
UNIVERSIDAD DE BUENOS AIRES
FACULTAD DE INGENIERÍA

TEORÍA DE LENGUAJES (75.31)

TRABAJO PRÁCTICO

# Trenes: Playa de maniobras

Demian Ferrerio 88443 epidemian@gmail.com

4 de julio de 2011

# Índice

# 1. General

Gracias al soporte de módulos que brinda Oz (a los que llama *functors*), el diseño de la solución pudo plantearse en distintas partes, cada una dedicada a resolver un subdominio del problema general.

## 1.1. Archivos

Los módlos de la solución se mapean en forma directa a archivos en el directorio `src` del proyecto. Éstos archivos son:

- `Lista.oz`: define las funciones para trabajar con listas utilizadas en el resto de la aplicación.
- `Trenes.oz`: tiene las funciones para el manejo de trenes y sus movimientos.
- `PruebasLista.oz`: las pruebas del módulo `Lista`.
- `PruebasTrenes.oz`: las pruebas del módulo `Trenes`.
- `Prueba.oz`: las funciones básicas para hacer pruebas.
- `Visualizador.oz`: las funciones para visualizar gráficamente los trenes y sus movimientos.
- `TrenesPrincipal.oz`: el punto de entrada de la aplicación.
- `makefile.oz`: archivo utilizario usado para compilar el proyecto.

## 1.2. Cómo Compilar y Ejecutar

Para compilar el proyecto, básta con ejecutar desde la linea de comandos, sobre el directorio `src`:

```
$ ozmake
```

Esto generará los archvios `.ozf`, los functors, correspondientes a cada archivo `.oz`.

Una vez compilado, la aplicación puede ejecutarse con:

```
$ ozengine TrenesPrincipal.ozf
```

Las prubeas, a su vez, pueden ejecutarse de forma análoga:

```
$ ozengine PruebasLista.ozf
$ ozengine PruebasTrenes.ozf
```

### 1.3. Herramientas Utilizadas

Para la realización del trabajo práctico se utilizó:

- **Mozart/Emacs** Para codificar y hacer pruebas interactivas.
- Las herramientas de linea de comandos de Mozart: **ozmake** y **ozengine** para compilar y ejecutar las pruebas automáticas y el programa principal.
- El visualizador de Peter Van Roy y colaboradores[1] para algunas pruebas interactivas y para mejor presentación del trabajo.
- LaTeX para la redacción de éste informe.

## 2. Pruebas

Una de las ventajas de programar en el paradigma funcional es la facilidad con la que se pude probar el código. Al no tener estado mutable, o *side effect*, hacer "TDD" (*Test Driven Development*) se reduce a especificar la salida esperada de una función dados algunos valores de sus parámetros.

### 2.1. Módulo de Pruebas

Con el fin de facilitar la definición de pruebas, se escribió un sencillo módulo utilitario de pruebas. Su definición completa es:

[1] http://www.info.ucl.ac.be/people/PVR/ds/CS2104/Assignments/Visualizer.zip

```
functor
export
   NuevaPrueba
import
   System
define
   fun {NuevaPrueba}
      Total = {NewCell 0} % Cantidad de pruebas realizadas
      Exitosas = {NewCell 0} % Cantidad de pruebas exitosas
      proc {Iguales Valor Esperado Descripcion}
         if Valor == Esperado then
            Exitosas := @Exitosas + 1
         else
            % show -> con endline. print -> sin endline.
            % printInfo/showInfo -> sólo strings (no muestran bien las listas)
            {System.showInfo 'Fallo en prueba: '#Descripcion}
            {System.printInfo '  Se esperaba: '}
            {System.show Esperado}
            {System.printInfo '  Pero se obtuvo: '}
            {System.show Valor}
         end
         Total := @Total + 1
      end
      proc {Finalizar}
         if @Total == @Exitosas then
            {System.showInfo 'Todas las pruebas pasaron exitosamente!'}
         end
         {System.showInfo 'Pruebas exitosas/total: '#@Exitosas#'/'#@Total}
      end
   in
      prueba(iguales:Iguales finalizar:Finalizar)
   end
end
```

Lo más importante de éste módulo es el procedimiento `Iguales` que evalúa si dos valores, el esperado y el obtenido en una prueba, son iguales y, en caso de no serlo, imprime por pantalla un mensaje de error con la descripción de la prueba y los valores testeados.

Se puede observar que, irónicamente, en vez de utilizarse un diseño fucional puro, se optó por un diseño "orientado a objetos" (o, con estado). Esto se hizo así méramente para poder contar la cantidad de pruebas realizadas de una forma sencilla. Distintos paradigmas pueden resolver distintos problemas de forma más o menos "natural". Y así, Oz, al no forzar un paradigma determinado permite expresar una solución de la forma que más natural le resulte al programador.

# 3. Listas

## 3.1. Pruebas

Si bien la implementación de las funciones para el manejo de listas es sencilla, es conveniente escribir antes algunas pruebas que especifiquen el comportamiento esperado de las mismas.

Por ejemplo, éstas podrían ser las pruebas de la función `Tomar`:

```
P = {Prueba.nuevaPrueba}
L = [1 2 3 2 1]
{P.iguales {Lista.tomar L 3} [1 2 3] 'Tomar con N < Longitud'}
{P.iguales {Lista.tomar L 0} nil 'Tomar con N == 0'}
{P.iguales {Lista.tomar L 6} L 'Tomar con N > Longitud'}
```

Se puede empezar con una implementación vacía de `tomar` ...

```
fun {Tomar Xs N}
   nil
end
```

... y ejecutar las pruebas. Estas, obviamente, fallan y nos idican qué es lo que se esperaba:

```
$ ozengine PruebasLista.ozf
Fallo en prueba: Tomar con N < Longitud
  Se esperaba: [1 2 3]
  Pero se obtuvo: nil
Fallo en prueba: Tomar con N > Longitud
  Se esperaba: [1 2 3 2 1]
  Pero se obtuvo: nil
Pruebas exitosas/total: 2/3
```

Así, se puede ir implementando las funciones a medida que se van especificando en forma de pruebas automatizadas.

Más importante aún, se puede escribir una implementación rápida y desprolija de una función, sólo para probar si de esa forma anda, y luego refactorizarla de una forma más prolija y elegante y coroborar que sigue andando. La confianza que se gana a la hora de refactorizar es uno de las ventajas más importantes de hacer TDD.

## 3.2. Implementación

Las funciones del módulo `Lista` no resultan muy interesantes, ya que son bastante sencillas. Todas se basan en interar una lista recursivamente y checkear la condición de corte con pattern matching.

La implementación entera del módulo es:

```
functor
export
   Longitud
   Tomar
   Eliminar
   Agregar
   Miembro
   Posicion
define
   % Devuelve la longitud de la lista Xs.
   fun {Longitud Xs}
      case Xs of nil then 0
      [] _|Xr then 1 + {Longitud Xr}
      end
   end

   % Devuelve una lista con los primeros {Min N {Longitud Xs}} elementos de la
   % lista Xs.
   fun {Tomar Xs N}
      case Xs of nil then nil
      [] X|Xr then
         if N == 0 then nil else X|{Tomar Xr N-1} end
      end
   end

   % Devuelve la lista que resulta de eliminar los primeros
   % {Min N {Longitud Xs}} elementos de la lista Xs.
   fun {Eliminar Xs N}
      case Xs of nil then nil
      [] _|Xr then
         if N == 0 then Xs else {Eliminar Xr N-1} end
      end
   end

   % Devuelve la lista que resulta de concatenar Xs y Ys.
   fun {Agregar Xs Ys}
      case Xs of nil then Ys
      [] X|Xr then X|{Agregar Xr Ys}
```

```
      end
   end

   % Verifica si Y pertenece a la lista Xs.
   fun {Miembro Xs Y}
      case Xs of nil then false
      [] X|Xr then X == Y orelse {Miembro Xr Y} end
   end

   % Devuelve la posición de Y en la lista Xs siendo la primer posición la 1.
   fun {Posicion Xs Y}
      case Xs of nil then raise elementoNoPertenece(Y Xs) end
      [] X|Xr then
         if Y == X then 1 else 1 + {Posicion Xr Y} end
      end
   end
end
```

# 4. Trenes

El módulo `Trenes` tiene las funciones necesarias para el manejo de trenes y sus movimientos. Las estructuras de datos utilizadas y las funciones se implementaron de acuerdo a las pautas del enunciado.

## 4.1. Aplicar Movimientos

La función `AplicarMovimientos` calcula los estados por los que pasa una estación de maniobras al aplicarse una lista de movimientos partiendo de un estado inicial.

### 4.1.1. Pruebas

Algunas pruebas para tener una idea de cómo debe funcionar `AplicarMovimientos` antes de lanzarnos a codear:

```
P = {Prueba.nuevaPrueba}
E = estado(principal:[a b] uno:nil dos:nil)
{P.iguales {Trenes.aplicarMovimientos E nil} [E]
           'AplicarMovimientos sin movimientos'}
```

```
{P.iguales {Trenes.aplicarMovimientos E [uno(0) dos(0)]} [E E E]
           'AplicarMovimientos con movimientos de 0 vagones'}

{P.iguales {Trenes.aplicarMovimientos E [uno(1) dos(1) uno(~1)]}
           [estado(principal:[a b] uno:nil dos:nil)
            estado(principal:[a] uno:[b] dos:nil)
            estado(principal:nil uno:[b] dos:[a])
            estado(principal:[b] uno:nil dos:[a])]
           'AplicarMovimientos con movimientos de 1 vagon'}

{P.iguales {Trenes.aplicarMovimientos E [uno(1) uno(1) uno(~1) uno(~1)]}
           [estado(principal:[a b] uno:nil dos:nil)
            estado(principal:[a] uno:[b] dos:nil)
            estado(principal:nil uno:[a b] dos:nil)
            estado(principal:[a] uno:[b] dos:nil)
            estado(principal:[a b] uno:nil dos:nil)]
           'AplicarMovimientos: los vagones deben agregarse al comienzo de'#
           ' las vías secundarias y al final de la principal'}
```

### 4.1.2. Implementación

Con el fin de mantener la complejidad del código baja, `AplicarMovimientos` delega parte del trabajo a otras funciones:

```
% Devuelve la lista de los estados que resulta de aplicar los movimientos Ms
% a partir del estado inicial E.
fun {AplicarMovimientos E Ms}
   case Ms of nil then [E]
   [] M|Mr then E1 = {AplicarMovimiento E M} in
      E|{AplicarMovimientos E1 Mr}
   end
end

% Devuelve el estado que resulta de aplicar el movimiento M al estado E.
fun {AplicarMovimiento E M}
   case M
   of uno(N) then E1 = {AplicarMovimientoEnVia N E.principal E.uno} in
      estado(principal:E1.1 uno:E1.2 dos:E.dos)
   [] dos(N) then E1 = {AplicarMovimientoEnVia N E.principal E.dos} in
      estado(principal:E1.1 uno:E.uno dos:E1.2)
   end
end

% Devuelve el resultado de mover N vagones desde el tren en la vía principal
```

```
% Ps hacia el tren en la vía secundaria Ss. El resultado es una tupla de la
% forma <tren en la vía principal>#<tren en la vía secundaria>.
fun {AplicarMovimientoEnVia N Ps Ss}
   if N > 0 then L = {Lista.longitud Ps} - N in
      {Lista.tomar Ps L}#{Lista.agregar {Lista.eliminar Ps L} Ss}
   else
      {Lista.agregar Ps {Lista.tomar Ss ~N}}#{Lista.eliminar Ss ~N}
   end
end
```

`AplicarMovimiento` se encarga de calcular el estado que resulta de aplicar un movimiento dado a un estado inicial haciendo pattern matching con el tipo de movimiento (*uno* o *dos*) y a su vez delega a `AplicarMovimientoEnVia` el trabajo de calcular cuántos vagones quedan en la vía principal y cuántos en la vía secundaria al realizar un movimiento de $N$ vagones, pudiendo ser $N$ positivo o negativo.

## 4.2. Calcular Movimientos

La función `CalcularMovimientos` es el corazón de la aplicación. Es la que se encarga de determinar una secuencia de movimientos capaz de alterar el orden de los vagones de un tren en la vía principal para llevarlos a un orden deseado.

### 4.2.1. Primer Implementación

Una primer implementación de `CalcularMovimientos` resulta relativamente sencilla:

```
% Primer versión de CalcularMovimientos. No comprime.
fun {CalcularMovimientos1 Xs Ys}
   case Xs#Ys
   of nil#nil then nil
   [] (X|Xr)#(X|Yr) then
      % Si el primer vagón ya está en la posición esperada paso al siguiente
      {CalcularMovimientos Xr Yr}
   [] !Xs#(Y|Yr) then
      Particion = {PartirTren Xs Y}
      HaciaUno = {Lista.longitud Particion.2} + 1
      HaciaDos = {Lista.longitud Particion.1}
      Xr = {Lista.agregar Particion.2 Particion.1}
   in
      uno(HaciaUno)
      |dos(HaciaDos)
```

```
      |uno(~HaciaUno)
      |dos(~HaciaDos)
      |{CalcularMovimientos Xr Yr}
   end
end
```

Es la típica solución recursiva con pattern matching, excepto que el matcheo se hace contra dos listas: el estado actual del tren en la vía principal, `Xs`; y el estado al que se quiere llegar, `Ys`. En cada llamada recursiva `CalcularMovimientos` añade tres movimientos más a la solución, los cuales se encargan de posicionar el primer vagón de `Ys`, `Y`, al comienzo de la vía principal. La llamada recursiva se hace, entonces, sobre el resto de los vagones de la secuencia buscada, `Yr`, y el resto de los vagones que quedaron en la vía principal luego del recién acomodado, `Xr`.

Se puede observar que esta solución ya implementa la optimización de que si el primer vagón en la vía principal coincide con el primer vagón de la secuencia buscada no se realiza movimiento alguno para acomodarlo.

También puede observarse que el trabajo de dividir al tren `Xs` en dós en base al vagón `Y` se delega a la función `PartirTren`.

### 4.2.2. Partir Tren

La función `PartirTren` es muy sencilla de implementar gracias a las primitivas de listas implementadas anteriormente.

```
% Devuelve la tupla que resulta de partir el tren Xs por el vagón Y. La tupla
% resultante tiene la forma: <vagones previos a Y>#<vagones posteriores Y>.
fun {PartirTren Xs Y}
   Pos = {Lista.posicion Xs Y}
in
   {Lista.tomar Xs Pos-1}#{Lista.eliminar Xs Pos}
end
```

### 4.2.3. Compresión de Movimientos

La función `ComprimirMovimientos` toma una lsita de movimientos y la "comprime", descartando movimientos de 0 vagones y juntando movimientos consecutivos sobre la misma vía:

```
fun {ComprimirMovimientos Ms}
   Ns = {EliminarMovimientosRepetidos {EliminarMovimientosNulos Ms}}
```

```
in
   if Ns == Ms then Ms else {ComprimirMovimientos Ns} end
end

fun {EliminarMovimientosNulos Ms}
   case Ms of nil then nil
   [] uno(0)|Mr then {EliminarMovimientosNulos Mr}
   [] dos(0)|Mr then {EliminarMovimientosNulos Mr}
   [] M|Mr then M|{EliminarMovimientosNulos Mr}
   end
end

fun {EliminarMovimientosRepetidos Ms}
   case Ms of nil then nil
   [] uno(N)|uno(M)|Mr then uno(N + M)|{EliminarMovimientosRepetidos Mr}
   [] dos(N)|dos(M)|Mr then dos(N + M)|{EliminarMovimientosRepetidos Mr}
   [] M|Mr then M|{EliminarMovimientosRepetidos Mr}
   end
end
```

Como a veces una secuencia de movimientos necesita de varias pasadas de compresión para quedar comprimida al máximo, `ComprimirMovimientos` se llama recursivamente, aplicando una pasada por vez, hasta que la salida de una pasada da igual a su entrada.

Es interesante notar como el pattern matching simplifica gratamente la implementación tanto de `EliminarMovimientosNulos` y `EliminarMovimientosRepetidos`.

### 4.2.4. Minimizar aún más los Movimientos

La última mejora para minimizar los movimientos que sugiere el enunciado es hacer que, en vez de regresar siempre todos los vagones a la vía principal después de cada llamada a `CalcularMovimientos`, mantenerlos en las vías secundarias e ir maniobrando entre las vías uno y dos para llevar los vagones a la principal.

El código de `CalcularMovimientos`, aplicando también la compresión de movimientos, cambia sustancialmente:

```
% Devuelve la lista de movimientos necesarios para transformar el estado
% estado(principal:Xs uno:nil dos:nil) en
% estado(principal:Ys uno:nil dos:nil)
fun {CalcularMovimientos Xs Ys}
   case Ys of nil then nil
   [] Y|Yr then
```

```
      Particion = {PartirTren Xs Y}
      HaciaUno = {Lista.longitud Particion.2} + 1
      HaciaDos = {Lista.longitud Particion.1}
      % Los primeros tres movimientos dejan Y en la vía principal y el resto
      % de los vagones en las vías uno y dos.
      Ms = uno(HaciaUno)
           |dos(HaciaDos)
           |uno(~1)
           |{CalcularMovimientosUnoYDos Particion.2 Particion.1 Yr}
   in
      {ComprimirMovimientos Ms}
   end
end

fun {CalcularMovimientosUnoYDos Us Ds Ys}
   case Ys of nil then nil
   [] Y|Yr then Particion L U1s D1s in
      if {Lista.miembro Us Y} then
         Particion = {PartirTren Us Y}
         L = {Lista.longitud Particion.1}
         U1s = Particion.2
         D1s = {Lista.agregar Particion.1 Ds}
         uno(~L)|dos(L)|uno(~1)|{CalcularMovimientosUnoYDos U1s D1s Yr}
      elseif {Lista.miembro Ds Y} then
         Particion = {PartirTren Ds Y}
         L = {Lista.longitud Particion.1}
         U1s = {Lista.agregar Particion.1 Us}
         D1s = Particion.2
         dos(~L)|uno(L)|dos(~1)|{CalcularMovimientosUnoYDos U1s D1s Yr}
      end
   end
end
```

`CalcularMovimientos` calcula los primeros tres movimientos para llevar los vagones a las vías uno y dos y el primer vagon ordenado a la vía principal. Delega el resto del trabajo a `CalcularMovimientosUnoYDos` y finalmente comprime los resultados.

`CalcularMovimientosUnoYDos`, al igual que la primer implementación de `CalcularMovimientos` (4.2.1) calcula los movimientos para colocar un vagón ordenado de `Ys` en la vía principal en cada llamada recursiva, pero en vez de mantener el resto de los vagones en la vía principal los mantiene en las vías uno y dos (listas `Us` y `Ds`).

A simple vista puede parecer que esta implementación podría incurrir en varios movimientos innecesarios. Por ejemplo, no implementa la optimización básica de no calcular movimientos si el vagón a procesar `Y` ya se encuentra en la posición correcta. Pero resulta que esa

optimización es innecesaria si se aplica la compresión de movimientos.

### 4.3. Maniobrar

Con la ayuda de las funciones anteriormente descriptas, la implmentación de la función principal del programa, `Maniobrar`, resulta trivial:

```
% Devuelve una lista de estados por los que tiene que pasar la playa de
% maniobras para pasar del estado
% estado(principal:Xs uno:nil dos:nil) a
% estado(principal:Ys uno:nil dos:nil)
fun {Maniobrar Xs Ys}
   Ms = {CalcularMovimientos Xs Ys}
   E = estado(principal:Xs uno:nil dos:nil)
in
   {AplicarMovimientos E Ms}
end
```

## 5. Visualizador

### 5.1. Modificaciones

Se realizazon las modificaciones mínimas al código del visualizador de Peter Van Roy para adaptarlo al diseño del trabajo práctico. Estas modificaciones fueron:

- Se metió el código dentro de un functor que sólo exporta el procedimiento `Visualizar`.
- Se adaptaron los nombres de las estructuras al castellano. Esto es, cambiar “main” por “principal”, “one” por “uno”, etc.
- Se cambió el nombre del procedimiento principal “Visualize” a “Visualizar”.
- Se cambió el nombre del título de la pantalla visualizada a “Visualizador de Playa de Maniobras”.

Es resto del visualizador se dejó intacto, es decir, que la implementación sigue estando en inglés y sigue utilizando las funciones estándard de Oz para el manejo de listas (e.g. `Length`,

`Nth`, `List.forAllInd`) por más de que ello no sea acorde con el resto de la implementación del trabajo práctico. Se decidió dejar eso así porque la interfaz del módulo, después de los cambios hechos ya resulta acorde al resto del trabajo.

# 6. Corridas de prueba

## 6.1. Desde la Linea de Comandos

El functor ejecutable de la aplicación, `TrenesPrincipal` está definido como:

```
functor
import
   Trenes
   Visualizador
define
   Inicial = [a b c d e f g]
   Final = [c f a e g b d]
   {Visualizador.visualizar {Trenes.maniobrar Inicial Final}}
end
```

Y puede ejecutarse con:

```
$ ozmake
$ ozengine TrenesPrincipal.ozf
```

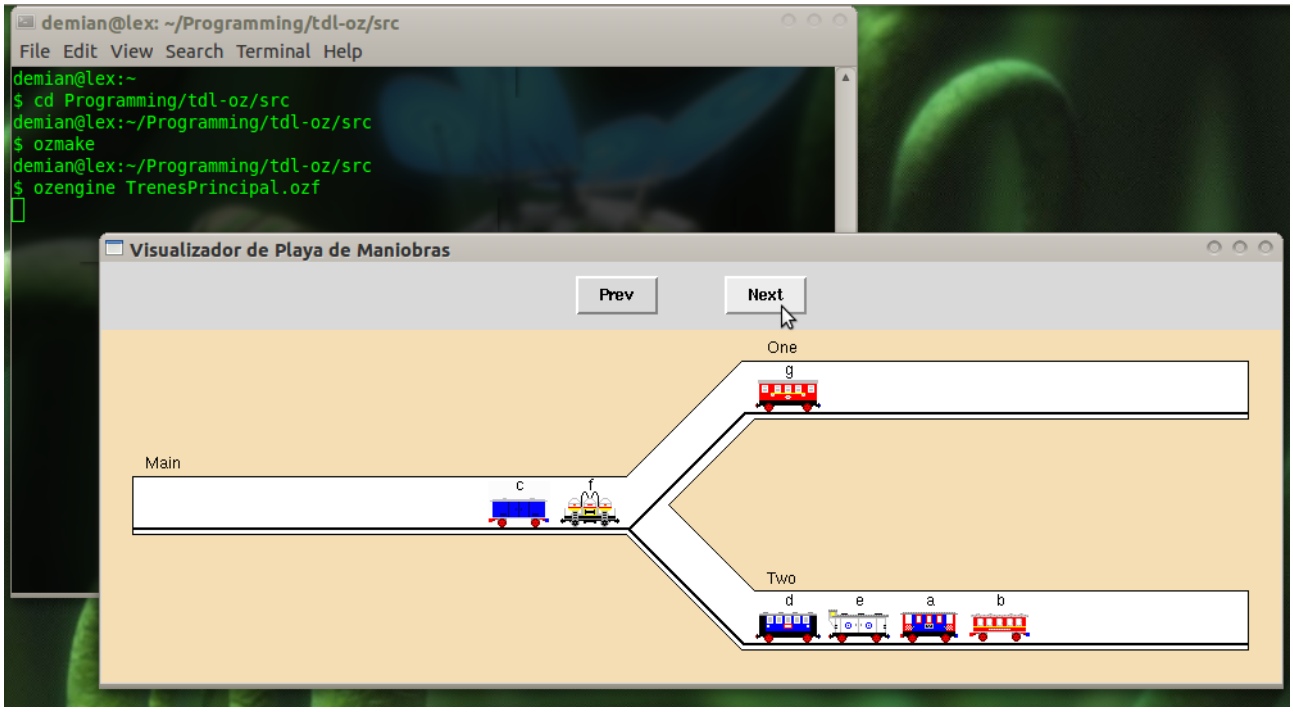


Figura 1: Corrida de prueba desde la linea de comandos.

## 6.2. Desde Emacs

También se puede probar la aplicación desde Emacs. Para ejecutar código en un módulo basta con cargarlo en el ambiente usando la función `Module.link`. Una vez hecho eso, se puede programar interactivamente en Emacs y usar, por ejemplo, el browser de Oz.

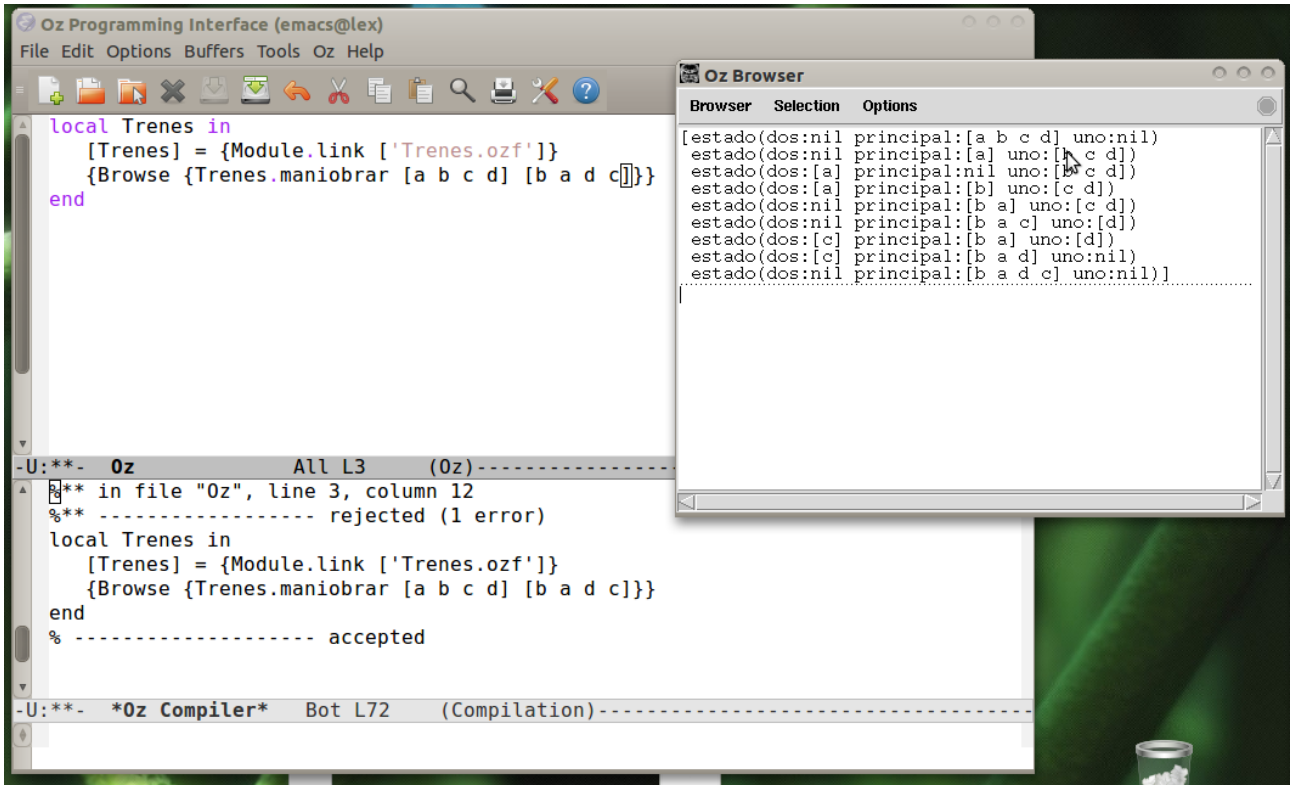


Figura 2: Corrida de prueba desde Emacs usando el procedimiento `Browse`.