Universidad Nacional de Rosario

Facultad de ciencias exactas, ingeniería y agrimensura

Análisis de Lenguajes de Programación

Trabajo práctico Final

Alumno:
Petruskevicius Ignacio

Marzo 2022

ALP - Informe TP Final

Petruskevicius Ignacio, LCC UNR FCEIA 02/03/2022

1. Introducción

Ante la propuesta del trabajo práctico y teniendo en cuenta que el tema a escoger debía ser coherente, comencé a pensar posibilidades. Entre las ideas que surgieron una fue tener en cuenta algún tema que se haya dado en la carrera y hubiera estado bueno poder complementarlo con una herramienta. Remontándome a el primer cuatrimestre de 2^{do} año, específicamente la materia de Lenguajes Formales y Computabilidad, encontré el tópico Funciones Recursivas de Listas. Repasando, recordé que en la práctica se pedía confeccionar funciones que cumplan con especificaciones dadas, pero las cuales eran difíciles de corroborar dada su complejidad. Por lo tanto una herramienta que permita realizar tests de manera mas simple podría ayudar al estudiante.

2. Nociones teóricas sobre FRL

Definición: Una lista es una secuencia ordenada y finita de cero o más elementos de $\mathbb{N}_0$.

Definición: Las funciones de listas son funciones que van del conjunto de las listas en el conjunto de las listas.

Sobre las listas, se definen funciones base de las cuales surgen otras aplicando la composición de funciones. Estas son:

- Cero a izquierda: $O_i[x_1, x_2, ..., x_k] = [0, x_1, x_2, ..., x_k]$
- Cero a derecha: $O_d[x_1, x_2, ..., x_k] = [x_1, x_2, ..., x_k, 0]$
- Borrar a izquierda: $D_i[x_1, x_2, ..., x_k] = [x_2, x_3, ..., x_k]$
- Borrar a derecha: $D_d[x_1, x_2, ..., x_k] = [x_1, x_2, ..., x_k 1]$
- Sucesor a izquierda: $S_i[x_1, x_2, ..., x_k] = [x_1 + 1, x_2, ..., x_k]$
- Sucesor a derecha: $S_d[x_1, x_2, ..., x_k] = [x_1, x_2, ..., x_k + 1]$
- Operador repetición: <F>= $\begin{cases} [x, Y, z] & si \quad x == z \\ F[x, Y, z] & si \quad x \neq z \end{cases}$

Además de esas funciones base, nos son de interes las siguientes funciones derivadas:

- Mover a izquierda: $M_i[x_1, x_2, ..., x_k] = [x_k, x_1, x_2, ..., x_{k-1}]$
- Mover a derecha: $M_d[x_1, x_2, ..., x_k] = [x_2, ..., x_k, x_1]$
- Duplicar a izquierda: $DD_i[x_1, x_2, ..., x_k] = [x_1, x_1, x_2, x_3, ..., x_k]$
- Duplicar a derecha: $DD_d[x_1, x_2, ..., x_k] = [x_1, x_2, ..., x_k, x_k]$
- Intercambiar extremos: $INT[x_1, x_2, ..., x_k] = [x_k, x_2, ..., x_1]$

Notese que la composición natural de funciones es de la siguiente forma: $F \circ G[x] = F(G[x])$, pero esto dificulta la escritura puesto que tenemos que leer de izquierda a derecha la "ejecución" de funciones. Por lo tanto se redefine la composición: $F \circ G[x] = G(F[x])$.

3. Descripción

Se desarrolló un **EDSL** el cual permite construir dos tipos de elementos, funciones y listas (a veces derivadas de la aplicación de funciones (primitivas o no)). Estas listas al ser declaradas, en caso de estar formadas por la aplicación de funciones, se las evalúa y se almacena el resultado. Por lo tanto luego podremos observar el estado final de las listas con un comando.

La entrada puede ser por consola o por un archivo de texto con la sintaxis que luego veremos. Se implementaron las funciones atómicas de las **FRL** y además algunas derivadas como mover izquierda, duplicar o intercambiar, las cuales son comúnmente utilizadas.

4. Instalación y uso

Para dejar listo para usar el **EDSL** se deben ejecutar los siguientes comandos:

```
$ stack setup
$ stack build
```

Luego se puede ejecutar de dos maneras dependiendo de el modo de uso que se le quiera dar. Para utilizarlo como consola:

```
$ stack exec FRL-exe
```

Para leer un archivo con extensión .frl:

```
$ stack exec FRL-exe "path/file.frl"
```

Cada linea ingresada ya sea por consola o en el archivos es parseada como un comando. Los comandos validos son los siguientes:

Comando	Función
Fun var = decl	Declaración de una función con el nombre dado
Var var = decl	Declaración de una variable que almacena una lista
Print var	Imprime en pantalla el valor asociado a una función o variable con el nombre dado
Exit	Termina la ejecución.

La decl (declaración) se construye utilizando las siguientes funciones base definidas y además concatenando nombres de variables o funciones.

	Función
Si	Sucesor izquierda
Sd	Sucesor derecha
Oi	Cero a izquierda
Od	Cero a derecha
Di	Eliminar izquierda
Dd	Eliminar derecha
Mi	Mover izquierda
Md	Mover derecha
DDi	Duplicar izquierda
DDd	Duplicar derecha
INT	Intercambiar extremos
<F>	Operador repetición

Ejemplos de declaraciones de variables y funciones:

```
Var nombre = Si Si [1,2]
Var lista = [1,2]
Fun funcion1 = Si Si
Var nombre2 = funcion1 lista
```

Tenga en cuenta que cada utilización de función o variable está ligada a que tenga sentido, es decir, si se ingresa un nombre de variable en la ultima posición de una declaración de función, esta debe ser una función.

Para evitar errores de parseo escribir los operadores separados por un espacio, ejemplo Oi Si Od.

4.1. Aclaraciones

En modo lectura de archivos no se puede imprimir, ya que al final de la ejecución se muestra el estado final de cada variable y función declarada.

Existe un espacio de nombres distinto para las funciones y para las variables, es decir que una función puede llevar el mismo nombre que una variable. Por eso el comando **Print** puede imprimir una función y una variable a la vez. Por otro lado, se infiere el tipo de una variable escrita en una declaración mediante la posición y utilización de la misma.

Ejemplos de comandos:

```
$ Var a = [1,2,3]
$ Print a
[1,2,3]
$ Fun hola = Oi Si
$ Print hola
Oi Si
$ Var b = hola Oi a
$ Print b
[0, 1, 1, 2, 3]
```

Se adjuntan más ejemplos en la carpeta **test**.

5. Implementación

5.1. Estructura

El código relevante se encuentra en **app/main.hs**, **src/eval.hs**, **src/parse.y**, **src/common.hs**, **src/PPrinter.hs** y **src/monads.hs**.

En el primero se encuentra la lógica de ejecución de la interfaz por consola y la lectura de archivos si se pasa este como argumento. En **src/common.hs** se definen todos los tipos de datos utilizados, en particular **Lista** el cual es el AST que representa tanto a las listas como a las funciones de lista. **src/parser.y** es el archivo de Happy, herramienta utilizado para parsear la entrada. En **src/eval.hs** se realiza la evaluación de un comando, de la cual se hablará mas adelante. Y por ultimo, en **src/PrettierPrinter.hs** se definen funciones que permiten mostrar por pantalla de manera intuitiva las variables y funciones definidas.

5.2. Parseo

Como se comentó, se utilizo Happy para crear un parser para comandos. Para hacer esto se define la gramática del lenguaje aceptado además de una función encargada de tokenizar la entrada. Me gustaría hablar un poco de esta ultima ya que como se vio en el apartado de Nociones teóricas, la aplicación de funciones se realiza de izquierda a derecha. Lo cual complica la creación del AST que luego será evaluado, ya que si construimos el árbol a medida que se lee la entrada, la función de más a la izquierda quedará mas arriba en el AST y por lo tanto se aplicará ultima, contrario a lo que se quiere. Por lo tanto para solucionar este inconveniente decidí invertir el orden de la lista de tokens leída antes de que se ejecute el parser, entonces el árbol se construirá de la forma querida. Esta inversión no se puede realizar sobre el completo de los tokens, ya que esto puede generar cadenas invalidas según la gramática dada. Por lo tanto se tienen en cuenta 2 escenarios, cuando se declaran funciones o cuando se declaran variables. En el primer caso, podemos dar vuelta todo menos los <>del operador repetición. En cambio en el caso de variables sabemos que al final se debe encontrar una lista, ya sea explicita ($[1, 2, 3]$) o una variable, la cual debe preservar su posición. Entonces se evalúa cada caso y se invierte la posición según corresponda.

5.3. Tipos de datos

Dados los comandos que se implementaron, a su vez se declaró un tipo de datos para representar cada uno. Y también un tipo de datos para representar una lista. Esta puede ser directamente una lista de naturales o una secuencia de aplicaciones de distintos operadores unarios. Por lo tanto el árbol que representa a la lista resulta ser parecido a una lista simplemente enlazada. Solo el operador repetición toma 2 argumentos, una función que toma una lista y devuelve otra, la cual se representa utilizando las funciones de Haskell.

Por otro lado se definieron varios tipos de datos para representar errores tanto de evaluación como parseo.

5.4. Evaluación

La idea general del **EDSL** planteado es declarar funciones y variables que pueden ser utilizadas múltiples veces para definir las **FRL** que se quiera. Por lo tanto es importante contar con un entorno en el cual estas declaraciones de almacenen y se puedan consultar. Por otro lado, dadas las definiciones de los

operadores vistas en la sección de noción teórica, vemos que no todos se pueden aplicar sobre cualquier lista. Por ejemplo, no podemos hacer $S_i[]$ ya que no hay elemento al cual sumarle 1. Es por esto que es necesario que se puedan arrojar errores durante la evaluación.

Por este motivo se definió un par de monadas, una encargada de mantener el entorno y otra de los errores. Esta última es similar a la usada en previos trabajos prácticos de la materia. En cuanto a la primera, se definió un entorno (Env) el cual posee dos mapas, uno en donde se almacenan las variables y otro para las funciones, por lo tanto fue necesario poder actualizar y observar cada uno de manera independiente.

```
type Env = (M.Map Nombre (Lista -> Lista), (M.Map Nombre [Integer]))
```

Los posibles errores que pueden ocurrir son de 2 tipos, los relacionados con el dominio de las funciones y los que pueden estar presentes a la hora de buscar variables o funciones a imprimir o utilizar en otra definición.

Como se explicó previamente se implementó funciones derivadas de las bases, a la hora de evaluarlas estas se traducen en base a las funciones base y se las evalúa.

5.5. Print

Como se vió, en el entorno las funciones se almacenan como funciones de Haskell, es decir que no se puede mostrarlas así como así, sino que se debe definir una manera de hacerlo. En mi caso elegí evaluar estas funciones con una lista vacía que luego no seria impresa. Por lo tanto puedo mostrar todos los operadores aplicados. Por otro lado, como también se vió, la composición de funciones funciona al revés de lo comúnmente utilizado. Por lo tanto el primer elemento de la función (primer elemento del AST una vez que se la evalúa) debe ser el ultimo en imprimirse. Por esto se recorrer el árbol de manera inversa.

6. Mejoras posibles

Entre las mejoras que se pueden implementar encontramos:

- Especificación de los mensajes de error tanto de parseo como de evaluación, brindando una explicación del error puntual.
- Agregar potencia de una función, se aplica x veces una función dada.
- Añadir más comandos de inspección de funciones, como por ejemplo analizar el dominio de una función dada. Esto requiere además de un trabajo de implementación en Haskell, un trabajo de investigación teoría sobre algoritmos para calcularlo.
- Cargar de archivos para poder seguir trabajando sobre las funciones y variables que se declaró.

7. Bibliografía

Para la parte teórica de **FRL** se utilizó el material dado en la materia de Lenguajes Formales y Computabilidad, del segundo año de la carrera.

Para escribir el código:

- TPs dados en la materia, y material de la misma.

- Documentación Happy.
- Para la interfaz de usuario System Console Haskeline.