154                         IdExp "n"
155                     ));
156 > val iter3 =
157     LetExp(ValDecl(NoRekursiva, IdPat "n", ConstExp(Entera 1)),
158           IterExp([( "n", ConstExp(Entera 1),
159                       ApExp(IdExp "+", ParExp(IdExp "n", ConstExp(Entera 1))))],
160                       ApExp(IdExp ">", ParExp(IdExp "n", ConstExp(Entera 10))),
161                       IdExp "n")) : Expression
162 - evalProg iter3;
163 > val it = ConstInt 11 : Valor
164
165 (* Iteracion para que devuelva error de no reconocer a n, en la inicializacion de m *)
166 - val iter4 = IterExp([(
167     "n", ConstExp(Entera 1), ApExp(IdExp "+", ParExp(IdExp "n", ConstExp(Entera 1))))
168     , ("m", IdExp "n", ApExp(IdExp "+", ParExp(IdExp "n", IdExp "m")))
169     ],
170     ApExp(IdExp ">", ParExp(IdExp "n", ConstExp(Entera 10))),
171     IdExp "n");
172 > val iter4 =
173     IterExp([( "n", ConstExp(Entera 1),
174                 ApExp(IdExp "+", ParExp(IdExp "n", ConstExp(Entera 1))))],
175                 ("m", IdExp "n", ApExp(IdExp "+", ParExp(IdExp "n", IdExp "m")))],
176                 ApExp(IdExp ">", ParExp(IdExp "n", ConstExp(Entera 10))),
177                 IdExp "n") : Expression
178 - evalProg iter4;
179 ! Uncaught exception:
180 ! NoEstaEnElDominio "n"
181
182 (* Iteracion que devuelve un registro *)
183 - val iter5 = LetExp(ValDecl(NoRekursiva, IdPat "a", ConstExp(Entera 3)),
184                     IterExp([(
185                         "n", ConstExp(Entera 1), ApExp(IdExp "+", ParExp(IdExp "n", ConstExp(
186                             Entera 1))))
187                         ],
188                         ApExp(IdExp ">", ParExp(IdExp "n", ConstExp(Entera 10))),
189                         RegExp[("a", IdExp "a"), ("n", IdExp "n")]
190                     ));
191 > val iter5 =
192     LetExp(ValDecl(NoRekursiva, IdPat "a", ConstExp(Entera 3)),
193           IterExp([( "n", ConstExp(Entera 1),
194                       ApExp(IdExp "+", ParExp(IdExp "n", ConstExp(Entera 1))))],
195                       ApExp(IdExp ">", ParExp(IdExp "n", ConstExp(Entera 10))),
196                       RegExp [("a", IdExp "a"), ("n", IdExp "n")])) : Expression
197 - evalProg iter5;
```



```

197 > val it = Registros [("a", ConstInt 3), ("n", ConstInt 11)] : Valor
198
199
200 (*****
201 (*****CONDICIONAL*****
202
203 (* Condicional exitoso *)
204 - val cond1 = LetExp(ValDecl(NoRekursiva, IdPat "a", ConstExp(Entera 1)),
205                      CondExp([
206                          (ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 1))),
207                            ConstExp(Entera 1)),
208                          (ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 2))),
209                            ConstExp(Entera 2))
210                      ],
211                      Nothing));
212 > val cond1 =
213     LetExp(ValDecl(NoRekursiva, IdPat "a", ConstExp(Entera 1)),
214           CondExp([ (ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 1))),
215                        ConstExp(Entera 1)),
216                     (ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 2))),
217                       ConstExp(Entera 2)) ], Nothing)) : Expresion
218 - evalProg cond1;
219 > val it = ConstInt 1 : Valor
220
221 (* Condicional exitoso, se evalua que se ejecute la primera que encuentra *)
222 - val cond2 = LetExp(ValDecl(NoRekursiva, IdPat "a", ConstExp(Entera 1)),
223                      CondExp([
224                          (ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 1))),
225                            ConstExp(Entera 1)),
226                          (ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 2))),
227                            ConstExp(Entera 2)),
228                          (ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 1))),
229                            ConstExp(Entera 3))
230                      ],
231                      Nothing));
232 > val cond2 =
233     LetExp(ValDecl(NoRekursiva, IdPat "a", ConstExp(Entera 1)),
234           CondExp([ (ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 1))),
235                        ConstExp(Entera 1)),
236                     (ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 2))),
237                       ConstExp(Entera 2)),
238                     (ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 1))),
239                       ConstExp(Entera 3)) ], Nothing)) : Expresion
240 - evalProg cond2;
241 > val it = ConstInt 1 : Valor
242
243 (* Se evalua la ejecucion del else *)
244 - val cond3 = LetExp(ValDecl(NoRekursiva, IdPat "a", ConstExp(Entera 3)),
245                      CondExp([
246                          (ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 1))),
247                            ConstExp(Entera 1)),
248                          (ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 2))),
249                            ConstExp(Entera 2))
250                      ],
251                      Something (ConstExp(Entera 3))));
252 > val cond3 =
253     LetExp(ValDecl(NoRekursiva, IdPat "a", ConstExp(Entera 3)),
254           CondExp([ (ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 1))),
255                        ConstExp(Entera 1)),
256                     (ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 2))),
257                       ConstExp(Entera 2)) ], Something (ConstExp(Entera 3)))) :
258     Expresion
259 - evalProg cond3;
260 > val it = ConstInt 3 : Valor
261
262 (* Condicional no exitoso sin else *)
263 - val cond4 = LetExp(ValDecl(NoRekursiva, IdPat "a", ConstExp(Entera 4)),
264                      CondExp([

```

```
265         (ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 1))),
266           ConstExp(Entera 1)),
267         (ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 2))),
268           ConstExp(Entera 2))
269     ],
270     Nothing));
271 > val cond4 =
272     LetExp(ValDecl(NoRecurSiva, IdPat "a", ConstExp(Entera 4)),
273       CondExp([(ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 1))),
274         ConstExp(Entera 1)),
275         (ApExp(IdExp "=", ParExp(IdExp "a", ConstExp(Entera 2))),
276           ConstExp(Entera 2))], Nothing)) : Expression
277 - evalProg cond4;
278 ! Uncaught exception:
279 ! NoHayClausulaElse "CondExp:_No_hay_Else"
280
281 (*****
282 (*****CONCORDANCIA*****
283
284 (* Concordancia de patrones con dominios no disyuntos *)
285 - val conpat1 = ParPat(IdPat "a", ParPat(IdPat "b", IdPat "a"));
286 > val conpat1 = ParPat(IdPat "a", ParPat(IdPat "b", IdPat "a")) : Patron
287 - val conpat2 = (ParExp(ConstExp(Entera 1), ParExp(ConstExp(Entera 2), ConstExp(Entera 3))));
288 > val conpat2 =
289     ParExp(ConstExp(Entera 1), ParExp(ConstExp(Entera 2), ConstExp(Entera 3)))
290     : Expression
291 - val concord1 = LetExp(ValDecl(NoRecurSiva, conpat1, conpat2),
292     IdExp "a");
293 > val concord1 =
294     LetExp(ValDecl(NoRecurSiva,
295       ParPat(IdPat "a", ParPat(IdPat "b", IdPat "a")),
296       ParExp(ConstExp(Entera 1),
297         ParExp(ConstExp(Entera 2), ConstExp(Entera 3)))),
298     IdExp "a") : Expression
299 - evalProg concord1;
300 ! Uncaught exception:
301 ! DominiosNoDisyuntos
302
303 (* Concordancia de registros con dominios no disyuntos *)
304 - val conreg1 = RegPat ["a", "b", "a"];
305 > val conreg1 = RegPat ["a", "b", "a"] : Patron
306 - val conreg2 = RegExp [("a", ConstExp(Entera 1)),
307     ("b", ConstExp(Entera 2))
308     ];
309 > val conreg2 = RegExp [("a", ConstExp(Entera 1)), ("b", ConstExp(Entera 2))] :
310     Expression
311 - val concord2 = LetExp(ValDecl(NoRecurSiva, conreg1, conreg2),
312     IdExp "a");
313 > val concord2 =
314     LetExp(ValDecl(NoRecurSiva, RegPat ["a", "b", "a"],
315       RegExp [("a", ConstExp(Entera 1)),
316         ("b", ConstExp(Entera 2))]), IdExp "a") : Expression
317 - evalProg concord2;
318 ! Uncaught exception:
319 ! DominiosNoDisyuntos
320
321 (* Concordancia de registros de tamannos diferentes *)
322 - val conreg3 = RegPat ["a", "b"];
323 > val conreg3 = RegPat ["a", "b"] : Patron
324 - val conreg4 = RegExp [("a", ConstExp(Entera 1)),
325     ("b", ConstExp(Entera 2)),
326     ("c", ConstExp(Entera 3))
327     ];
328 > val conreg4 =
329     RegExp [("a", ConstExp(Entera 1)), ("b", ConstExp(Entera 2)),
330     ("c", ConstExp(Entera 3))] : Expression
331 - val concord3 = LetExp(ValDecl(NoRecurSiva, conreg3, conreg4),
332     ApExp(IdExp "+", ParExp(IdExp "a", IdExp "b")));
```

```

333 > val concord3 =
334     LetExp(ValDecl(NoRecurSiva, RegPat ["a", "b"],
335         RegExp [("a", ConstExp(Entera 1)),
336             ("b", ConstExp(Entera 2)),
337             ("c", ConstExp(Entera 3))]),
338         ApExp(IdExp "+", ParExp(IdExp "a", IdExp "b"))) : Expresion
339 - evalProg concord3;
340 > val it = ConstInt 3 : Valor
341
342 (* Concordancia de registros no exitosa *)
343 - val conreg5 = RegPat ["a", "d"];
344 > val conreg5 = RegPat ["a", "d"] : Patron
345 - val conreg6 = RegExp [("a", ConstExp(Entera 1)),
346     ("b", ConstExp(Entera 2)),
347     ("c", ConstExp(Entera 3))
348 ];
349 > val conreg6 =
350     RegExp [("a", ConstExp(Entera 1)), ("b", ConstExp(Entera 2)),
351     ("c", ConstExp(Entera 3))] : Expresion
352 - val concord4 = LetExp(ValDecl(NoRecurSiva, conreg5, conreg6),
353     ApExp(IdExp "+", ParExp(IdExp "a", IdExp "d")));
354 > val concord4 =
355     LetExp(ValDecl(NoRecurSiva, RegPat ["a", "d"],
356         RegExp [("a", ConstExp(Entera 1)),
357             ("b", ConstExp(Entera 2)),
358             ("c", ConstExp(Entera 3))]),
359         ApExp(IdExp "+", ParExp(IdExp "a", IdExp "d"))) : Expresion
360 - evalProg concord4;
361 ! Uncaught exception:
362 ! NoEstaEnElDominio "d"
363
364
365 (* Concordancia de un registro con una constante *)
366 - val conreg7 = RegPat ["a", "b"];
367 > val conreg7 = RegPat ["a", "b"] : Patron
368 - val concord5 = LetExp(ValDecl(NoRecurSiva, conreg7, ConstExp(Entera 8)),
369     IdExp "a");
370 > val concord5 =
371     LetExp(ValDecl(NoRecurSiva, RegPat ["a", "b"], ConstExp(Entera 8)),
372     IdExp "a") : Expresion
373 - evalProg concord5;
374 ! Uncaught exception:
375 ! PatronesNoConcuerdan
376
377 (*****
378 (*****COMOPAT*****
379
380 (* Concordancia exitosa *)
381 - val comop1 = ComoPat("x", ParPat(IdPat "y", IdPat "z"));
382 > val comop1 = ComoPat("x", ParPat(IdPat "y", IdPat "z")) : Patron
383 - val comop2 = ParExp(ConstExp(Entera 2), ConstExp(Entera 3));
384 > val comop2 = ParExp(ConstExp(Entera 2), ConstExp(Entera 3)) : Expresion
385 - val compat1 = LetExp(ValDecl(NoRecurSiva, comop1, comop2),
386     ParExp(ApExp(IdExp "+", ParExp(IdExp "y", IdExp "z")),
387     IdExp "x"));
388 > val compat1 =
389     LetExp(ValDecl(NoRecurSiva, ComoPat("x", ParPat(IdPat "y", IdPat "z")),
390     ParExp(ConstExp(Entera 2), ConstExp(Entera 3))),
391     ParExp(ApExp(IdExp "+", ParExp(IdExp "y", IdExp "z")), IdExp "x")) :
392     Expresion
393 - evalProg compat1;
394 > val it = Par(ConstInt 5, Par(ConstInt 2, ConstInt 3)) : Valor
395
396 (* Concordancia donde el se tienen nombres repetidos, error *)
397 - val comop3 = ComoPat("x", ParPat(IdPat "x", IdPat "z"));
398 > val comop3 = ComoPat("x", ParPat(IdPat "x", IdPat "z")) : Patron
399 - val compat2 = LetExp(ValDecl(NoRecurSiva, comop3, comop2),
400     IdExp "y");

```

```
401 > val compat2 =
402     LetExp(ValDecl(NoRecursiva, ComoPat("x", ParPat(IdPat "x", IdPat "z"))),
403           ParExp(ConstExp(Entera 2), ConstExp(Entera 3))), IdExp "y")
404     : Expresion
405 - evalProg compat2;
406 ! Uncaught exception:
407 ! DominiosNoDisyuntos
408
409 (* Concordancia donde no se tiene exito *)
410 - val comop4 = ComoPat("x", ParPat(IdPat "y", ConstPat(Entera 1)));
411 > val comop4 = ComoPat("x", ParPat(IdPat "y", ConstPat(Entera 1))) : Patron
412 - val compat3 = LetExp(ValDecl(NoRecursiva, comop4, comop2),
413                       IdExp "y");
414 > val compat3 =
415     LetExp(ValDecl(NoRecursiva,
416                   ComoPat("x", ParPat(IdPat "y", ConstPat(Entera 1))),
417                   ParExp(ConstExp(Entera 2), ConstExp(Entera 3))), IdExp "y")
418     : Expresion
419 - evalProg compat3;
420 ! Uncaught exception:
421 ! PatronesNoConcuerdan
422
423 (***** Declaraciones colaterales, secuenciales y locales *****)
424 (*****DECLARACIONES COLATERALES, SECUENCIALES y LOCALES*****)
425
426 - val val1 = ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 1));
427 > val val1 = ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 1)) : Declaracion
428 - val val2 = ValDecl(NoRecursiva, IdPat "b", ConstExp(Entera 2));
429 > val val2 = ValDecl(NoRecursiva, IdPat "b", ConstExp(Entera 2)) : Declaracion
430 - val val3 = ValDecl(NoRecursiva, IdPat "c", ApExp(IdExp "+", ParExp(IdExp "a", ConstExp(Entera 2))));
431 > val val3 =
432     ValDecl(NoRecursiva, IdPat "c",
433             ApExp(IdExp "+", ParExp(IdExp "a", ConstExp(Entera 2)))) :
434     Declaracion
435 - val val4 = ValDecl(NoRecursiva, IdPat "d", ApExp(IdExp "+", ParExp(IdExp "f", ConstExp(Entera 3))));
436 > val val4 =
437     ValDecl(NoRecursiva, IdPat "d",
438             ApExp(IdExp "+", ParExp(IdExp "f", ConstExp(Entera 3)))) : Declaracion
439
440 (***** Declaraciones colaterales *****)
441
442 (* Declaracion colateral exitosa *)
443 - val colat1 = LetExp(AndDecl(val1, val2),
444                     ApExp(IdExp "+", ParExp(IdExp "a", IdExp "b")));
445 > val colat1 =
446     LetExp(AndDecl(ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 1)),
447                   ValDecl(NoRecursiva, IdPat "b", ConstExp(Entera 2))),
448           ApExp(IdExp "+", ParExp(IdExp "a", IdExp "b"))) : Expresion
449 - evalProg colat1;
450 > val it = ConstInt 3 : Valor
451
452 (* Declaracion colateral exitosa *)
453 - val colat2 = LetExp(AndDecl(val1, val2),
454                     IdExp "a");
455 > val colat2 =
456     LetExp(AndDecl(ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 1)),
457                   ValDecl(NoRecursiva, IdPat "b", ConstExp(Entera 2))),
458           IdExp "a") : Expresion
459 - evalProg colat2;
460 > val it = ConstInt 1 : Valor
461
462 (* Declaracion colateral no exitosa *)
463 - val colat3 = LetExp(AndDecl(val1, val3),
464                     IdExp "c");
465 > val colat3 =
466     LetExp(AndDecl(ValDecl(NoRecursiva, IdPat "a", ConstExp(Entera 1)),
467                   ValDecl(NoRecursiva, IdPat "c",
468                         ApExp(IdExp "+",
```

```

469                                     ParExp(IdExp "a", ConstExp(Entera 2))))),
470         IdExp "c") : Expression
471 - evalProg colat3;
472 ! Uncaught exception:
473 ! NoEstaEnElDominio "a"
474
475 (* Declaracion colateral no exitosa *)
476 - val colat4 = LetExp(AndDecl(val1, val1),
477                           IdExp "a");
478 > val colat4 =
479     LetExp(AndDecl(ValDecl(NoRekursiva, IdPat "a", ConstExp(Entera 1)),
480                     ValDecl(NoRekursiva, IdPat "a", ConstExp(Entera 1))),
481           IdExp "a") : Expression
482 - evalProg colat4;
483 ! Uncaught exception:
484 ! 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(NoRekursiva, IdPat "a", ConstExp(Entera 1)),
493                     ValDecl(NoRekursiva, IdPat "b", ConstExp(Entera 2))),
494           ApExp(IdExp "+", ParExp(IdExp "a", IdExp "b"))) : Expression
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(NoRekursiva, IdPat "a", ConstExp(Entera 1)),
503                     ValDecl(NoRekursiva, IdPat "c",
504                               ApExp(IdExp "+",
505                                     ParExp(IdExp "a", ConstExp(Entera 2))))),
506           IdExp "c") : Expression
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(NoRekursiva, IdPat "a", ConstExp(Entera 1)),
518                     ValDecl(NoRekursiva, IdPat "c",
519                               ApExp(IdExp "+",
520                                     ParExp(IdExp "a", ConstExp(Entera 2))))),
521           IdExp "c") : Expression
522
523 (* Declaracion local no exitosa *)
524 - val loc2 = LetExp(LocalDecl(val1, val4),
525                     IdExp "d");
526 > val loc2 =
527     LetExp(LocalDecl(ValDecl(NoRekursiva, IdPat "a", ConstExp(Entera 1)),
528                     ValDecl(NoRekursiva, IdPat "d",
529                               ApExp(IdExp "+",
530                                     ParExp(IdExp "f", ConstExp(Entera 3))))),
531           IdExp "d") : Expression
532 - evalProg loc2;
533 ! Uncaught exception:
534 ! NoEstaEnElDominio "f"
535
536 (* Declaracion local no exitosa *)

```

```
537 - val loc3 = LetExp(LocalDecl(vall, 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 *)
553 - val fact1 = LetExp(ValDecl(Recursiva, IdPat "fact",
554                             AbsExp[(IdPat "f",
555                                     CondExp([
556                                         (ApExp(IdExp "=", ParExp(IdExp "f", ConstExp(Entera 0))),
557                                           ConstExp(Entera 1)),
558                                         (ApExp(IdExp "=", ParExp(IdExp "f", ConstExp(Entera 1))),
559                                           ConstExp(Entera 1))
560                                     ],
561                                     Something
562                                     (ApExp(IdExp "*", ParExp(IdExp "f", ApExp(IdExp "fact",
563                                         ApExp(IdExp "-", ParExp(
564                                             IdExp "f", ConstExp(
565                                                 Entera 1))))))
566                                     )))
567                               ],
568                               ApExp(IdExp "fact", ConstExp(Entera 5)))
569 > val fact1 =
570     LetExp(ValDecl(Recursiva, IdPat "fact",
571                 AbsExp[(IdPat "f",
572                         CondExp([ (ApExp(IdExp "=",
573                                     ParExp(IdExp "f",
574                                             ConstExp(Entera 0))),
575                                     ConstExp(Entera 1)),
576                                     (ApExp(IdExp "=",
577                                             ParExp(IdExp "f",
578                                                     ConstExp(Entera 1))),
579                                     ConstExp(Entera 1))],
580                         Something(ApExp(IdExp "*",
581                                     ParExp(IdExp "f",
582                                             ApExp(IdExp "fact",
583                                                     ApExp(IdExp "-",
584                                                             ParExp(#,
585                                                                     #)))))))]),
586           ApExp(IdExp "fact", ConstExp(Entera 5))) : Expresion
587 - evalProg fact1;
588 > val it = ConstInt 120 : Valor
589
590 (* Factorial recursivo no exitoso *)
591 - val fact2 = LetExp(ValDecl(NoRecursiva, IdPat "fact",
592                             AbsExp[(IdPat "f",
593                                     CondExp([
594                                         (ApExp(IdExp "=", ParExp(IdExp "f", ConstExp(Entera 0))),
595                                           ConstExp(Entera 1)),
596                                         (ApExp(IdExp "=", ParExp(IdExp "f", ConstExp(Entera 1))),
597                                           ConstExp(Entera 1))
598                                     ],
599                                     Something
600                                     (ApExp(IdExp "*", ParExp(IdExp "f", ApExp(IdExp "fact",
```

```

601                                     ApExp(IdExp "-", ParExp(
602                                     IdExp "f", ConstExp(
603                                     Entera 1)))
604                                     )))
605                                     ))
606                                     ),
607                                     ApExp(IdExp "fact", ConstExp(Entera 5)));
608 > val fact2 =
609     LetExp(ValDecl(NoRecurSiva, IdPat "fact",
610     AbsExp [(IdPat "f",
611     CondExp [(ApExp(IdExp "=",
612     ParExp(IdExp "f",
613     ConstExp(Entera 0))),
614     ConstExp(Entera 1)),
615     (ApExp(IdExp "=",
616     ParExp(IdExp "f",
617     ConstExp(Entera 1))),
618     ConstExp(Entera 1))],
619     Something(ApExp(IdExp "*",
620     ParExp(IdExp "f",
621     ApExp(IdExp "fact",
622     ApExp(IdExp "-",
623     ParExp(
624     #)))))))]),
625     ApExp(IdExp "fact", ConstExp(Entera 5))) : Expression
626 - evalProg fact2;
627 ! Uncaught exception:
628 ! NoEstaEnElDominio "fact"
629 (* Funciones mutuamente recursivas exitosas *)
630 - val mut1 = LetExp(ValDecl(RecurSiva, ParPat(IdPat "f", IdPat "g"),
631     (ParExp(
632     AbsExp[(IdPat "x",
633     ApExp(IdExp "+", ParExp(ConstExp(Entera 1),
634     ApExp(IdExp "g",
635     ApExp(IdExp "-", ParExp(IdExp "x",
636     ConstExp(Entera 1))))
637     )]),
638     AbsExp[(IdPat "y",
639     IfExp( ApExp(IdExp "<", ParExp(IdExp "y",
640     ConstExp(Entera 0))),
641     IdExp "y",
642     ApExp(IdExp "+", ParExp(ConstExp(Entera 1),
643     ApExp(IdExp "f",
644     ApExp(IdExp "-", ParExp(IdExp "y"
645     ,
646     ConstExp(
647     Entera
648     1)
649     )
650     )
651     )
652     )
653     )]),
654     )))
655     ApExp(IdExp "f", ConstExp(Entera 10)));
656 > val mut1 =
657     LetExp(ValDecl(RecurSiva, ParPat(IdPat "f", IdPat "g"),
658     ParExp(AbsExp [(IdPat "x",
659     ApExp(IdExp "+",
660     ParExp(ConstExp(Entera 1),
661     ApExp(IdExp "g",
662     ApExp(IdExp "-",

```

```

663                                     ParExp (IdExp "x",
664                                             ConstExp#)))))],
665
666     AbsExp [(IdPat "y",
667              IfExp (ApExp (IdExp "<",
668                           ParExp (IdExp "y",
669                                   ConstExp (Entera 0))),
670                    IdExp "y",
671                    ApExp (IdExp "+",
672                          ParExp (ConstExp (Entera 1),
673                                ApExp (IdExp "f",
674                                      ApExp (IdExp "-",
675                                              ParExp#)))))])],
676
677     ApExp (IdExp "f", ConstExp (Entera 10))) : Expression
678
679 - evalProg mut1;
680 > val it = ConstInt 10 : Valor
681
682 (* Funciones mutuamente recursivas no exitosas *)
683 - val mut2 = LetExp (ValDecl (NoRecursiva, ParPat (IdPat "f", IdPat "g"),
684              (ParExp (
685                  AbsExp [(IdPat "x",
686                          ApExp (IdExp "+", ParExp (ConstExp (Entera 1),
687                                                    ApExp (IdExp "g",
688                                                          ApExp (IdExp "-", ParExp (IdExp "x",
689                                                                  ConstExp (Entera 1))))
690                          )],
691                  AbsExp [(IdPat "y",
692                          IfExp (ApExp (IdExp "<", ParExp (IdExp "y",
693                                                              ConstExp (Entera 0))),
694                                IdExp "y",
695                                ApExp (IdExp "+", ParExp (ConstExp (Entera 1),
696                                                          ApExp (IdExp "f",
697                                                                ApExp (IdExp "-", ParExp (IdExp "y",
698                                                                    ConstExp (Entera 1))))
699                                )],
700                  )],
701              )],
702              ApExp (IdExp "f", ConstExp (Entera 10)))
703
704 > val mut2 =
705     LetExp (ValDecl (NoRecursiva, ParPat (IdPat "f", IdPat "g"),
706              ParExp (AbsExp [(IdPat "x",
707                              ApExp (IdExp "+",
708                                    ParExp (ConstExp (Entera 1),
709                                          ApExp (IdExp "g",
710                                                  ApExp (IdExp "-",
711                                                          ParExp (IdExp "x",
712                                                                  ConstExp#)))))],
713                              AbsExp [(IdPat "y",
714                                      IfExp (ApExp (IdExp "<",
715                                                      ParExp (IdExp "y",
716                                                              ConstExp (Entera 0))),
717                                            IdExp "y",
718                                            ApExp (IdExp "+",
719                                                  ParExp (ConstExp (Entera 1),
720                                                        ApExp (IdExp "f",
721                                                                ApExp (IdExp "-",
722                                                                    ParExp#)))))])],
723                              ApExp (IdExp "f", ConstExp (Entera 10))) : Expression
724
725 - evalProg mut2;

```



```
727 | ! Uncaught exception:  
728 | ! NoEstaEnElDominio "g"
```

