UNR - FCEIA

LCC - Análisis de Lenguajes de Programación

Trabajo Práctico Final: Funciones Recursivas de Lista

Vallejos Vigier, Franco
V-2952/1

19 de Diciembre, 2022

Profesora:
Manzino, Cecilia

Introducción

Mi motivación para este último trabajo práctico fue implementar un DSL que sea de utilidad en el ámbito de los Lenguajes Formales más precisamente en la Funciones Recursivas de Listas. Lo que busco con este DSL es facilitar el uso de dichas funciones usando de lenguaje base Haskell, y así brindar esta herramienta a los alumnos de segundo año, recordando que en dicha materia muchas veces veíamos las funciones en papel y era difícil predecir su comportamiento cuando se tornaban más complejas. Si bien el dominio escogido puede considerarse humilde logramos estructurar un DSL robusto con un fuerte control de errores, todo esto gracias a la mónada escogida para este trabajo que será presentada más adelante.

Índice general

Capítulo 1

Explicación del Programa

1.1. Modulos del programa

En este TP Final quise respetar la misma forma de trabajar que venía acarreando en la materia, obteniendo un intérprete modular teniendo bien separadas las funcionalidades. Presentamos la siguiente estructura del proyecto:

```
.
├── scr
│   ├── Errors.hs
│   ├── Eval.hs
│   ├── FRLMonad.hs
│   ├── Global.hs
│   ├── Lang.hs
│   ├── Parse.y
│   └── PPrinterFRL.hs
├── app
│   └── Main.hs
├── EjemplosBuenosLoadFile
│   ├── AplicacionesEnAplicaciones.frl
│   ├── Cascada.frl
│   ├── OrdenDeReduccion.frl
│   └── Reescritura.frl
├── EjemplosErroresLoadFile
│   ├── ErrorComandoNoDeclarativo.frl
│   ├── ErrorEmptyFile.frl
│   ├── ErrorEval.frl
│   ├── ErrorNoComando.frl
│   ├── ErrorParseoSimple.frl
│   └── ErrorVarInexistente.frl
└── EjemploSintaxis
    └── Sintaxis del Interprete.ttx
```

1.1.1 Errors.hs

Básicamente en este módulo implementamos la estructura de errores que va a tener nuestro programa. En el proyecto un error va a estar definido por un constructor y una string que es su descripción. Al hacer un DSL robusto en errores es mejor que no haya una relación de 1 a 1 entre constructores y errores, sino que directamente teniendo la string de descripción del error y que esta sea verdaderamente significativa la podemos capturar con la mónada y devolverla en top level.

1.1.2 Global.hs

Aquí describimos la estructura que va a usar como estado nuestra mónada, el estado global de nuestras ejecuciones de comando. Simplemente es una estructura de record que contiene un array de declaraciones de variables. Además como en nuestro programa las declaraciones pueden ser de Listas como de Funciones aplicadas a estas, se puede decir que en el estado global conviven ambos entornos sin problemas de nombres. Por esta convivencia declaro dos funciones observadoras que me retoran el .^entorno"de las Listas y el .^entorno"de las Funciones

1.1.3 Lang.hs

En este módulo declaramos los data a usarse tanto en comandos, declaraciones y operaciones

```
Comm ::= Exit | Def Decl | PrintVar Name
         | PrintFoo Name | Eval Exp | ErrorCommando String
         |LoadFile Path | Flush | Peek
```

Aquí le damos forma a la estructura de los comandos que puede analizar nuestro interprete. Sin entrar a explicar cada uno en detalle podemos procesar el comando de salida, una declaración, podemos buscar una Lista/Función, evaluar una expresión, arrastrar un error del happy, cargar un archivo, borrar el entorno y examinarlo

```
Decl ::= DeclVarLista { declName :: Name, declBody :: Exp }
       | DeclVarFoo { declName :: Name, declbody :: [Ops] }
```

En esta declaración en forma de record establecemos las declaraciones de las variables que podemos crear/declarar. Declaramos Listas y Funciones, guardando en el record correspondiente su nombre y su cuerpo declarativo que en el caso de las funciones no son más que una lista de funciones a aplicar. En contracara las Listas en su cuerpo son unas expresion a evaluar.

```
Exp ::= ListaBase [Integer] | Aplication [Ops] Exp
      | VarEnvList Name
```

Las Expresiones anteriormente mencionadas, se definen recursivamente como una lista de enteros, una aplicacion de operaciones a una lista de enteros o una Lista guardada en el entorno. En otras palabras siempre una expresion en última instancia es una lista de enteros.

```
type Name = String
type Path = String
```

Aqui renombramos por cuestiones de comodidad tipos ya existentes para darles un nombre más significativo. Los nombres son bastantes lineales, pero no pueden contener números tienen que ser totalmente alfabéticos

```
Ops ::= Ol
    |   Or --Zero a Der
    |   Sl --Sucesor a Izq
    |   Sr --Sucesor a Der
    |   Dl --Eliminar a Izq
    |   Dr -- Eliminar a Der
    |   Ml -- Mover a Izq
    |   Mr --Mover a Der
    |   DDl --Duplicar a Izq
    |   DDr --Duplicar a Der
```

```
|   Swap --Intercambiar extremos
|   Rep [Ops] --Repeticion
|   VarEnvFoo Name --Buscar una foo en el env
```

La data que usamos para las Operaciónes, simplemente son Constructores que vamos a usar en la evaluación para aplicar la operación sobre la lista correspondiente

1.1.4 Parse.y

Aqui en el Happy definimos la siguiente gramática

```
Comm ::=  Defexp | "Flush" | "Peek" | "LookFoo" Name
     | "LookVar" Name | "Quit" |"LoadFile" "./"Path | Exp ";"
Defexp ::= "DefVar" Name "=" Exp | "DefFoo" Name "=" OpsLista
Path ::= Name"/"Path | Name"."Name
Exp ::= Lista | OpsLista "$" Exp | Name
OpsLista ::= Ops | Ops OpsLista
Ops ::= "Oi"|"Od"|"Si"|"Sd"|"Di"|"Dd"|"Mi"|"Md"|"DDi"
        |"DDd"|"Swap"|"<"OpsLista">"|Name

Name ::= String
Lista ::= [Integer]
```

Cabe destacar que que el path es recursivo y siempre es el absoluto desde la carpeta del trabajo final y no me importa hacer un checkeo aquí de la extensión solo exigo que el path termine en un archivo con alguna extensión luego en el main controlaré si ese sufijo es el ".frl". Otra cosa a destacar es que la lista de Operaciones es basicamente usa seguidilla de operaciones separadas por un espacio.

Luego en el happy hacemos el manejos de errores de entrada haciendo diferencia entre ellos gracias a los token de error. Usando la data E como recomendaba happy en su página (https://www.haskell.org/happy/doc/html/sec-monads.html)

```
data E a = Ok a | Failed String

thenE :: E a -> (a -> E b) -> E b
m `thenE` k =  case m of
                  Ok a -> k a
                  Failed e -> Failed e

returnE :: a -> E a
returnE a = Ok a

failE :: String -> E a
failE err = Failed err

catchE :: E a -> (String -> E a) -> E a
catchE m k =  case m of
                 Ok a -> Ok a
                 Failed e -> k e

happyError :: [Token] -> E a
happyError tokens | elem TNullVar tokens = failE "\nError de parseo
nombre de variable  null o palabras reservada missmatch.
Revisar gramatica\n"
                  | elem TParserError tokens = failE "\nError
en de Parseo, caracter no reconocido. Revisar LEXER\n"
                  | elem TErrorNumSuelto tokens = failE "\nError
de Parseo, número por fuera de [ ].\n"
                  | elem TBadNumberList tokens = failE "\nError de
Parseo, lista integrada por caracter NO numerico.\n"
                  | otherwise = failE "\nError de parseo\n"
```

Como indica en su web happy, para hacer el manejo de errores tenemos que implementar una función que sea un handler de los Token de Error que vamos a lanzar en determinados momentos en el lexer y ese Token

de error va a encapsular el error en la estructura data E a, que a grandes rasgos es similar a un Maybe

1.1.5 FRLMonad.hs

En la definición de la mónada me valí de lo aprendido este año en compiladores con Mauro, ya que en dicho proyecto usamos definiciones de instancias y contextos flexibles para definir la MonadFD4 abusando del hecho de obviar la definición explícita de las operaciones Bind y Return de la misma. Con esta flexibilidad solo nos basta decir que nuestra mónada es una instancia de mónadas ya dadas por el lenguaje como son la MonadIO, MonadState, MonadError. Con esta definición flexible digo que mi mónada FRL es instancia de todas las anterioress citadas expecificando el tipo del estado usado(Que va a ser el definido en el módulo Global.hs) y la estructura el errores que es el definido en el módulo Errors.hs. Abusando un poco de esta definición, al ser una instancia de las anteriores mónadas ya contamos de antemano con primitivas para construir nuevas(están bien explicadas en archivo FRLMonad.hs). Dados los comandos del DSL es primordial que la mónada exporte operaciones como failFRL que tira unn error en forma de string, catchErrors que catchea los errores de las operaciones que se le componen y los muestra (primordial si queremos hacer un DSL robusto), tenemos que poner adulterar el entorno con las funciones addOp y addVar, podemos observar el entorno con los pares de funciones lookup, tenemos que poder mostrar mensajes con printFRL asi como ser capaces de borrar el entorno a demanda con eraseEnv. Todas estas funciones y me parecen de suma importancia destacarlas porque en mi trabajo apunto a que la mónada tenga control total de los errores y de las etapas runFRL de la ejecución y que nada pase por fuera de ella así podemos capturar los errores.

1.1.6 PPrinterFRL.hs

Sin mucho que aclarar en esta sección, buscamos hacer un pprinter en cascada de nuestros datas declarados en el modulo Lang.hs, sin que quede

ningún data sin representar en el mismo. El inconveniente es que en pocos casos el pprint de comandos no tiene mucha diferencias con lo ingresado por teclado

1.1.7 Eval.hs

En el evaluador monádico vamos a empezar a explicar de arriba para bajo. Empezamos con la evaluacion de los comandos por la estructuración de los mismos no hay una salida unificada para todos, por lo tanto comandos como Flus Peek Exit LoadFile van a ser resueltos a nivel de Main luego del parseo matcheo del comando. Mientras que los comandos de declaración y evaluación de variables Listas, Funciones, Expresiones pueden ser resueltos en una misma funcion (evalCommand) teniendo en cuenta que en las declaraciones de Listas hacemos callbyValue, en las declaraciones ambas no podemos definir una Lista o Funcion con una variable inexistente en el entorno. En los comandos de busqueda y devolución de variables tienen un evaluador propio siempre controlando que la variable exista en el entorno. En la evaluación de las Exp simplemente tengo 3 posibilidades, devuelvo la lista, la evaluación de una lista de operaciones en una lista o una lista en el entorno. Si queremos evaluar una lista de operaciones en una lista, hacemos una evaluación recursiva de la misma aplicando paso a paso la operación.

1.1.7.1 Orden de reducción

Basicamente el orden de escritura de la aplicación es el inverso en la evaluación, es decir sea

```
Op1 Op2 Op3 $ xs;
```

El orden de aplicación de las funciones a la lista xs es

```
Op1(Op2(Op3 xs))
```

1.1.8 Main.hs

Aquí en el main vamos runear nuestra mónadaFRL con runOrFail() para que pueda tirarse abajo con el failFRL() en caso de Quit por ejemplo. Y aplicamos el runFRL al runInputT para así con la foo getinputs capturar línea a línea los input atrapados por consola. En getinputs, como parseamos los comandos a nivel de happy directamente, a la linea capturada la analizamos con parse y nos va a dar un comando parseado.

- ErrorCommand err: Si el happy en caso de absorber un token de error, vamos a capturar esa string significativa, la pasamos por su pprinter y la mostramos con la mónada
- Eval: Una vez parseado el comando y sabiendo que es un comando de evaluación entonces llamamos a evalCommand, teniendo resguardo de que tenemos que catchear los errores de este. Y como el catch encapsula en una monad maybe tenemos que analizar el resultado que esta en un Maybe Maybe [Integer], un maybe por el catchErrors y otro por el evalCommand. Mostramos el resultado por pantalla con el pp correspondiente y por medio de la mónada FRL
- Def: Si es un comando de definicón, de forma análoga al comando anterior catcheo el error de la evaluación del comando. Terminando igual que el comando anterior
- PrintVar, PrintFoo: Si es un comando de búsqueda en el entorno, sabemos que esta búsqueda puede fallar (en caso de inexistencia). Por eso catcheamos el error si es que se da y de caso afirmativo mostramos la Lista/Función
- Exit: Si queremos salir del interprete directamente derivamos en el printFRL de una cadena de despedida y tiramos abajo la mónada
- LoadFile : Para cargar un archivo por medio de path global, verificamos su extensión(sino devuelvo su error corespondiente). En caso

de poder abrirlo colapso sus lineas en una lista Strings. Para luego mapear el parse del contenido, parseando linea a linea, así luego mapear el evaluador de Comandos

- Flush: Se pisa el entorno con el initialEnv del Global.hs
- Peek: Traemos el entorno por pantalla

1.2. Operaciones FRL al detalle

Vamos a hacer una descripción de las operaciones aceptadas por el interprete en cuanto a las FRL

Operacion	**Definición**
$ZeroLeft : Oi$ xs	$(0 : xs)$
$ZeroRigth : Od$ xs	$(xs : [0])$
$DeleteLeft : Di$ (x:xs)	xs
$DeleteRigth : Dd$ (xs++[x])	xs
$SucesorLeft : Si$ (x:xs)	$(x + 1 : xs)$
$SucesorRigth : Sd$ (xs++[x])	$xs + +[x + 1]$
$LeftShift : Mi$(xs++[x])	$(x : xs)$
$RigthShift : Md$(x:xs)	$(xs + +[x])$
$LeftDuplicate : DDi$(x:xs)	$(x : x : xs)$
$RigthDuplicate : DDd$(xs++[x])	$(xs + +[x, x])$
$Swap : Swap$([x]++xs++[y])	$([y] + +xs + +[x])$
$RepetitionOperator : Rep\ f\ ([x] + +xs + +[x])$	$[x] + +xs + +[x]$
$RepetitionOperator : Rep\ f\ ([x] + +xs + +[y])$	$Rep\ f\ (f([x] + +xs + +[y]))$
$RepetitionOperator : Rep\ f\ ([x])$	$[x]$

1.3. Robustez en la lectura de Archivos .frl

En la última consulta, no tenía la gestión de archivos siquiera empezada. Por eso quería explicar detalles de como pude realizar un control de

errores sobre los archivos de entrada de tal forma si se encuentra un error se puede trackear cuál es y en qué linea está. Vamos a ir parte a parte de lo que pasa cuando cargamos un archivo, recordemos que si queremos cargar un archivo nuevo el entorno es persistente por lo tanto si queremos evitar colision en el sentido de actualizaciones de variables inesperadas es recomendable hacer un Flush. Definiendo entorno persistente como aquel que no se borra cada vez que cargamos un nuevo archivo, sino que se actualiza. Cabe destacar que el archivo no puede tener líneas vacías, es decir que dos declaraciones consecutivas no pueden estar separadass por una linea vacía entre ellas

```
if ".frl" `isSuffixOf` path then
  do
   lf <-  liftIO $ catchError (do
                               lf'<- lines <$> readFile path
                               return (Just lf'))
           (\err  -> do
                     let pperror = ppError ("Error al abrir el File "
                                            ++show (err))
                     putStrLn $ pperror--Printeo el error acarreado
                     return Nothing)
```

En esta primera parte de la evaluación del LoadFile Path, simplemente vamos a controlar la finalizacion o extension del path que tiene que ser .frl. De caso que sea .frl entonces lo intentamos leer a la vez que colapsamos todo su contenido en una Lista de String (Línea a Línea), pero como sabemos que la lectura del archivo puede acarrear errores por ejemplo un path inexistente... lo encapsulamos en un catchError que en el caso de recibir un error va a informar el error que va a presentar readFile

```
    case lf of
        Nothing -> loop
        Just [] -> do
                     let pperror = ppError ("Error: Archivo Vacio" )
                     (lift $ (printFRL pperror))>> loop
```

En esta segunda parte básicamente analizo el retorno de la lectura del archivo. En el caso de Nothing entonces hubo un error que ya fue catcheado por catchError anterior y si es Just [] implica que el archivo es vacio por lo que me tome la libertad de verlo como un Error

```
Just lfcontenido -> do
   let ppPrologo=ppError("Cargando el contenido del archivo: "++path)
   (lift $ (printFRL(ppPrologo)))
   let listaDeComandos = map parse lfcontenido
   contenidoCheckParse<-  (lift$ catchErrorsWithPPError$
                          controlParsingFile listaDeComandos 1 path)
   case contenidoCheckParse of
      Nothing -> loop
      Just contenidoOKParse ->(lift$ catchErrorsWithPPError$
                                 controlEvaluationFile
                                 contenidoOKParse 1 path)  >> loop
```

Como 3ra opción tenemos que es un archivo con contenido por lo tanto voy a aplicarle a cada linea del archivo la funcion parse (parseo todas las lineas), obteniendo una lista de comandos Comm.Luego filtro esa lista obtenida (controlParsingFile()), tirando errores si me encuentro con algo que no es una DEF DECL (recordemos que el archivo solo sirve para importar declaraciones nuevas) en el caso de encontrar otro tipo de comando tiro una descripción del error con el número de línea donde lo encontré (facilmente se sabe el numero porque estamos en un lista de String) Una vez hayamos pasado el check del Parseo, es decir que solo haya en el archivo comandos declarativos nada nos impide que esos comandos declarativos aunque esten bien formulados no tengan errores de evaluación... Por eso controlEvaluationFile(), va a seguir trabajando con la lista de comandos que dejo el anterior filtro y esta vez nos tenemos que olvidar de los demás comandos, ya sabemos que solo tengo declaraciones. Por lo tanto controlEvaluationFile() va a evaluar solamente las declaraciones para así si en la evaluación salta algún error lo vamos a poder atrapar con (catchE-

rrorsWithPPError$controlEvaluationFile). Lo mágico de esto es que los errores que tira el primer filtro (controlParsingFile()) se meten dentro de los errores que puede tirar el segundo (controlEvaluationFile()). Por lo si tenemos un archivo que parsea bien pero que tiene errores de evaluacion en alguna de sus lineas, NO se va a actualizar el entorno de la mónadaFRL hasta que la evaluación del archivo sea correcta totalmente. Esto esta bueno porque si tenemos un error en la linea N no queremos que las N-1 lineas anteriores nos modifiquen el entorno porque quedariamos en un entorno inconsistente con respecto a la totalidad del archivo

```
else  --Error de extension no .frl
    do
      let pperror =  ppError "Error en la extension del file tiene
      (lift $ (printFRL pperror))>> loop
```

En esta última parte directamente sabemos que el path no termina con ".frl"por lo tanto no lo podemos leer

Capítulo 2

Otras Cuestiones

2.1. Instalación y Uso del Programa

Para Instalar el DSL FRL debemos correr:

```
...:$stack setup
...:$stack build
```

Luego para ejecutar el programa y correrlo

```
...:$stack exec VigierFRL-exe
```

Esto hace que se abra el interprete que parsea línea a línea lo ingresado en comandos y lo procesa. Y estos comandos van a ser válidos siempre y cuando respeten la gramática descripta anteriormente, sino vamos a devolver un error

2.2. Mejoras...

Sinceramente estoy bastante satisfecho con el trabajo realizado. Pero algunos puntos a mejorar son:

- Rectificar el orden de aplicacion de las operaciones, que coincidan con el orden ingresado y no al revés
- Lograr pasar por linea de comandos el archivo así inicia el interprete con el entorno cargado. Ej

  ```
  ...$stack exec VigierFRL-exe ./Ejemplos/test.frl
  ```

- Poder compilar archivos .frl, si bien podemos cargarlos pero son meramente declaraciones, estaría bueno poder compilarlos y agregar efectos al lenguaje para que sea más cómodo y no manejar todo desde un interprete interactivo

- Si bien el pprinter quedó de un tamaño considerable, lo mejor sería darle un retoque y ver en qué comandos se podría pprintear mejor
- Hacer un pprinter de files .frl sin duda
- Añadir más FRL, en las slides de Pablo Verdes hay más funciones como la potencia o proyecciones con lo que su implementación ayudaría mucho porque expandemos la frontera a las FR.

2.3. Bibliografía

- Apuntes teóricos de la materia de Lenguajes Formales
- Happy: https://www.haskell.org/happy/doc/html/sec-monads.html
- Trabajo de Compiladores 2022.
- Tps de la cátedra más que nada para orientar el eval y el pprinter

Capítulo 3

Testing

3.1. Casos de Prueba y Archivos de prueba

Brindamos, en el directorio del proyecto, archivos .frl verificados para poder testear la funcionalidad del intérprete como su detección de errores en los mismos.

- Empezando con los archivos de error vamos a poder encontrar ejemplos de numerosos tipos de error por ejemplo si metemos contenido no parseable por el happy hasta errores que dependen de la semántica de la declaraciones y no de la sintaxis como lo es una declaración con errores de evaluación en su cuerpo o con una variable inexistente en el entorno.

- Con los EjemplosBuenosLoadFile directamente intentamos visibilizar las numerosas formas de escribir declaraciones que el interprete provee, además de anidamiento de operaciones y la posibilidad de reescribir variables en el entorno utilizando su valor hasta el momento. Además brindamos un ejemplo sobre la asociatividad del operador $ que denota la aplicación de una lista de operaciones a su izq a una lista a su derecha

- También brindamos un .txt con ejemplos y una breve explicación de los comandos posibles en el intérprete. Dicho archivo es

  `Sintaxis del Interprete.txt`

  De último momento se pudo adicionar el operador Rep f de FRL, por lo cual proporciono un archivo OpRepeticion.frl dónde defino operaciones FRL derivadas del Rep como son MoverAIzq/Der, Duplicar y Intercambiar Extremos. Si bien estas operaciones ya eran contempladas por el evaluador son resueltas en el mismo de forma directa por temas de eficiencia para que no se dependa de la evaluación del Rep. Pero en el archivo se encuentra sus versiones basadas en Rep totalmente funcionales.