Un intérprete para SQL

Guido Arnaldo Marzi

1 El intérprete μSQL.

En 1.970, E. F. Codd publicó el primer artículo del llamado "Modelo Relacional de las Bases de Datos"[1], sentando los fundamentos de los modernos sistemas de tratamiento de información (ver Ullman 1989a y Ullman 1989b). Es interesante notar que, a pesar de las dificultades para implementarla de forma eficiente, la teoría subyacente es notablemente simple. Reflejando eso, nuestro intérprete μSQL será lo más sencillo posible, aunque sin sacrificar potencia; claro que lo conseguiremos a costa de algunas cosas: μSQL sólo permitirá consultas. No soportará inserciones ni modificaciones de datos. Las tablas serán inmutables, y deberán ser creadas *a priori* dentro del intérprete; no tendrá nombres de columnas ni mecanismos de alias; tampoco usará índices ni intentará optimizar las consultas.

Si se quiere profundizar los tópicos referentes a bases de datos, se recomiendan (Ullman 1989a) y (Ullman 1989b). Para la construcción de intérpretes, son adecuados (Abelson and Sussman 1996), (Grune et al. 2001) y (Kamin 1990); sobre generadores de *parsers*, (Aho et al. 1986), (Hopcroft and Ullman 1993) y (Mandrioli and Ghezzi 1987); sobre técnicas modernas de compilación, (Appel 1998).

Es interesante notar que el mismo esquema seguido por este intérprete es usado por varios lenguajes, entre ellos *Haskell* (ver (Bird 1998)), *Gofer*, *Clean* y *Python* para generar las, así llamadas, *listas por comprensi'on*[2]. Sobre esto, puede consultarse (Plasmeijer and Eekelen 1993).

1.1 Archivos fuentes.

Para construir μSQL será necesario tener previamente siete archivos:

[1] "A relational model for large shared data banks," *Comm. ACM* **13**:6, pp377–387.
[2] List Comprehension.

Nombre	Página	Propósito
`tab.sml`	7	Tablas que μSQL usará.
`ast.sml`	10	Los tipos abstractos.
`compila.bat`	10	Un *script*, apto para DOS^{TM}, Windows^{TM} y Unix.
`ops.sml`	10	Operaciones.
`sql.sml`	10	Arranque de μSQL.
`lex.lex`	13	Especificación de las expresiones regulares para los *lexemas* usados por μSQL.
`grm.y`	17	Especificación de la gramática libre de contexto para μSQL.

Se recomienda buscar los comienzos de las definiciones (marcados con ✯) en las páginas citadas arriba, e ir armando los archivos fuentes a partir de ellas.

1.2 Implementación de las estructuras para el Algebra Relacional.

Usaremos listas para implementar secuencias y conjuntos. Una *base de datos* es un conjunto de estructuras llamadas *tablas*; en ML

2.1 ⟨*Definición de base de datos* 2.1⟩≡ (10.1)

```
type BaseDeDatos = Tabla list
```

Define:
`BaseDeDatos`, nunca usado.
Usa `Tabla` 2.2.

éstas son conjuntos de *tuplas*,

2.2 ⟨*Definición de* `Tabla` 2.2⟩≡ (10.1)

```
type Tabla = Tupla list
```

Define:
`Tabla`, usado en fragmentos 2.1 y 14.3.
Usa `Tupla` 2.3.

Para honrar la definición anterior debiéramos asegurar que una tabla no tenga elementos repetidos (un conjunto **no** los tiene); pero, para simplificar las cosas, no haremos ninguna verificación.

Una *tupla* es una secuencia de *atributos*,

2.3 ⟨*Definición de* `Tupla` 2.3⟩≡ (10.1)

```
type Tupla = Atrib list
```

Define:
`Tupla`, usado en fragmentos 2.2 y 14.3.
Usa `Atrib` 3.

que son valores atómicos, pertenecientes a un cierto tipo de datos, que condiciona sus posibles valores y operaciones. Por ahora nos contentaremos con tener números enteros y textos.

3 ⟨*Definición de* `Atrib` 3⟩≡ (10.1)

```
datatype Atrib =
    Nro of int
    | Txt of string
```

Define:
`Atrib`, usado en fragmentos 2.3 y 14.3.

El Algebra Relacional sostiene que la respuesta a cualquier consulta a la información contenida en una base de datos puede conseguirse componiendo cinco operaciones básicas: proyección, selección, producto cartesiano[3], unión e intersección. En realidad, la mayoría de las consultas puede hacerse usando sólo las tres primeras, y nos limitaremos a ellas.

De lo anterior se desprende –sin demostración– que toda consulta podrá hacerse efectuando el producto de las tablas involucradas (si hay más de una), filtrando el resultado (si se indica una selección) y proyectando el nuevo resultado (en caso de ser necesario). La relación entre una consulta *SQL* y la cadena de operaciones se indica en la figura 1.

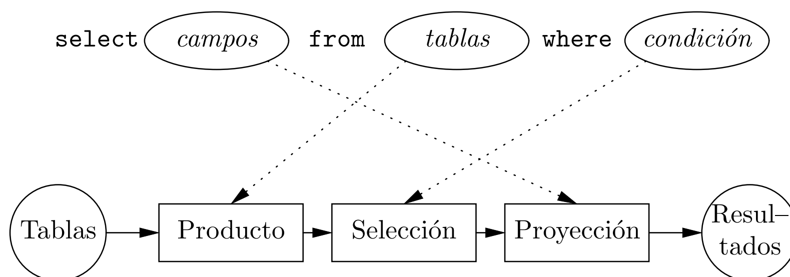

Figura 1: Relación entre consultas y operaciones.

1.3 Simplificaciones para μSQL.

SQL es un lenguaje que permite indicar qué proyecciones, selecciones y productos cartesianos se quieren realizar. Para mantener simple a μSQL nos referiremos a los atributos según el orden que tengan en las tuplas resultantes del producto cartesiano. Así, si tenemos `ph` ⊗ `prefiere` ⊗ `nrostexto`, los atributos de las tuplas resultado serán

$$\overbrace{\#0\#1}^{atribs.\ de\ \texttt{ph}} \quad \overbrace{\#2\#3}^{atribs.\ de\ \texttt{prefiere}} \quad \overbrace{\#4\#5}^{atribs.\ de\ \texttt{nrostexto}} \quad .$$

[3]O producto exterior.

Por supuesto, si hubiéramos tomado `prefiere` ⊗ `nrostexto` ⊗ `ph`, los atributos de las tuplas resultado hubiesen sido

$$\overbrace{\#0\#1}^{atribs.\ de\ \texttt{prefiere}} \quad \overbrace{\#2\#3}^{atribs.\ de\ \texttt{nrostexto}} \quad \overbrace{\#4\#5}^{atribs.\ de\ \texttt{ph}} \ .$$

1.4 Proyección.

Una proyección toma una tabla y una lista de columnas a conservar, y entrega una nueva tabla compuesta sólo por esas columnas.

El módulo `List` de ML exporta la función `nth` que, tomando un entero n y una lista, devuelve el n–ésimo elemnto de esa lista. Con ella es fácil escribir otra función que tomando $[i_1, i_2, \ldots, i_k]$ y una lista, devuelva $[item_{i_1}, item_{i_2}, \ldots, item_{i_k}]$.

4.1 ⟨*Función que saca los* $i_1, \ldots, i_k$ *items* 4.1⟩≡ (4.2)

```
fun aux l = List.map (fn x => List.nth(l, x)) li
```

Define:
`aux`, usado en fragmentos 4.2 y 6.2.

La proyección ahora será:

4.2 ⟨*Definición de proyección* 4.2⟩≡ (10.2)

```
fun proy(li, tabla) =
    let ⟨Función que saca los i1,...,ik items 4.1⟩
    in  map aux tabla end
```

Define:
`proy`, usado en fragmento 15.4.
Usa `aux` 4.1 y `tabla` 9.1.

1.5 Selección.

La selección toma como operandos una tabla y un *predicado*, o sea una función que, aplicada a una tupla, devuelve un `bool`. Devuelve una nueva tabla, compuesta por las tuplas que devuelven `true` de acuerdo al predicado. Por esto, la selección se podrá escribir componiendo `List.filter` con el predicado.

Necesitaremos funciones base para estar en condiciones de armar todos los posibles predicados, o sea, funciones que puedan evaluar condiciones como $\#3 \leq 4$, $\#1 > \#4$, etc.; también debe ser posible componerlas para formar condiciones más complicadas. Vemos que los operandos de estas condiciones son constantes (enteras o de texto) o atributos; las constantes son conocidas de antemano, pero los atributos sólo se podrán evaluar durante la ejecución, luego de componer el predicado. Por esto, definimos una función *"curryed"*

4.3 ⟨*Definición de* `Atr` 4.3⟩≡ (10.2)

```
fun Atr n l = List.nth(l, n)
```

Define:
`Atr`, usado en fragmento 17.2.

que toma la ubicación del atributo en cuestión, espera que le sea dada una tupla (lista), y extrae el atributo requerido. Para respetar el tipado, la función que denote constantes será

5.1 ⟨*Definición de* `Cte` 5.1⟩≡ (10.2)

```
fun Cte c _ = c
```

Define:
`Cte`, usado en fragmento 17.2.

Seguimos con las funciones. Los conectivos *and*, *or* y *not*.

5.2 ⟨*Definición de* `And`, `Or` *y* `Not` 5.2⟩≡ (10.2)

```
fun And p1 p2 x = p1(x) andalso p2(x)
fun Or p1 p2 x = p1(x) orelse p2(x)
fun Not p x = not(p x)
```

Define:
`And`, usado en fragmento 17.1.
`Not`, usado en fragmentos 5.3 y 17.1.
`Or`, usado en fragmentos 5.3 y 17.1.

Funciones que hacen las comparaciones. Si lo tipos no concuerdan es un error.

5.3 ⟨*Definición de las comparaciones* 5.3⟩≡ (10.2)

```
fun Igual m n x =
    case (m x, n x) of
    (Nro i, Nro j) => i=j
    | (Txt i, Txt j) => i=j
    | _ => raise Fail "error: tipos distintos"
fun Menor m n x =
    case (m x, n x) of
    (Nro i, Nro j) => i<j
    | (Txt i, Txt j) => i<j
    | _ => raise Fail "error: tipos distintos"
fun MenIg m n = (Or (Igual m n) (Menor m n))
fun Mayor m n = Not(Or (Igual m n) (Menor m n))
fun MayIg m n = Not(Menor m n)
fun Dist m n = Not(Igual m n)
```

Define:
`Dist`, usado en fragmento 16.5.
`Igual`, usado en fragmento 16.5.
`MayIg`, usado en fragmento 16.5.
`Mayor`, usado en fragmento 16.5.
`MenIg`, usado en fragmento 16.5.
`Menor`, usado en fragmento 16.5.
Usa `Not` 5.2 y `Or` 5.2.

Acá los constructores denotan

Menor: $<$	MenIg: $\leq$
Igual: $=$	Mayor: $>$
MayIg: $\geq$	Dist: $\neq$

Figura 2: Operaciones de condición.

1.6 Producto cartesiano.

El producto cartesiano es una operación muy usada en teoría de conjuntos: construye un conjunto de pares ordenados a partir de dos conjuntos dados, donde el primer elemento de cada par pertenece al primer conjunto, y el segundo elemento al segundo conjunto. Se define así:

$$\{m_1, m_2, \ldots, m_k\} \otimes \{n_1, n_2, \ldots, n_r\} =$$
$$\{(m_1, n_1), (m_1, n_2), \ldots, (m_1, n_r), \ldots, (m_k, n_r)\}$$

Si reemplazamos en esta definición el término 'par ordenado de elementos' por 'secuencia formada por la concatenación de tuplas' tenemos el producto cartesiano que necesitamos.

6.1 ⟨*Producto cartesiano* 6.1⟩≡ (6.2)

```
fun prodCart(x, y) =
        let fun cartB _ [] = []
                | cartB x (h::t) = (x@h)::(cartB x t)
                fun cartA [] _ = []
                | cartA (h::t) l = (cartB h l)@(cartA t l)
        in      cartA x y end
```

Define:
`prodCart`, usado en fragmento 6.2.

Será conveniente, sin embargo, contar con una función que haga el producto de varias tablas, en lugar de sólo dos.

6.2 ⟨*Producto cartesiano de varias tablas* 6.2⟩≡ (10.2)

```
fun prodN l =
    let fun aux [] = raise Fail "producto sin tablas!"
        | aux [x] = x
        | aux(h::t) =
            let ⟨Producto cartesiano 6.1⟩
            in  prodCart(h, aux t) end
    in  aux l end
```

Define:
`prodN`, usado en fragmento 15.
Usa `aux` 4.1 y `prodCart` 6.1.

1.7 Mostrando los resultados.

Mostrar los resultados de las consultas es directo: primero tenemos una función que convierte los atributos de las tuplas a *strings*.

7.1 ⟨*Conversión de atributos a* `strings` 7.1⟩≡ (7.4)
```
fun atrStr(Nro n) = Int.toString(n)
| atrStr(Txt s) = s
```

Define:
`atrStr`, usado en fragmento 7.2.

Luego otra función que aplica `atrStr` a una tabla, convirtiéndola en una lista de listas de `strings`.

7.2 ⟨*Conversión de una tabla en una lista de listas de* `strings` 7.2⟩≡ (7.4)
```
fun mapAtrStr t = map (fn x => map atrStr x) t
```

Define:
`mapAtrStr`, usado en fragmento 7.4.
Usa `atrStr` 7.1.

Por fin, otra función que muestra una lista de `strings` separándolas con espacios y agregando un fin de línea al terminar.

7.3 ⟨*Exhibición de una lista de* `strings` 7.3⟩≡ (7.4)
```
fun printLStr l = (List.app (fn x => print(x^" ")) l; print "\n")
```

Define:
`printLStr`, usado en fragmento 7.4.

7.4 ⟨*Exhibición de resultados* 7.4⟩≡ (10.3)
```
fun printTabla t =
    let ⟨Conversión de atributos a strings 7.1⟩
        ⟨Conversión de una tabla en una lista de listas de strings 7.2⟩
        ⟨Exhibición de una lista de strings 7.3⟩
        val t' = mapAtrStr t
    in  List.app printLStr t' end
```

Define:
`printTabla`, usado en fragmento 9.2.
Usa `mapAtrStr` 7.2 y `printLStr` 7.3.

1.8 Las tablas.

Conforme a lo establecido al principio, las tablas serán listas de tuplas y serán guardadas en `tab.sml`,

☆ 7.5 ⟨*tab.sml* 7.5⟩≡
```
open ast

⟨tablas 8.1⟩
```

Las primeras tablas son adecuadas para probar consultas sobre ascendencia.

8.1 ⟨*tablas* 8.1⟩≡ (7.5) 8.2▷

```
val ph =
    [[Txt "Adan" , Txt "Cain"],
    [Txt "Cain" , Txt "David"],
    [Txt "David", Txt "Salomon"],
    [Txt "Salomon" , Txt "Jose"],
    [Txt "Jose" , Txt "Jesus"]]
```

Define:
ph, usado en fragmento 9.1.

Las que siguen son ejemplos para resolver el problema de los bebedores de cerveza.

8.2 ⟨*tablas* 8.1⟩+≡ (7.5) ◁8.1 8.3▷

```
val prefiere =
    [[Txt "Pepe", Txt "Quilmes"],
    [Txt "Juan", Txt "Palermo"],
    [Txt "Pedro", Txt "Guiness"]]
val concurre =
    [[Txt "Pepe", Txt "Perro's"],
    [Txt "Juan", Txt "Tribunales"],
    [Txt "Pedro", Txt "Coblan"]]
val sirve =
    [[Txt "Coblan", Txt "Guiness"],
    [Txt "Tribunales", Txt "Brama"],
    [Txt "Perro's", Txt "Quilmes"]]
```

Define:
concurre, usado en fragmento 9.1.
prefiere, usado en fragmento 9.1.
sirve, usado en fragmento 9.1.

Las que siguen son para pruebas.

8.3 ⟨*tablas* 8.1⟩+≡ (7.5) ◁8.2 9.1▷

```
val nrostexto =
    [[Nro 1, Txt "uno"],
    [Nro 2, Txt "dos"],
    [Nro 3, Txt "tres"],
    [Nro 4, Txt "cuatro"],
    [Nro 5, Txt "cinco"],
    [Nro 6, Txt "seis"],
    [Nro 7, Txt "siete"]]
```

Define:
nrostexto, usado en fragmento 9.1.

Necesitamos ponerle nombre a las tablas y una función que devuelva esta ligazón. Nada mejor que usar la función `find` del módulo `List`. Si la tabla

buscada no existe, levantamos la excepción `Fail`.

9.1 ⟨*tablas* 8.1⟩+≡ (7.5) ◁8.3

```
      val nombreTablas = [
          ("prefiere", prefiere),
          ("concurre", concurre),
          ("sirve", sirve),
          ("nrostexto", nrostexto),
          ("ph", ph)
      ]
  fun tabla s =
      let val tabla =
                 case  List.find (fn(x, _) => x=s) nombreTablas of
                 SOME(_, tabla) => tabla
                 | NONE => raise Fail "tab! la tabla no existe!"
      in  tabla end
```

Define:
`tabla`, usado en fragmentos 4.2 y 16.3.
Usa `concurre` 8.2, `nrostexto` 8.3, `ph` 8.1, `prefiere` 8.2, y `sirve` 8.2.

1.9 Arranque de μSQL.

Permitiremos que las consultas puedan ser escritas previamente en un archivo y procesadas invocando a μSQL seguido por el nombre de dicho archivo: `musql` *archivo*; si se omite el archivo, μSQL tomará la consulta desde el teclado (en la jerga de UNIX se conoce como entrada estándar o *stdin*; en ML `std_in`).

9.2 ⟨*Arranque del intérprete* 9.2⟩≡ (10.3)

```
  fun createLexerStream (is : instream) =
      Lexing.createLexer (fn buff =>
          fn n => Nonstdio.buff_input is buff 0 n)

  val _ =
      let val entr = open_in(hd(CommandLine.arguments()))
                  handle _ => (print "usando std_in\n"; std_in)
          val lexbuf = createLexerStream entr
          val result = grm.query lex.Tok lexbuf
      in
          print "Respuesta:\n";
          printTabla result
      end
      handle Fail s => print(s^"\n")
      | _ => print "Aughh!\n"
```

Define:
`createLexerStream`, nunca usado.
Usa `printTabla` 7.4, `query` 15.4, y `Tok` 13.7.

1.10 Juntando las piezas.

Las definiciones deben ir al archivo `ast.sml` para evitar dependencias circulares.

10.1 ⟨*ast.sml* 10.1⟩≡
  ⟨*Definición de* `Atrib` 3⟩
  ⟨*Definición de* `Tupla` 2.3⟩
  ⟨*Definición de* `Tabla` 2.2⟩
  ⟨*Definición de base de datos* 2.1⟩

Idem para las operaciones de condición y de conjunto, en el archivo `ops.sml`.

10.2 ⟨*ops.sml* 10.2⟩≡

```
open ast
```

  ⟨*Definición de* `Atr` 4.3⟩
  ⟨*Definición de* `Cte` 5.1⟩
  ⟨*Definición de* `And`*,* `Or` *y* `Not` 5.2⟩
  ⟨*Definición de las comparaciones* 5.3⟩

  ⟨*Definición de proyección* 4.2⟩
  ⟨*Producto cartesiano de varias tablas* 6.2⟩

Juntamos lo definido hasta ahora en el archivo `sql.sml`.

10.3 ⟨*sql.sml* 10.3⟩≡

```
(*
    Interprete para SQL.
*)

open BasicIO Nonstdio
open ast
open ops
```

  ⟨*Exhibición de resultados* 7.4⟩
  ⟨*Arranque del intérprete* 9.2⟩

Para tener a μSQL operativo, deberemos compilar y unir las partes. Como muchos de Uds. no usan UNIX (una lástima), trataremos de hacer algo neutral, como un archivo `.BAT`. Acá va.

10.4 ⟨*compila.bat* 10.4⟩≡

```
mosmllex lex.lex
mosmlyac -v grm.y
mosmlc -c -liberal ast.sml
mosmlc -c -liberal ops.sml
mosmlc -c -liberal grm.sig
mosmlc -c -liberal lex.sml
mosmlc -c -liberal sql.sml
```

```
mosmlc -c -liberal tab.sml
mosmlc -c -liberal grm.sml
mosmlc -o musql sql.uo
```

1.11 El *scanner*.

El *scanner* para μSQL es un módulo que debe leer, caracter a caracter, y convertirlos en símbolos o ignorarlos. Qué hacer y cuándo hacerlo, es lo que debemos codificar en lo que sigue.

Ignoraremos los espacios, las tabulaciones y los cambios de línea

11.1 ⟨*Acciones del scanner* 11.1⟩≡ (13.7) 11.2▷

```
[‘ ‘‘\t‘‘\n‘]+      { Tok lexbuf }
```

Usa `Tok` 13.7.

La regla anterior establece que, si aparece un espacio, se debe reinvocar el *scanner* olvidando lo que produjo que esta regla se use.

El texto $\#n$ se usa para nombrar al n–ésimo atributo, de una tupla, por lo que deberemos reconocer $\#$.

11.2 ⟨*Acciones del scanner* 11.1⟩+≡ (13.7) ◁11.1 11.3▷

```
| "#"              { CARD }
```

Usa `CARD` 14.1.

Siguen las secuencias que denotan operaciones o aspectos especiales.

11.3 ⟨*Acciones del scanner* 11.1⟩+≡ (13.7) ◁11.2 11.4▷

```
| "<>"             { DIST }
| "="              { IGUAL }
| ">="             { MAYIG }
| ">"              { MAYOR }
| "<="             { MENIG }
| "<"              { MENOR }
```

Usa `DIST` 14.1, `IGUAL` 14.1, `MAYIG` 14.1, `MAYOR` 14.1, `MENIG` 14.1, y `MENOR` 14.1.

Un asterisco entre `select` y `from` indica ausencia de proyección.

11.4 ⟨*Acciones del scanner* 11.1⟩+≡ (13.7) ◁11.3 11.5▷

```
| "*"              { ASTER }
```

Usa `ASTER` 14.1.

Necesitaremos otros caracteres para separar y agrupar; generalmente se usan para esto los paréntesis, la coma y el punto y coma.

11.5 ⟨*Acciones del scanner* 11.1⟩+≡ (13.7) ◁11.4 12.4▷

```
| ","              { COMA }
| ";"              { PCOMA }
| ")"              { PD }
| "("              { PI }
```

Usa `COMA` 14.1, `PCOMA` 14.1, `PD` 14.1, y `PI` 14.1.

También debemos reconocer palabras claves. Podemos reducir significativamente el *scanner* con el siguiente truco: una tabla *hash* que determina si una palabra es in identificador o una palabra clave. Construímos la tabla así.

12.1 ⟨*Tabla hash para palabras clave* 12.1⟩≡ (13.7)

```
exception noEsta
val palClaves = Polyhash.mkPolyTable(19, noEsta)
```

Debemos llenar la tabla con las palabras claves y los lexemas que denotan.

12.2 ⟨*Llenado de la tabla hash* 12.2⟩≡ (13.7)

```
val _ = List.app (fn x => Polyhash.insert palClaves x)
            [⟨Lista de palabras claves/lexemas 12.3⟩]
```

Ahora damos los pares palabras claves/lexemas.

12.3 ⟨*Lista de palabras claves/lexemas* 12.3⟩≡ (12.2)

```
("and",          AND),
("from",         FROM),
("not",          NOT),
("or",           OR),
("select",       SELECT),
("where",        WHERE)
```

Usa `AND` 14.1, `FROM` 14.1, `NOT` 14.1, `OR` 14.1, `SELECT` 14.1, y `WHERE` 14.1.

Los números son secuencias de dígitos. Nótese cómo evitamos números "anormales" como 0001, etc.

12.4 ⟨*Acciones del scanner* 11.1⟩+≡ (13.7) ◁11.5 12.5▷

```
| `0`|[`1`-`9`][`0`-`9`]*      { ENT(atoi(getLexeme lexbuf)) }
```

Usa `ENT` 14.2.

Las palabras claves y los nombres de las tablas que intervienen en la consulta son secuencias que empiezan con una letra, seguidas de cero o más letras y/o dígitos.

12.5 ⟨*Acciones del scanner* 11.1⟩+≡ (13.7) ◁12.4 13.1▷

```
| [`A`-`Z``a`-`z`][`A`-`Z``a`-`z``_``0`-`9`]+
                          { idOPalClave(getLexeme lexbuf) }
```

`IdOPalClave` es una función que decide si estamos enpresencia de un identificador o una palabra clave.

12.6 ⟨*Función* `idOPalClave` 12.6⟩≡ (13.7)

```
fun idOPalClave s =
    case Polyhash.peek palClaves s of
    SOME v => v
    | NONE => ID s
```

Usa `ID` 14.2.

La función `atoi` que aparece arriba se define como sigue.

12.7 ⟨*Funciones para el scanner* 12.7⟩≡ (13.7)

```
val atoi = valOf o Int.fromString
```

Queda lo más complicado: las constantes textuales o `strings`. Estas constantes se forman con texto delimitado con comillas. Para permitir que entre los caracteres de este texto se puedan encontrar comillas literales y otros caracteres especiales (como cambios de línea), cuando estos caracteres deban figurar literalmente, deberán estar codificados como `\"` (comillas) o `\n` (cambio de línea).

13.1 ⟨*Acciones del scanner* 11.1⟩+≡ (13.7) ◁12.5 13.2▷

```
  | ‘"‘                { TEXTO(String lexbuf) }
```

Usa `String` 13.7 y `TEXTO` 14.2.

Finalmente, si aparece otro caracter, es un error.

13.2 ⟨*Acciones del scanner* 11.1⟩+≡ (13.7) ◁13.1

```
  | _                  { raise Fail("caracter ["^
                                getLexeme(lexbuf)^"] raro!") }
```

Para procesar correctamente las constantes textuales, activamos otro *scanner*, en este caso `String`.

Debe indicar un error si la entrada termina antes de la comilla que cierre la constante, o sea, si aparece `eof`. Idem si se llega al fin de la línea sin haber terminado.

13.3 ⟨*Scanner para* `String` 13.3⟩≡ (13.7) 13.4▷

```
  eof                  { raise Fail "EOF en string!" }
  | ‘\n‘               { raise Fail "string incompleta!" }
```

Una barra seguida por una comilla (`\"`) es una comilla (”) literal. Una comilla (`"`) indica el fin de la cadena.

13.4 ⟨*Scanner para* `String` 13.3⟩+≡ (13.7) ◁13.3 13.5▷

```
  | "\\\""             { "\""^String(lexbuf) }
  | ‘"‘                { "" }
```

Usa `String` 13.7.

`String` debe colectar todo otro caracter legítimo que aparezca,

13.5 ⟨*Scanner para* `String` 13.3⟩+≡ (13.7) ◁13.4 13.6▷

```
  | [‘ ‘-‘z‘]          { getLexeme(lexbuf)^String(lexbuf) }
```

Usa `String` 13.7.

Finalmente, otro caracter, aparte de los anteriores, es un error:

13.6 ⟨*Scanner para* `String` 13.3⟩+≡ (13.7) ◁13.5

```
  | _                  { raise Fail "caracter desconocido!" }
```

Juntamos todo para formar el *scanner*, que quedará en el archivo `lex.lex`.

13.7 ⟨*lex.lex* 13.7⟩≡

```
  {
  open grm
  ⟨Tabla hash para palabras clave 12.1⟩
  ⟨Llenado de la tabla hash 12.2⟩
  ⟨Función idOPalClave 12.6⟩
  ⟨Funciones para el scanner 12.7⟩
  }

  rule Tok =
```

```
    parse
    ⟨Acciones del scanner 11.1⟩
and String =
    parse
    ⟨Scanner para String 13.3⟩
    ;
```

Define:
`String`, usado en fragmento 13.
`Tok`, usado en fragmentos 9.2 y 11.1.

1.12 El *parser*.

Declaramos como *tokens* o símbolos a las palabras clave y signos de puntuación.

14.1 ⟨*Declaración de tokens* 14.1⟩≡ (17.3) 14.2 ▷

```
%token CARD COMA PCOMA PI PD ASTER
%token FROM SELECT WHERE
%token AND NOT OR
%token MENOR MENIG IGUAL MAYIG MAYOR DIST
```

Define:
`AND`, usado en fragmentos 12.3, 15.1, y 17.1.
`ASTER`, usado en fragmentos 11.4 y 15.5.
`CARD`, usado en fragmentos 11.2 y 16.2.
`COMA`, usado en fragmentos 11.5 y 16.
`DIST`, usado en fragmentos 11.3, 15.2, y 16.5.
`FROM`, usado en fragmentos 12.3 y 15.
`IGUAL`, usado en fragmentos 11.3, 15.2, y 16.5.
`MAYIG`, usado en fragmentos 11.3, 15.2, y 16.5.
`MAYOR`, usado en fragmentos 11.3, 15.2, y 16.5.
`MENIG`, usado en fragmentos 11.3, 15.2, y 16.5.
`MENOR`, usado en fragmentos 11.3, 15.2, y 16.5.
`NOT`, usado en fragmentos 12.3, 15.1, y 17.1.
`OR`, usado en fragmentos 12.3, 15.1, y 17.1.
`PCOMA`, usado en fragmentos 11.5 y 15.
`PD`, usado en fragmentos 11.5 y 17.1.
`PI`, usado en fragmentos 11.5 y 17.1.
`SELECT`, usado en fragmentos 12.3 y 15.
`WHERE`, usado en fragmentos 12.3 y 16.4.

Siguen `ENT`, `ID`, y `TEXTO`, que deben portar atributos.

14.2 ⟨*Declaración de tokens* 14.1⟩+≡ (17.3) ◁ 14.1

```
%token<int> ENT
%token<string> ID TEXTO
```

Define:
`ENT`, usado en fragmentos 12.4, 16.2, y 17.2.
`ID`, usado en fragmentos 12.6 y 16.3.
`TEXTO`, usado en fragmentos 13.1 y 17.2.

Los no–terminales, con sus tipos.

14.3 ⟨*Declaración de no–terminales* 14.3⟩≡ (17.3)

```
%type<ast.Tabla> query
%type<int> atr
```

```
%type<int list> listaAtrs
%type<ast.Tabla list> listaTablas
%type<ast.Tabla -> ast.Tabla> seccWhere
%type<ast.Tupla -> bool> cond
%type<ast.Tupla -> ast.Atrib> expr
```

Usa `atr` 16.2, `Atrib` 3, `cond` 16.5, `expr` 17.2, `listaAtrs` 16.1, `listaTablas` 16.3, `query` 15.4, `seccWhere` 16.4, `Tabla` 2.2, y `Tupla` 2.3.

Las precedencias y asociatividades de los operadores. Las precedencias más bajas corresponden a `or`, `and` y `not`, en ese orden. Los dos primeros asocian a izquierda; el último a derecha.

15.1 ⟨*Declaración de precedencias* 15.1⟩≡ (17.3) 15.2 ▷

```
%left OR
%left AND
%right NOT
```

Usa `AND` 14.1, `NOT` 14.1, y `OR` 14.1.

Siguen los operadores de comparación. Asignamos a todos la misma precedencia. Por supuesto, no son asociativos.

15.2 ⟨*Declaración de precedencias* 15.1⟩+≡ (17.3) ◁ 15.1

```
%nonassoc MENOR MENIG IGUAL MAYIG MAYOR DIST
```

Usa `DIST` 14.1, `IGUAL` 14.1, `MAYIG` 14.1, `MAYOR` 14.1, `MENIG` 14.1, y `MENOR` 14.1.

Tenemos un solo no–terminal de arranque: `query`.

15.3 ⟨*No–terminal de arranque* 15.3⟩≡ (17.3)

```
%start query
```

Usa `query` 15.4.

Empecemos con las reglas de producción. Primeramente la que corresponde a `query` (fig. 3).

15.4 ⟨*Producción para* `query` 15.4⟩≡ (17.3) 15.5 ▷

```
query: SELECT listaAtrs FROM listaTablas seccWhere PCOMA
                { proy($2, $5(prodN $4)) }
```

Define:
`query`, usado en fragmentos 9.2, 14.3, y 15.3.
Usa `FROM` 14.1, `listaAtrs` 16.1, `listaTablas` 16.3, `PCOMA` 14.1, `prodN` 6.2, `proy` 4.2, `seccWhere` 16.4, y `SELECT` 14.1.

Un `query` puede no necesitar proyección. Un asterisco señala esto.

15.5 ⟨*Producción para* `query` 15.4⟩+≡ (17.3) ◁ 15.4

```
    | SELECT ASTER FROM listaTablas seccWhere PCOMA
                { $5(prodN $4) }
```

Usa `ASTER` 14.1, `FROM` 14.1, `listaTablas` 16.3, `PCOMA` 14.1, `prodN` 6.2, `seccWhere` 16.4, y `SELECT` 14.1.

Una lista de atributos necesita, al menos, un elemento.

16.1 ⟨*Producción para* `listaAtrs` 16.1⟩≡ (17.3)
```
listaAtrs: atr          { [$1] }
    | atr COMA listaAtrs    { $1::$3 }
```

Define:
`listaAtrs`, usado en fragmentos 14.3 y 15.4.
Usa `atr` 16.2 y `COMA` 14.1.

Un atributo se debe escribir $\#n$, con $n \in N^{+}$.

16.2 ⟨*Producción para* `atr` 16.2⟩≡ (17.3)
```
atr: CARD ENT           { $2 }
```

Define:
`atr`, usado en fragmentos 14.3, 16.1, y 17.2.
Usa `CARD` 14.1 y `ENT` 14.2.

Recordemos que este valor será usado posteriormente por `List.nth`, que numera de cero en adelante a los elementos de una lista.

Una tabla se denota con su nombre. La lista de tablas requiere, al menos, un elemento, y es armada por la próxima regla.

16.3 ⟨*Producción para* `listaTablas` 16.3⟩≡ (17.3)
```
listaTablas: ID         { [tabla $1] }
    | ID COMA listaTablas   { tabla($1)::$3 }
```

Define:
`listaTablas`, usado en fragmentos 14 y 15.
Usa `COMA` 14.1, `ID` 14.2, y `tabla` 9.1.

En cambio, la condición es opcional. Si está presente, componemos `List.filter` con el predicado; si no lo está, generamos una función anónima que devuelve lo que recibe.

16.4 ⟨*Producción para* `seccWhere` 16.4⟩≡ (17.3)
```
seccWhere: WHERE cond       { List.filter $2 }
    |           { fn x => x }
```

Define:
`seccWhere`, usado en fragmentos 14 y 15.
Usa `cond` 16.5 y `WHERE` 14.1.

La sintaxis de una condición es (fig. 4)

16.5 ⟨*Producciones para* `cond` 16.5⟩≡ (17.3) 17.1 ▷
```
cond: expr MENOR expr       { Menor $1 $3 }
    | expr MENIG expr   { MenIg $1 $3 }
    | expr IGUAL expr   { Igual $1 $3 }
    | expr MAYIG expr   { MayIg $1 $3 }
    | expr MAYOR expr   { Mayor $1 $3 }
    | expr DIST expr    { Dist $1 $3 }
```

Define:
`cond`, usado en fragmentos 14.3, 16.4, y 17.1.

Usa `DIST` 14.1, `Dist` 5.3, `expr` 17.2, `IGUAL` 14.1, `Igual` 5.3, `MAYIG` 14.1, `MayIg` 5.3, `MAYOR` 14.1, `Mayor` 5.3, `MENIG` 14.1, `MenIg` 5.3, `MENOR` 14.1, y `Menor` 5.3.

Las producciones que siguen son obvias: los paréntesis sirven para alterar las precedencias.

17.1 ⟨*Producciones para* `cond` 16.5⟩+≡ (17.3) ◁16.5

```
      | PI cond PD        { $2 }
      | cond AND cond     { And $1 $3 }
      | cond OR cond      { Or $1 $3 }
      | NOT cond      { Not $2 }
```

Usa `AND` 14.1, `And` 5.2, `cond` 16.5, `NOT` 14.1, `Not` 5.2, `OR` 14.1, `Or` 5.2, `PD` 14.1, y `PI` 14.1.

Una expresión para un condicional puede ser (fig. 4)

17.2 ⟨*Producciones para* `expr` 17.2⟩≡ (17.3)

```
  expr: atr           { Atr $1 }
      | ENT           { Cte(Nro $1) }
      | TEXTO         { Cte(Txt $1) }
```

Define:
`expr`, usado en fragmentos 14.3 y 16.5.
Usa `Atr` 4.3, `atr` 16.2, `Cte` 5.1, `ENT` 14.2, y `TEXTO` 14.2.

Armamos las especificaciones del *parser* en `grm.y`.

✯ 17.3 ⟨*grm.y* 17.3⟩≡

```
  %{
  open ast
  open tab
  open ops
  %}
  ⟨Declaración de tokens 14.1⟩
  ⟨Declaración de no–terminales 14.3⟩
  ⟨Declaración de precedencias 15.1⟩
  ⟨No–terminal de arranque 15.3⟩
  %%
  ⟨Producción para query 15.4⟩
  ⟨Producción para listaAtrs 16.1⟩
  ⟨Producción para atr 16.2⟩
  ⟨Producción para listaTablas 16.3⟩
  ⟨Producción para seccWhere 16.4⟩
  ⟨Producciones para cond 16.5⟩
  ⟨Producciones para expr 17.2⟩
  %%
```

1.13 Ejemplos.

Ejemplo 1:

```
select #0, #3 from ph, ph where #1 = #2;

Respuesta:
Adan David
Cain Salomon
David Jose
Salomon Jesus
```

Ejemplo 2:

```
select #0, #5 from ph, ph, ph
where #1 = #2 and #3 = #4;

Respuesta:
Adan Salomon
Cain Jose
David Jesus
```

Ejemplo 3:

```
select #0, #1 from prefiere, concurre, sirve
where #0 = #2 and #3 = #4 and #1 = #5;

Respuesta:
Pepe Quilmes
Pedro Guiness
```

Ejemplo 4:

```
select * from ph, ph, ph
where #1 = #2 and #3 = #4;

Respuesta:
Adan Cain Cain David David Salomon
Cain David David Salomon Salomon Jose
David Salomon Salomon Jose Jose Jesus
```

Ejemplo 5:

```
select #0, #5 from ph, ph, ph
where #1 = #2 and #3 = #4 and #0 = "Adan";

Respuesta:
Adan Salomon
```

Ejemplo 6:

```
select #0, #5 from ph, ph, ph
where #1 = #2 and #3 = #4 and #0 <> "Adan";

Respuesta:
Cain Jose
David Jesus
```

Ejemplo 7:

```
select #0, #5 from ph, ph, ph
where #1 = #2 and #3 = #4 and not #0 <> "Adan";

Respuesta:
Adan Salomon
```

1.14 Extensiones propuestas.

El intérprete puede extenderse de varias maneras. Enumeremos algunas de ellas.

- Extienda las condiciones de la selección, de tal modo que acepte relaciones con operaciones como suma, diferencia, producto, etc.
  Un ejemplo puede ser `select #1 from cuentas where #2+#3>#4;`.
- Extienda los tipos de atributos con `null`, que indica que ese atributo no tiene un valor preciso. Las comparaciones con `null` siempre dan como resultado `false`, excepto para dos predicados nuevos: `isnull` e `isnotnull`.
- Agregue la posibilidad de ordenar la salida según cualquier atributo que aparezca en el producto cartesiano. Esta ordenación podrá ser ascendente o descendente. Su sintaxis debe ser

  `select #1 from ph order by #2;`.

  `select #1 from ph wher #2=#3 order by #2 desc;`.

  Por simplicidad, considere ordenamientos según un solo atributo.
- Extienda el ítem anterior para ordenar según varios atributos.
- Extienda los atributos con el tipo *fecha*. Cuide que las comparaciones respeten las convenciones para años bisiestos, etc.
- Extienda los predicados para la selección con manipuladores de textos como `substring`, `toupper` y `tolower`. El primero extrae una porción de un texto, la segunda convvirte el texto a mayúsculas, y el tercero a minúsculas. Posibles ejemplos del uso de éstos pueden ser

  `select #1 from ph where substring(#1, 1, 2) = "es";`.

  `select #1 from ph where tolower(#1) = "jesus";`.

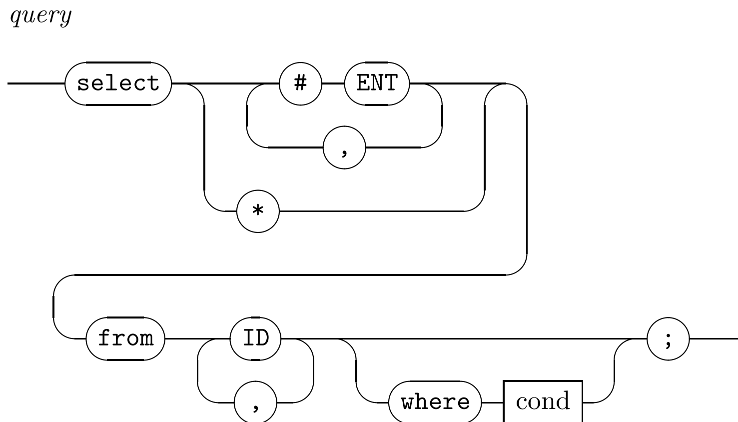

Figura 3: `lquery`

- Agregue el predicado `between`, equivalente a dos condiciones. Ej.: `#1 between(3, 5)` equivale a `#1>=3 and #1<=5`.

1.15 Diagramas ferroviarios de la gramática

Los diagramas ferroviarios, inventados por Conway, sirven para exhibir la estructura sintáctica de un lenguaje. Se deben leer como un juego de laberintos, donde hay ciertos caminos que nos llevan de la izquierda a la derecha; los nodos que debamos cruzar en el camino deberán estar presentes (y aparecer en el mismo orden) en una frase o sentencia concreta del lenguaje en cuestión.

A Breve Introducción a *ML*.

Describiremos suscintamente al lenguaje *ML*, en particular la versión de *MosML*. Para una exposición más detallada se puede consultar (Wikström 1987) y (Ullman 1998). Para la definición formal de *ML*, ver (Milner et al. 1997). Un libro sobresaliente para aprender a usarlo es (Paulson 1988).

A.1 Los primeros pasos.

MosML trae un intérprete, llamado `mosml` y un compilador, `mosmlc`. Para tomar práctica con el lenguaje, es más apropiado el intérprete, y se arranca con

```
mosml
```

A partir de este momento, lo que escribamos estará a continuación del prompt `-` hasta el `;`. La respuesta del intérprete comenzará con `>`.

Luego de ser invocado, el intérprete muestra un *banner* e indica que está listo para recibir órdenes con "-".

cond

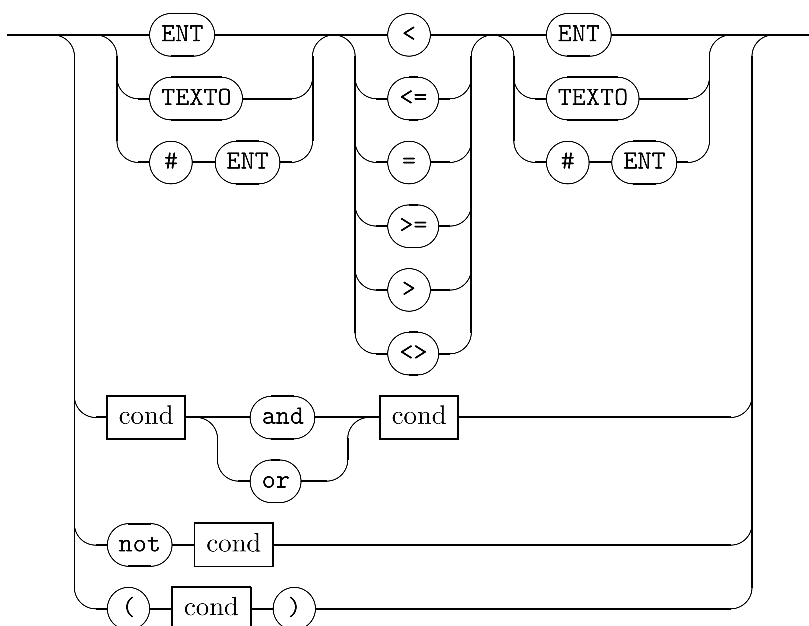

Figura 4: `cond`

```
Moscow ML version 2.00 (June 2000)
Enter 'quit();' to quit.
-
```

Para abandonar el intérprete debemos escribir `quit();`. El punto y coma indica al intérprete que la orden está completa.
- *quit()'*

MosML evaluará todas las expresiones que tipeemos, indicando el resultado, o mostrando el error. Por otra parte, cualquier cosa que esté entre `(*` y `*)` es ignorado, y tomado como un comentario; los comentarios pueden anidarse.

```
- 2+3;
> val it = 5 : int
- 2+true;
! Toplevel input:
! 2+true
!   ^^^^
! Type clash: expression of type
!   bool
! cannot have type
!   int
```

A.2 Valores y tipos primitivos.

ML tiene un rico sistema de tipos primitivos. Estos son los que siguen .

Nombre	Descripción	Ejemplos
`unit`	Valor que denota *vacío*	`()`
`bool`	Valores booleanos	`true`, `false`
`int`	Números enteros	`1`, `15`, `~7`[4]
`char`	Caracteres	`#"A"`, `#"\n"`, `#"^C"`
`word`	Enteros sin signo	`0wx1`, `0wxaa`, `0wxFFFF`
`word8`	*Bytes*	`0wx1`, `0wxaa`, `0wxFF`
`string`	Cadenas alfanuméricas	`"Hola"`, `"ho\  \la"`[5], `""`
`real`	Números reales	`3.0`, `~2.78`, `1e~3`

De los ejemplos de la subsección anterior, vemos que, a pesar de ser un lenguaje fuertemente tipado, no necesitamos declarar explícitamente los tipos: el intérprete los deduce.

A.2.1 *Strings* constantes.

Las *strings* constantes se denotan con cadenas de caracteres entre comillas, como `"hola mundo"`. Al igual que *C* y otros lenguajes, *ML*permite incluír caracteres de control, usando una codificación similar. Esta es la que sigue.

`\a`	*Bell* (*ASCII 7*).
`\b`	*Backspace* (*ASCII 8*).
`\t`	*Horizontal tab* (*ASCII 9*).
`\n`	*Line–feed* (*ASCII 10*).
`\v`	*Vertical tab* (*ASCII 11*).
`\f`	*Form–feed* (*ASCII 12*).
`\r`	*Return* (*ASCII 13*).
`\^`*c*	El caracter correspondiente al *ASCII* igual a $c - 64$.
`\`*ddd*	El caracter correspondiente al nro. *ddd* (de tres dígitos decimales).
`\u`*hhhh*	Idem, pero con el número con cuatro dígitos hexadecimales.

A.3 Operaciones sobre tipos primitivos.

Dividiremos las operaciones según qué tipos tienen sus operandos y qué tipo el resultado.

A.3.1 Operaciones `int -> int`.

Nombre	Descripción
`~`	Negación unaria
`abs`	Valor absoluto

A.3.2 Operaciones `int * int -> int`.

Nombre	Descripción		Nombre	Descripción
`+`	Suma		`-`	Diferencia
`*`	Producto		`div`	Cociente
`mod`	Módulo			

A.3.3 Operaciones `int * int -> bool`.

Nombre	Descripción		Nombre	Descripción
`<`	Menor		`<=`	Menor o igual
`>`	Mayor		`>=`	Mayor o igual
`=`	Igual		`<>`	Distinto

A.3.4 Operaciones `real -> real`.

Nombre	Descripción
`~`	Negación unaria
`abs`	Valor absoluto

A.3.5 Operaciones `real * real -> real`.

Nombre	Descripción		Nombre	Descripción
`+`	Suma		`-`	Diferencia
`*`	Producto		`/`	Cociente

A.3.6 Operaciones `real * real -> bool`.

Nombre	Descripción		Nombre	Descripción
`<`	Menor		`<=`	Menor o igual
`>`	Mayor		`>=`	Mayor o igual
`=`	Igual[6]		`<>`	Distinto

A.3.7 Operaciones `real -> int`.

Nombre	Descripción		Nombre	Descripción
`floor`	Piso		`ceil`	Superior
`round`	Redondeo		`trunc`	Truncamiento

El resultado de estas operaciones se muestra en la tabla que sigue.

Operación	1.1	1.9	-1.1	-1.9
`floor`	1	1	-2	-2
`ceil`	2	2	-1	-1
`round`	1	2	-1	-2
`trunc`	1	1	-1	-1

A.3.8 Operaciones `int -> real`.

Nombre	Descripción
`real`	Convierte un entero a real

A.3.9 Operaciones `int -> char`.

Nombre	Descripción
`chr`	Convierte un entero a caracter

A.3.10 Operaciones `char * char -> bool`.

Nombre	Descripción		Nombre	Descripción
`<`	Menor		`<=`	Menor o igual
`>`	Mayor		`>=`	Mayor o igual
`=`	Igual		`<>`	Distinto

Las comparaciones se hacen según el código *ASCII*.

A.3.11 Operaciones `char -> int`.

Nombre	Descripción
`ord`	Convierte un caracter a entero

A.3.12 Operaciones `char -> string`.

Nombre	Descripción
`str`	Convierte un caracter a *string*

A.3.13 Operaciones `string * string -> string`.

Nombre	Descripción
`^`	Concatenación

Las comparaciones se hacen según el orden lexicográfico (o de diccionario), siguiendo el código *ASCII*. La *string* nula es el menor valor.

A.3.14 Operaciones `string * string -> bool`.

Nombre	Descripción		Nombre	Descripción
`<`	Menor		`<=`	Menor o igual
`>`	Mayor		`>=`	Mayor o igual
`=`	Igual		`<>`	Distinto

Las comparaciones se hacen según el orden lexicográfico (o de diccionario), siguiendo el código *ASCII*. La *string* nula es el menor valor.

A.3.15 Operaciones `bool * bool -> bool`.

Nombre	Uso	Descripción
`if`	`if e then v else f`	Si `e` es cierta, evalúa `v`; caso contrario `f`[7]
`not`	`not a`	`if a then false else true`
`andalso`	`a andalso b`	`if a then b else false`
`orelse`	`a orelse b`	`if a then true else b`

A.3.16 Ejemplos.

Algunos ejemplos (correctos) de operaciones:

```
3+5*8
(3+5)*8
```

```
"hola">"Hola" orelse 4+2>5
"hola"^(if 3>2 then "que tal" else "chau")
```

A.4 Secuencias.

ML permite hacer secuencias de expresiones con `;`.
- *(print "hola "; print "chau
n"; 2+3);*

```
hola chau
> val it = 5 : int
```

El resultado de la expresión anterior es el resultado de `2+3`. También se puede indicar la evaluación de una expresión antes de devolver otra.
- *(2+3) before print "hum...
n";*

```
hum...
> val it = 5 : int
```

A.5 *Bindings*.

ML permite nombrar (poner nombre) a cualquier valor. Se hace como sigue:

`val` *nombre* = *expresión.*

Luego de esto, se dice que *nombre* es una ligadura o *binding* para el valor de *expresión.* Si usamos el mismo nombre para dos *bindings*, el último oculta al primero.

Ejemplos.
- *val x = 2+3;*
`> val it = 5 : int`
- *x;*
`> val x = 5 : int`
- *val h = "hola"^ str(chr 66);*
`> val it = "holaB" : string`
- *h;*
`> val h = "holaB" : string`

A.6 Tuplas, *records* y listas.

ML, además de los tipos primitivos, permite definir nuevos tipos, usando los ya existentes. Uno de éstos son las *tuplas.* Una tupla es un agregado de valores de otros tipos. Como ejemplo,
- *(true, "hola");*
`> val it = (true, "hola") : bool * string`

ML nos indica que *(true, "hola")* es una tupla, formada por un booleano y una *string.*

Otros ejemplos son

- *(1, ”hola”, false);*
> val it = (1, "hola", false) : int * string * bool
- *(1, (”hola”, false));*
> val it = (1, ("hola", false)) : int * (string * bool)

La primera es una tupla de **tres** elementos (un entero, una “/ y un booleano), en tanto que la segunda tiene **dos** elementos (un entero y una tupla).

A una tupla se le pueden extraer elementos.

- *val t = (1, (”hola”, false));*
> val t = (1, ("hola", false)) : int * (string * bool)
- *#1(t);*
> val it = 1 : int
- *#2(t);*
> val it = ("hola", false) : string * bool
- *#2(#2(t));*
> val it = false : bool

Un *record* es una tupla, donde sus elementos tienen nombre; este nombre sirve para extraerlos. Son similares a las estructuras de *C* (ver Kernighan and Ritchie 1988). Ejemplos.

- *val r = num=17, cartel=”hola”, tupla=(3.14, 2.78);*
> val r = {cartel = "hola", num = 17, tupla = (3.14, 2.78)} :
  {cartel : string, num : int, tupla : real * real}
- *r;*
> val it = {cartel = "hola", num = 17, tupla = (3.14, 2.78)} :
  {cartel : string, num : int, tupla : real * real}
- *#cartel(r);*
> val it = "hola" : string
- *#2(#tupla r);*
> val it = 2.78 : real

Una *lista* es un agregado de valores de un mismo tipo. Ejemplos de listas son

- *[1,2,3];*
> val it = [1, 2, 3] : int list
- *[true, true, false, true];*
> val it = [true, true, false, true] : bool list
- *[#”a”, #”z”];*
> val it = [#"a", #"z"] : char list
- *[”hola”];*
> val it = ["hola"] : string list
- *[];*
> val 'a it = [] : 'a list

El último ejemplo es una lista sin elementos, y se llama lista *vacía*; como no tiene elementos, *ML*le asigna un tipo “indefinido” o *“paramétrico”*.

A.6.1 Operaciones con listas.

Nombre	Descripción	Tipo
`::`	Inserta un elemento a la izquierda de una lista	`'a * 'a list -> 'a list`
`hd`	Retorna el primer elemento de una lista (*head*)	`'a list -> 'a`
`tl`	Retorna la lista sin su primer elemento (*tail*)	`'a list -> 'a list`
`null`	Retorna *true* si la lista es vacía	`'a list -> bool`
`@`	Concatena dos listas	`'a list * 'a list -> 'a list`
`length`	Retorna cuántos elementos tiene una lista	`'a list * 'a list -> 'a list`

Ejemplos de estas operaciones.

```
- val x = [1,2,3];
> val x = [1, 2, 3] : int list
- val y = [88, 99];
> val y = [88, 99] : int list
- hd x;
> val it = 1 : int
- tl y;
> val it = [99] : int list
- hd(x)::tl(y);
> val it = [1, 99] : int list
- x@y;
> val it = [1, 2, 3, 88, 99] : int list
- null x;
> val it = false : bool
- null [];
> val it = true : bool
- []@x;
> val it = [1, 2, 3] : int list
```

La última línea muestra que la lista vacía es el elemento nulo de la concatenación.

A.7 Vectores.

Los vectores son similares a las listas, pero permiten acceder a todos sus elementos con una sola operación `sub`. Para usar las funciones de vectores, si se usa *MosML*, hay que cargar el módulo `Vector` con `load "Vector";`.

```
- val v1 = #[1, 2, 3];
> val v1 = #[1, 2, 3] : int vector
```

```
- val v2 = #[[1, 2], [3]];
> val v2 = #[[1, 2], [3]] : int list vector
- Vector.sub(v1, 1);
> val it = 2 : int
- Vector.sub(v2, 1);
> val it = [3] : int list
```

También pueden formarse a partir de listas, concatenarse entre sí, convertirse a listas, y consultar la cantidad de sus elementos.

```
- val l1 = [1, 2, 3, 4];
> val l1 = [1, 2, 3, 4] : int list
- val l2 = [[1, 2], [3], [4]];
> val l2 = [[1, 2], [3], [4]] : int list list
- val v1 = Vector.fromList l1;
> val v1 = #[1, 2, 3, 4] : int vector
- val v2 = Vector.fromList l2;
> val v2 = #[[1, 2], [3], [4]] : int list vector
- val v3 = Vector.concat[v1, #[10, 11]];
> val v3 = #[1, 2, 3, 4, 10, 11] : int vector
- Vector.length v2;
> val it = 3 : int
- Vector.length v3;
> val it = 6 : int
```

A.8 Definición de funciones.

ML permite definir funciones nuevas, mediante la sintaxis

`fun` *nombre argumento*`=` *definición*

Por supuesto, estas funciones se aplican de la misma manera que las funciones primitivas. Para hacer funciones de varios argumentos se usan tuplas. Ejemplos de esto son

```
- fun doble x = 2*x;
> val doble = fn : int -> int
- fun esPar n = n mod 2 = 0;
> val esPar = fn : int -> bool
- fun promedio(m, n) = (real(m)+real(n))/2.0;
> val promedio = fn : int * int -> real
- fun fact n = if n=0 then 1 else n*fact(n-1);
> val fact = fn : int -> int
- doble 3;
> val it = 6 : int
- esPar 45;
> val it = false : bool
- promedio(7, 8);
> val it = 7.5 : real
- fact 7;
```

```
> val it = 5040 : int
```

El último ejemplo muestra que uns función puede ser recursiva; de hecho, éste es el principal método de programación en *ML*, como en los demás lenguajes funcionales.

A.8.1 Operaciones con funciones.

Nombre	Descripción	Tipo
`o`	Compone dos funciones	`('a -> 'b) * ('c -> 'a) -> 'c -> 'b`

Ejemplo.

```
- fun f1 x = x+1;
> val f1 = fn : int -> int
- val c = chr o f1 o ord;
> val c = fn : char -> char
- c #"A";
> val it = #"B" : char
```

A.9 *Bindings* locales.

ML tiene un mecanismo para permitir *bindings* que sólo puedan usarse en una región definida del programa, y no fuera de ella: el bloque `let` ··· `in` ··· `end`.

Veamos un ejemplo.

```
- val x = 10;
> val x = 10 : int
- val y = let val z = 11 in x+z end;
> val y = 21 : int
- val w = x+z;
! Toplevel input:
! val w = x+z;
!           ^
! Unbound value identifier: z
```

En la última evaluación, `z` no está definida: sólo se puede usar dentro de la expresión entre `in` y `end` en el `let` correspondiente; incluso no tiene existencia si el `let` no se está evaluando.

Todo *binding* puede figurar en un `let`, funciones inclusive. El resultado del `let` es el valor de su expresión. Si hay mś de un *binding*, éstos se evalúan por orden de aparición, y cada *binding* definido puede ser usado por los siguientes en el bloque.

A.10 Funciones anónimas y de orden superior.

ML permite definir funciones anónimas, mediante la sintaxis

`fn` *argumento* `=>` *expresión*

Estas funciones pueden ser pasadas como argumento a otras funciones, llamads de orden superior, retornadas como resultados, o se les puede asignar un nombre.

```
(* f es de orden superior *)
-  fun f(g, x) = g x;
> val ('a, 'b) f = fn : ('a -> 'b) * 'a -> 'b
(* lo mismo h *)
-  fun h m = fn n => m+n;
> val h = fn : int -> int -> int
-  f(fn x => 3*x, 5);
> val it = 15 : int
-  h 4;
> val it = fn : int -> int
-  h 4 5; (* se entiende como ((h 4) 5)*)
> val it = 9 : int
```

De hecho, cualquier definición como *fun f x = x+1* es tratada por *ML* como si fuera *val f = fn x => x+1*; en el caso de, p.ej., *fact*, que es recursiva, como *val rec fact = fn n => if n=0 then 1 else n*fact(n-1)*.

A.11 *Pattern matching.*

Una *plantilla* (o *pattern*) es una estructura con nombres y/o valores, y que puede confrontarse con un valor. Si es posible asignar valores a los nombres, sin ambigüedades, se dice que el valor *coincide* con la plantilla (*matches the pattern*); en caso contrario, no coincide.

Las plantillas se definen así:

Plantilla	Coincidencia
Constante (entera, booleana, etc.)	Ese valor
Un nombre	Cualquier valor
_	Idem un nombre, pero sin *binding*
Una tupla	Una tupla de igual cantidad de elementos, y que los elementos correspondientes de ambas tuplas coincidan
`[]` o `nil`	Lista vacía
Una lista detallada	Con otra lista de igual tipo y longitud, y que los elementos correspondientes coincidan
Una lista especificada como *cabeza*::*cola*, donde *cabeza* y *cola* son plantillas	Con una lista donde ambas cabezas y colas coincidan

Nótese que la definición anterior es recursiva. Recordar que un nombre, además de coincidir, provoca *binding*.

Siguen ejemplos.

```
- val (x, (y, 1)) = ("guau!", (true, 1));
> val x = "guau!" : string
  val y = true : bool
- val [i1, i2] = [1000, 2000];
> val i1 = 1000 : int
  val i2 = 2000 : int
- val l = [[1], [2,3], [4], [5,6,7]];
> val l = [[1], [2, 3], [4], [5, 6, 7]] : int list list
- val h::t = l;
> val h = [1] : int list
  val t = [[2, 3], [4], [5, 6, 7]] : int list list
```

Las plantillas permites definir funciones con expresiones múltiples, seleccionadas mediante *pattern matching*. Como ejemplo, el factorial puede definirse así

```
fun fact 0 = 1
| fact n = n*fact(n-1)
```

Hay que tener en cuenta que las plantillas se prueban por orden de aparición, y la primera que coincide evalúa la expresión; las restantes son ignoradas.

*ML*puede usar las plantillas usando `case` ··· `of`. Usando esto, el factorial puede rehacerse así

```
fun fact n =
    case n of
    0 => 1
    | n => n*fact(n-1)
```

Las plantillas pueden también usarse en funciones anónimas. Sigue otra versión de factorial.

```
val rec fact = fn 0 => 1 | n => n*fact(n-1)
```

A.12 Funciones infijas, asociatividades y precedencias.

Hasta ahora, *ML* parece tener dos tipos de funciones; las *prefijas*, que se invocan `F`*(x, y)*, y las *infijas*, que se usan *x*`F`*y*. Esta última notación se usa, además de en las operaciones denotadas por signos, sino también para otras funciones como `div` y `mod`. Podemos convertir cualquier función en infija, siempre que su tipo sea `'a * 'b -> 'c`, usando las declaraciones `infix`, `infixr` y `nonfix`. `Infix` hace que la composición de una función con sí misma sea asociativa a izquierda, `infixr` a derecha, y `nonfix` que no se puedan componer.

```
- fun f(x, y) = x+2*y;
> val f = fn : int * int -> int
- infix f;
> infix 0 f
- 1 f 2 f 3;
> val it = 11 : int
```

Se pueden indicar las precedencias con números entre cero y nueve en las declaraciones; por supuesto, a mayor número, mayor precedencia.

Inversamente, con `op` podemos convertir una función infija en una prefija.

```
- op+(2, 3);
> val it = 5 : int
- op::(1, [2, 3]);
> val it = [1, 2, 3] : int list
```

A.13 Excepciones.

Las excepciones son un mecanismo usado por *ML* para indicar la aparición de eventos anormales, o no muy esperados; pueden señalar tanto verdaderos errores como situaciones que no se desean alcanzar.

ML tiene muchas excepciones predefinidas, de las cuales podemos nombrar

Excepción	Situación
`Bind`	Intento de *binding* sin coincidencia de plantilla
`Chr`	Intento de formar un caracter irrepresentable
`Div`	Intento de división por cero
`Domain`	Intento de aplicar una función fuera de su dominio
`Empty`	Operación imposible sobre una lista vacía
`Match`	No hay coincidencia en un *pattern matching*
`Overflow`	Intento de computar un número fuera de cotas
`Subscript`	Intento de acceder a un elemento inexistente en una lista o *string*
`Size`	Intento de crear una *string* demasiado grande

Normalmente, la aparición de una excepción aborta el programa, como se muestra.

```
- hd [];
! Uncaught exception:
! Empty
```

ML permite que, al evaluar una expresión y producirse una excepción, ésta sea "capturada" y sus efectos controlados; esto se hace usando *pattern matching*. Sigue un ejemplo.

```
- (hd []) handle Empty => 0;
> val it = 0 : int
```

¿Qué pasó? Pues que, al producirse `Empty`, `handle Empty` la capturó y devolvió cero como retorno.

Podemos capturar varias excepciones con un mismo `handler`, incluyendo excepciones desconocidas.

```
- fun cosa 1 = 1 div 0 (* Div *)
| cosa 2 = hd [] (* Empty *)
| cosa _ = trunc(Math.sqrt( 1.0)) (* Domain *)
;
> val cosa = fn : int -> int
- fun eval n =
    (cosa n) handle Empty => ~10
                  | Div => ~20
                  | _ => ~30 (* cualquier exc. *)
;
> val eval = fn : int -> int
- eval 1;
> val it = ~20 : int
- eval 2;
> val it = ~10 : int
- eval 25;
> val it = ~30 : int
```

A.14 *Type*, *datatype* y *abstype*.

ML permite definir sinónimos para tipos con `type`.

```
- type IL = int list;
> type IL = int list
```

También permite crear nuevos tipos, definiendo sus constantes, con `datatype`.

```
- datatype Color = Blanco | Negro;
> New type names: =Color
  datatype Color = (Color,{con Blanco : Color, con Negro : Color})
  con Blanco = Blanco : Color
  con Negro = Negro : Color
- Negro;
> val it = Negro : Color
- fun negativo Negro = Blanco
 | negativo Blanco = Negro;
> val negativo = fn : Color -> Color
- negativo Negro;
> val it = Blanco : Color
```

`Blanco` y `Negro` se denominan los *constructores* del tipo, y se pueden usar en *pattern matching*.

Los constructores también pueden tomar argumentos, pasarse como argumentos y devolverse como resultados.

```
- datatype Cosa = E of int | T of bool * bool;
> New type names: =Cosa
  datatype Cosa = (Cosa,{con E : int -> Cosa, con T : bool * bool -> Cosa})
  con E = fn : int -> Cosa
  con T = fn : bool * bool -> Cosa
- T(true, false);
> val it = T(true, false) : Cosa
- fun f(x, y, z) = x(y, z);
> val ('a, 'b, 'c) f = fn : ('a * 'b -> 'c) * 'a * 'b -> 'c
- f(T, true, true);
> val it = T(true, true) : Cosa
- fun g() = E;
> val g = fn : unit -> int -> Cosa
```

Pueden crearse tipos *paramétricos*.

```
- datatype 'a Arbol = Hoja of 'a | Nodo of 'a Arbol * 'a Arbol;
> New type names: =Arbol
  datatype 'a Arbol =
  ('a Arbol,
   {con 'a Hoja : 'a -> 'a Arbol,
    con 'a Nodo : 'a Arbol * 'a Arbol -> 'a Arbol})
  con 'a Hoja = fn : 'a -> 'a Arbol
```

```
  con 'a Nodo = fn : 'a Arbol * 'a Arbol -> 'a Arbol
- Hoja(3);
> val it = Hoja 3 : int Arbol
```

`Datatype` es lo suficientemente potente como para permitir declarar el combinador *Y*.

```
- datatype 'a T = T of 'a T -> 'a;
> New type names: T
  datatype 'a T = ('a T,{con 'a T : ('a T -> 'a) -> 'a T})
  con 'a T = fn : ('a T -> 'a) -> 'a T
- val Y =
    fn f => (
        (fn (T x) => fn a => (f (x (T x))) a)
        (T (fn (T x) => fn a => (f (x (T x))) a))
    );
> val ('a, 'b) Y = fn : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b
- val ff = fn f => fn x => if x=1 then 1 else x*f(x-1);
> val ff = fn : (int -> int) -> int -> int
- val fact = Y ff;
> val fact = fn : int -> int
- map fact [1, 2, 3, 4];
> val it = [1, 2, 6, 24] : int list
```

ML permite definir tipos con constructores "ocultos".

```
- abstype T = A of int | B of string
with
fun f i = A i
fun g s = B s
end;
> New type names: T
  type T = T
  val f = fn : int -> T
  val g = fn : string -> T
- f 10;
> val it = <T> : T
- g "hola";
> val it = <T> : T
- A 10;
! Toplevel input:
! A 10;
! ^
! Unbound value identifier: A
- B "hola";
! Toplevel input:
! B "hola";
! ^
```

```
! Unbound value identifier: B
```

En el caso anterior, sólo `f` y `g` tienen acceso a los constructores `A` y `B`.

A.15 *Datatypes* especiales.

ML viene con `datatypes` ya declarados.

A.15.1 `Option.`

Permite tener valores *"opcionales"*. Su definición es

```
datatype 'a option = NONE | SOME of 'a
```

Normalmente se usa con la función `valOf`.

```
fun valOf(SOME x) = x
| valOf NONE = raise Option
```

A.15.2 `Ref.`

Permite tener valores *mutables*. Su definición es

```
datatype 'a ref = ref of 'a
```

Se usa normalmente con `!` y `:=`.

```
- val r = ref 13;
> val r = ref 13 : int ref
- r;
> val it = ref 13 : int ref
- !r;
> val it = 13 : int
- r:= !r+1; (* note el espacio entre := y !*)
> val it = () : unit
- !r;
> val it = 14 : int
```

A.15.3 `Order.`

Usado para las comparaciones (funciones `compare`).

```
datatype order = LESS | EQUAL | GREATER
```

A.16 Iteraciones.

El *datatype* `ref` permite iteraciones à la *C*, p.ej., usando `while`.

```
- val i = ref 0;
> val i = ref 0 : int ref
- while !i < 4 do (print "ay!\n"; i:= !i+1);
ay!
ay!
ay!
ay!
> val it = () : unit
```

A.17 Definiciones locales.

En ocasiones, el sistema de módulos es demasiado potente (ver A.18. *ML* permite hacer declaraciones locales persistentes (similares a las variables `static` de *C*), usadas por funciones del siguiente modo.

```
- local
  val r = ref 0
in
  fun inicial n = r:=n
  fun serie() =
    let val ret = !r
        val () = r := !r+1
    in ret end
end
> val inicial = fn : int -> unit
  val serie = fn : unit -> int
```

A.18 Módulos.

Un *módulo* o *estructura* es similar a una *clase* de *C++*(Stroustrup 1991), a un módulo de *Modula/2* (Wirth 1987), o a una unidad de *Ada*: es un mecanismo que permite definir valores y funciones, controlando cuáles de éstos serán visibles desde afuera. Un ejemplo puede ser

```
- structure S =
struct
val i = 12
fun masI n = n + i
end;
> structure S : {val i : int, val masI : int -> int}
- S.masI 67;
> val it = 79 : int
```

El prefijo `S.` es la modo de acceder a cualquier valor o función declarado en la estructura. Otra forma es "abrir" la estructura, con lo que no es necesario el prefijo.

```
- structure S =
struct
val i = 12
fun masI n = n + i
end;
> structure S : {val i : int, val masI : int -> int}
- open S;
> val i = 12 : int
  val masI = fn : int -> int
- masI 67;
> val it = 79 : int
```

Con una *signatura*[8] se pueden hacer varias cosas: una de ellas es asegurarse que una estructura implemente ciertos valores de tipos dados.

```
- signature S =
sig
val f: int -> int
end;
> signature S = {val f : int -> int}
- structure IS1: S =
struct
fun f x = x + 1
end;
> structure IS1 : {val f : int -> int}
- structure IS2: S =
struct
val f = 23
end;
! Toplevel input:
! ............: S =
! struct
! val f = 23
! end.
! Signature mismatch: the module does not match the signature ...
! Scheme mismatch: value identifier f
! is specified with type scheme
!   val f : int -> int
! in the signature
! but its declaration has the unrelated type scheme
!   val f : int
! in the module
```

[8] Firma.

```
! The declared type scheme should be at least as general
    as the specified type scheme
```

También se pueden usar para controlar qué definiciones sean visibles fuera.

```
- structure S :
sig
val masI: int -> int
end
=
struct
val i = 12
fun masI n = n+i
end;
> structure S : {val masI : int -> int}
- S.masI 17;
> val it = 29 : int
- S.i;
! Toplevel input:
! S.i;
! ^^^
! Unbound value component: S.i
```

Las signaturas se pueden definir aparte y usarse *a posteriori*.

```
- signature S1 =
sig
val masI: int -> int
val i: int
end;
> signature S1 = {val masI : int -> int, val i : int}
- signature S2 =
sig
val masI: int -> int
end;
> signature S2 = {val masI : int -> int}
- structure IS1: S1 =
struct
val i = 17
fun masI n = n + i
end;
> structure IS1 : {val i : int, val masI : int -> int}
- IS1.masI;
> val it = fn : int -> int
- IS1.i;
> val it = 17 : int
- structure IS2:> S2 =
struct
```

```
val i = 17
fun masI n = n + i
end;
> structure IS2 : {val masI : int -> int}
- IS2.masI;
> val it = fn : int -> int
- IS2.i;
! Toplevel input:
! IS2.i;
! ^^^^^
! Unbound value component: IS2.i
```

Si la signatura se coloca en un archivo aparte, la restricción en la declaración de una estructura se indica con `:>`.

```
structure S :> S1 =
struct
val i = 17
fun masI n = n + i
end
```

A.19 Functores.

Un *functor* es el equivalente a un *template* de *C++*: un mecanismo que genera un módulo a partir de otro(s). Un *functor* toma una estructura que cumpla una signatura dada y devuelve una nueva estructura; un *functor* puede usar polimorfismo.

```
- signature TIPO =
sig
type a
type b
end;
> signature TIPO = /\a b.{type a = a, type b = b}
- functor Pila(S: TIPO) :
sig
val push: S.a * S.b -> unit
val pop: unit -> S.a * S.b
end
=
struct
val p = ref []
fun push(x, y) = p:=(x, y):: !p
fun pop() = hd(!p) before p:=tl(!p)
end;
> functor Pila :
  !a b.
```

```
  {type a = a, type b = b}->
      {val pop : unit -> a * b, val push : a * b -> unit}
- structure P1 = Pila(struct type a=int type b=bool end);
> structure P1 : {val pop : unit -> int * bool, val push : int * bool -> unit}
- structure P2 = Pila(struct type a=real type b=char end);
> structure P2 :
  {val pop : unit -> real * char, val push : real * char -> unit}
- P1.push(10, true);
> val it = () : unit
- P2.push(3.14, #"P");
> val it = () : unit
- P1.pop();
> val it = (10, true) : int * bool
- P2.pop();
> val it = (3.14, #"P") : real * char
```

A.20 Entrada/salida.

ML reúne las acilidades para entrada/salida en los módulos `TextIO` y `BinIO`. `TextIO` define los tipos de *streams* [9] `instream`, para entrada, y `outstream` para salida. También define los *streams* usuales.

Nombre	Tipo	Descripción
`stdIn`	`instream`	Entrada estándar.
`stdOut`	`outstream`	Salida estándar.
`stdErr`	`outstream`	Salida de error.

Las funciones para manipular las entradas son

Nombre	Tipo	
`openIn`	`string -> instream`	Apertura para lectura.
`closeIn`	`instream -> unit`	Cierra lectura.
`input`	`instream -> string`	Lee lo que puede.
`input1`	`instream -> char option`	Lee un caracter, si es posible.
`inputN`	`instream -> string`	Lee lo que puede.
`inputAll`	`instream -> string`	Lee hasta `EOF`.
`inputNoBlock`	`instream -> string option`	Lee, si puede; retorna una opción.
`inputLine`	`outstream -> string`	Lee una línea.

Para las salidas, tenemos

[9] Archivos secuenciales.

Nombre	Tipo	
`openOut`	`string -> outstream`	Apertura para escritura (con truncamiento).
`openAppend`	`string -> outstream`	Apertura para escritura (modo *append*).
`closeOut`	`outstream -> unit`	Cierra escritura.
`output`	`outstream * string -> string`	Escribe una `string`.
`output1`	`outstream * char -> char option`	Escribe un caracter,

Siguen dos ejemplos. El primero.

```
- fun siONo() =
    let val sale = ref false
        val resp = ref ""
    in
        while not(!sale) do (
            print "S/N...
n";
            case TextIO.input1 TextIO.stdIn of
            SOME #"S" => (resp:="S"; sale := true)
            — SOME #"N" => (resp:="N"; sale := true)
            — NONE => (resp:="?"; sale:=true)
            — _ => ()
        );
        !resp
    end;
> val siONo = fn : unit -> string
- siONo();
S/N...
> val it = "?" : string
```

El segundo.

```
- fun copia(origen, destino) =
    let val orig = TextIO.openIn origen
        val dest = TextIO.openOut destino
        val fin = ref false
        val buff = ref ""
    in
        while not(!fin) do (
            buff:=TextIO.input orig;
            if size(!buff)>0 then
                TextIO.output(dest, !buff)
            else fin:=true
        );
        TextIO.closeIn orig;
```

TextIO.closeOut dest
end;
```
> val copia = fn : string * string -> unit
```
\- *copia("zzz1", "zzz2");*
```
> val it = () : unit
```

A.21 El compilador `mosmlc`.

La distribución de *MosML* trae un compilador capaz de producir ejecutables, llamado `mosmlc`. El equivalente al *main* de *C* es, en *MosML*, una expresión llamada `_` o `it`. Si el archivo `hello.sml` contiene
```
val _ = print "Hello world!\n"
```
la siguiente orden
`mosmlc -o hello hello.sml`,
producirá un ejecutable llamado `hello`.

Bibliografía.

Abelson, H. and G. J. Sussman (1.996). *Structure and Interpretation of Computer Programs* (second ed.). Cambridge, MA: The MIT Press and McGraw–Hill.

Aho, A. V., R. Sethi, and J. D. Ullman (1.986). *Compilers: Principles, Techniques, and Tools*. Reading, MA, USA: Addison–Wesley.

Appel, A. E. (1.998). *Modern Compiler Implementation in ML* (first ed.). Cambridge CN2 1RP, UK: Cambridge University Press. Excelente libro. Usa técnicas bastante más avanzadas (y complejas) que las usadas aquí.

Bird, R. J. (1.998). *Introduction to Functional Programming Using Haskell*. Englewood Cliffs, NJ 07632, USA: Prentice–Hall. La continuación de un clásico.

Grune, D., H. E. Bal, C. J. H. Jacobs, and K. G. Langendoen (2.001). *Modern Compiler Design*. New York, NY, USA: John Wiley & Sons, Ltd. De estilo enciclopédico. Muy actualizado.

Hopcroft, J. E. and J. Ullman (1.993). *Introducción a la Teoría de Autómatas, Lenguajes y Computación*. México, D.F.: COMPAÑIA EDITORIAL CONTINENTAL, S.A. DE C.V. Buen tratamiento de la teoría subyacente.

Kamin, S. N. (1.990). *Programming Languages: An Interpreter–Based Approach* (first ed.). Reading, MA, USA: Addison–Wesley. Excelente discusión sobre varios paradigmas. Incluye intérpretes *LISP*, *APL*, *Scheme*, *SASL*, *CLU*, *Smalltalk* y *Prolog*. En las discusiones presenta también a *ML*, λ–cálculo y *Ada*.

Kernighan, B. and D. Ritchie (1.988). *The C Programming Language*. Englewood Cliffs, NJ 07632, USA: Prentice–Hall. El primer libro sobre *C* y, a la fecha, el mejor. Nonca programe sin él.

Mandrioli, D. and C. Ghezzi (1.987). *Theoretical Foundations of Computer Science*. New York, NY, USA: John Wiley & Sons, Ltd. Excelente introducción a la teoría de lenguajes y máquinas abstractas.

Milner, R., M. Tofte, R. Harper, and D. MacQueen (1.997). *The Definition of Standard ML (Revised)*. Cambridge, MA, USA: The MIT Press. Expone la definición formal de *ML*. Su estilo es, por fuerza, bastante seco; aún así, su lectura es, casi, una obligación.

Paulson, L. C. (1.988). *ML for the Working Programmer* (second ed.). University Press, Cambridge, UK: Cambridge University Press. Uno de los mejores libros, sobre *ML* en particular, y sobre programación, en general. Incluye un intérprete para λ–cálculo (en sus variantes *eager* y *lazy*) y un probador táctico de teoremas. Sumamente recomendable.

Plasmeijer, R. and M. v. Eekelen (1.993). *Functional Programming and Parallel Graph Rewriting*. Reading, MA, USA: Addison–Wesley.

Stroustrup, B. (1.991). *The C++ Programming Language* (second ed.). Reading, MA, USA: Addison–Wesley. La referencia clásica sobre *C++*. Muy completo.

Ullman, J. D. (1.989a). *Principles of Database and Knowledgebase Systems*, Volume 1. Rockville MD 20850, USA: Computer Science Press. Posiblemente el mejor tratado sobre bases de datos.

Ullman, J. D. (1.989b). *Principles of Database and Knowledgebase Systems*, Volume 2. Rockville MD 20850, USA: Computer Science Press. Posiblemente el mejor tratado sobre bases de datos.

Ullman, J. D. (1.998). *Elements of ML Programming*. Englewood Cliffs, NJ 07632, USA: Prentice–Hall. Más actual y detallado que (Wikström 1987).

Wikström, Å. (1.987). *Functional Programming Using Standard ML*. Englewood Cliffs, NJ 07632, USA: Prentice–Hall. Un clásico. De ritmo pausado quizás en demasía y, a la fecha, algo obsoleto.

Wirth, N. (1.987). *Programación en Modula–2* (tercera corregida ed.). Buenos Aires, Argentina: Librería "El Ateneo" Editorial. Buen libro sobre *MODULA/2*. No se caracteriza por ser ameno, y la traducción es deficiente.

Indice.

1 El intérprete μSQL. **1**
- 1.1 Archivos fuentes. 1
- 1.2 Implementación de las estructuras para el Algebra Relacional. . . 2
- 1.3 Simplificaciones para μSQL. 3
- 1.4 Proyección. 4
- 1.5 Selección. 4
- 1.6 Producto cartesiano. 6
- 1.7 Mostrando los resultados. 7
- 1.8 Las tablas. 7
- 1.9 Arranque de μSQL. 9
- 1.10 Juntando las piezas. 10
- 1.11 El *scanner*. 11
- 1.12 El *parser*. 14
- 1.13 Ejemplos. 17
- 1.14 Extensiones propuestas. 19
- 1.15 Diagramas ferroviarios de la gramática 20

A Breve Introducción a *ML*. **20**
- A.1 Los primeros pasos. 20
- A.2 Valores y tipos primitivos. 22
  - A.2.1 *Strings* constantes. 22
- A.3 Operaciones sobre tipos primitivos. 23
  - A.3.1 Operaciones `int -> int`. 23
  - A.3.2 Operaciones `int * int -> int`. 23
  - A.3.3 Operaciones `int * int -> bool`. 23
  - A.3.4 Operaciones `real -> real`. 23
  - A.3.5 Operaciones `real * real -> real`. 24
  - A.3.6 Operaciones `real * real -> bool`. 24
  - A.3.7 Operaciones `real -> int`. 24
  - A.3.8 Operaciones `int -> real`. 24
  - A.3.9 Operaciones `int -> char`. 24
  - A.3.10 Operaciones `char * char -> bool`. 24
  - A.3.11 Operaciones `char -> int`. 25
  - A.3.12 Operaciones `char -> string`. 25
  - A.3.13 Operaciones `string * string -> string`. 25
  - A.3.14 Operaciones `string * string -> bool`. 25
  - A.3.15 Operaciones `bool * bool -> bool`. 25
  - A.3.16 Ejemplos. 25
- A.4 Secuencias. 26
- A.5 *Bindings*. 26
- A.6 Tuplas, *records* y listas. 26
  - A.6.1 Operaciones con listas. 28
- A.7 Vectores. 28
- A.8 Definición de funciones. 29

A.8.1 Operaciones con funciones. 30
A.9 *Bindings* locales. 30
A.10 Funciones anónimas y de orden superior. 31
A.11 *Pattern matching.* 31
A.12 Funciones infijas, asociatividades y precedencias. 33
A.13 Excepciones. 33
A.14 *Type*, *datatype* y *abstype*. 35
A.15 *Datatypes* especiales. 37
A.15.1 `Option`. 37
A.15.2 `Ref`. 37
A.15.3 `Order`. 37
A.16 Iteraciones. 38
A.17 Definiciones locales. 38
A.18 Módulos. 38
A.19 Functores. 41
A.20 Entrada/salida. 42
A.21 El compilador `mosmlc`. 44