Universidad Nacional de Rosario
Facultad de Ciencias Exactas,
Ingeniería y Agrimensura
Departamento de Ciencias de la
Computación

Análisis de Lenguajes de Programación

TP FINAL

Lucas Bachur

Abril 2024

1 Descripción del proyecto

1.1 Introducción

Hearthstone[1] es un videojuego de cartas coleccionables en línea. Los jugadores comienzan el juego con una colección de cartas básicas, pero pueden obtener más cartas de diferente rareza y poder. Con las cartas de su colección, un jugador puede armar un **mazo**, para luego usarlo en una partida contra otros jugadores.
Cada mazo tiene asociado una de las 11 **clases** del juego, lo que limita las cartas que se pueden poner dentro de ese mazo, ya que algunas cartas solo se pueden usar para clases específicas.

1.2 Objetivos

El lenguaje desarrollado en este trabajo se encarga de manipular mazos de cartas de **Hearthstone**. Se podrán definir mazos y realizar operaciones básicas entre estos (unión, intersección, etc.), y también ver los datos de una carta en específico. Además, se podrán utilizar los **deckstrings** para importar y exportar mazos desde y hacia el juego.

2 Sintaxis del lenguaje

2.1 Sintaxis abstracta

$$
\begin{array}{rcl}
\mathit{filter} & ::= & \mathit{field} \\
& | & \mathit{fieldn} \textbf{ : } \mathit{nat} \\
\\
\mathit{carddef} & ::= & \mathit{name} \\
& | & \mathit{var} \\
\\
\mathit{cardunit} & ::= & \mathit{carddef} \\
& | & \textbf{2x } \mathit{carddef} \\
\\
\mathit{deckexp} & ::= & \mathit{deckexp} + \mathit{deckexp} \\
& | & \mathit{deckexp} \text{ - } \mathit{deckexp} \\
& | & \mathit{deckexp} \cap \mathit{deckexp} \\
& | & \textbf{filter } [\mathit{filter}]\ \mathit{deckexp} \\
& | & [\mathit{cardunit}] \\
& | & \mathit{var} \\
& | & \textbf{import } \mathit{code} \\
\\
\mathit{deckhero} & ::= & \mathit{var} \\
& | & \mathit{name} \\
\\
\mathit{comm} & ::= & \mathit{var} = \mathit{deckexp} \\
& | & \textbf{show } \mathit{deckexp} \\
& | & \textbf{isDeck } \mathit{deckexp} \\
& | & \textbf{export } \mathit{deckexp}\ \mathit{deckhero} \\
& | & \textbf{cardData } \mathit{carddef}
\end{array}
$$

donde:

- *var* representa al conjunto de identificadores de variables
- *nat* representa al conjunto de los números naturales
- *name* representa al conjunto de identificadores de nombres de objectos del programa
- *code* representa al conjunto de cadenas que pueden llegar a ser un **deckstring**
- $field$ y $fieldn$ representan al conjunto de identificadores de los filtros posibles que se usan solos o necesitan de un número acompañado, respectivamente

2.2 Sintaxis concreta

$$
\begin{array}{rcl}
digit & ::= & \texttt{'0'} \mid \ldots \mid \texttt{'9'} \\
letter & ::= & \texttt{'a'} \mid \ldots \mid \texttt{'Z'} \\
codechar & ::= & letter \mid digit \mid \texttt{'+'} \mid \texttt{'/'} \mid \texttt{'='} \\
nat & ::= & digit \mid digit\ nat \\
var & ::= & letter \mid letter\ var \\
code & ::= & codechar \mid code \\
name' & ::= & var\ \texttt{' '}\ var \\
 & \mid & var \\
name & ::= & \texttt{'"'}\ name'\ \texttt{'"'} \\
 & & \\
field & ::= & \texttt{'Minion'} \mid \texttt{'Spell'} \mid \texttt{'Weapon'} \\
 & \mid & \texttt{'Hero'} \mid \texttt{'Location'} \\
fieldn & ::= & \texttt{'Attack'} \mid \texttt{'Health'} \mid \texttt{'Cost'} \\
 & & \\
filter & ::= & field \\
 & \mid & fieldn\ \texttt{':'}\ nat \\
 & & \\
filterlist & ::= & filter\ \texttt{','}\ filterlist \\
 & \mid & filter \\
 & \mid & \\
 & & \\
carddef & ::= & name \\
 & \mid & var \\
 & & \\
cardunit & ::= & carddef \\
 & \mid & \texttt{'2x'}\ carddef \\
 & & \\
cardlist & ::= & cardunit\ \texttt{','}\ cardlist \\
 & \mid & cardunit \\
 & \mid & \\
 & & \\
deckexp & ::= & deckexp\ \texttt{'+'}\ deckexp \\
 & \mid & deckexp\ \texttt{'-'}\ deckexp \\
 & \mid & \texttt{'inCommon ('}\ deckexp\ deckexp\ \texttt{')'}
\end{array}
$$

	| 'filter ([' *filterlist* ']' *deckexp* ')'
	| '[' *cardlist* ']'
	| *var*
	| 'import' *code*
deckhero	::= *var*
	| *name*
comm	::= *var* '=' *deckexp*
	| 'show' *deckexp*
	| 'isDeck' *deckexp*
	| 'export' *deckexp deckhero*
	| 'cardData' *carddef*

3 Instalación

Para la instalación es necesario tener en nuestro sistema **Haskell** y **Stack**. La descarga del proyecto en sí se realiza desde https://github.com/LucasBachur/ALP-TPFinal.

4 Manual de uso

Para usar el intérprete, nos situamos en la carpeta principal del proyecto y usamos `stack build`, seguido de `stack exec TPFinal-exe`. Alternativamente se pueden unir estos 2 pasos usando el comando `stack run`.

4.1 Comandos de uso general

Apenas abrimos el intérprete, podremos utilizar el comando `:?` para visualizar los formatos de entrada válidos que podemos ingresar. Los comandos de uso general disponibles para usar son:

- `:?` o `:help` : coma ya fue mencionado, muestra los posibles comandos que se pueden ingresar.
- `:load <file>` : compila las lineas que se encuentren dentro de un archivo objetivo. El archivo se tiene que encontrar dentro de la carpeta `Files` del proyecto, y solo puede contener definiciones de mazos, ningún otro comando. Alternativamente, se puede compilar uno o más archivos junto con el intérprete. Para eso, agregamos los nombres de los archivos a la hora de ejecutarlo. Por ejemplo: `stack exec TPFinal-exe Ejemplo1.hes`.
- `:browse` : muestra los nombres de los mazos guardados en el entorno.
- `:quit` : cierra el programa.

4.2 Comandos de uso específico

Los otros comandos que podemos usar son específicos de este lenguaje y ya los definimos dentro de la sintaxis:

- `<var> = <expr>` : define un mazo, guardándolo dentro del entorno. El nombre asignado al mazo puede contener cualquier combinación de caracteres alfanuméricos, aunque siempre debe empezar con una letra.
- `show <expr>` : evalúa una expresión de mazo, mostrándolo carta por carta.
- `isDeck <expr>` : muestra si una expresión corresponde a un mazo válido para usar dentro del juego o no. Para este proyecto se toma el siguiente criterio: si el mazo contiene 30 cartas (excepto en el caso de que el mazo contenga a `"Prince Renathal"`, que tiene el efecto de cambiar el tamaño de tu mazo a **40** cartas) y si cada carta aparece **hasta** 2 veces dentro del mazo, entonces el mazo es válido.
- `export <expr> <hero>` : toma una expresión de mazo y un héroe que asignarle, los transforma a un **deckstring** y lo muestra. Esta cadena se puede copiar al portapapeles para luego importar el mazo dentro del juego.
- `cardData <id>/"<name>"` : dado el id o el nombre de una carta, muestra datos asociados a la misma.

4.2.1 Héroes

Como dijimos al comienzo, el juego tiene 11 clases distintas y cada mazo tiene asignada una de estas, lo cual limita las cartas que se pueden utilizar en este. Cada clase, a su vez, tiene distintos `héroes` jugables, que cambian únicamente la manera en la que se ve nuestro personaje dentro del juego, pero no tiene efecto alguno en ningún aspecto relevante de una partida. Cuando creamos un mazo dentro del juego, se le asigna un héroe o un grupo de héroes. Para simplificar el lenguaje definido en este proyecto, se tomó la decisión de que el héroe se le asigne a un mazo solamente a la hora de exportalo, y que la cantidad este limitada a uno solo. Cada clase tiene un héroe por defecto para asignar, así que dentro de este lenguaje definimos el héroe que le asignamos a un mazo como:

- El nombre de la clase: le asigna el héroe por defecto de esa clase.
- El nombre del héroe: le asigna ese mismo héroe, buscando el id dentro de la data que se encuentra en el estado del programa. Se escribe entre comillas para diferenciarlo con el posible nombre de la clase.

4.3 Expresiones de mazo

La manera más elemental de definir un mazo es mediante el uso de `'[' ']'` como constructores de lista, con una `','` separando cada elemento. Cada uno de los elementos va a ser una definición de una carta en particular, la cual puede estar expresada por su id (un número natural), o por su nombre (una cadena que escribimos entre comillas). Si alguna de las cartas no puede encontrada de la manera que fue definida, esa entrada en particular será ignorada y el mazo va a tratarse como si fuera definido sin esa carta en particular. Además, a la definición de una carta se le puede aclarar si hay 2 copias dentro de un mazo con el prefijo `'2x'`. Se tomó la decisión de solamente poder definir hasta 2 copias de una carta a la vez ya que, dentro del juego, los mazos solo pueden consistir de hasta 2 apariciones de cada posible carta. Igualmente, los mazos que se manejan en el intérprete pueden manejar más cantidad de copias por carta, simplemente no existe un atajo para escribir esto dentro de la sintaxis.

Ejemplos de posibles definiciones de mazos:

```
deck1 = [34,78]
deck2 = [4567, "Cheesemonger"]
deck3 = []
deck4 = [2x"Cheesemonger", "Reno Jackson"]
```

Luego, tenemos constructores que nos permiten definir expresiones de mazos más complejas:

- `deckexp + deckexp` : unión de dos mazos.
- `deckexp - deckexp` : diferencia entre dos mazos.
- `inCommon(deckexp deckexp)` : intersección de dos mazos.
- `filter([filterlist] deckexp)` : filtrado de un mazo en base a una lista de filtros dada. Los filtros están separdos por `','`, y si tienen un número adjunto se le agrega con `':'`.
- `import code` : transformo un **deckstring** en el mazo que representa.

Por último, puedo también referenciar un mazo ya definido en el entorno a través de su nombre. Otros ejemplos de posibles definiciones de mazos:

```
deck5 = [34,78]+[4567, "Cheesemonger"]
deck6 = deck4-["Cheesemonger"]
deck7 = inCommon(deck1 deck4)
deck8 = filter ([Attack:3,Minion] deck5)
deck9 = import AAEBAfHhBAblsAS/zgTt/wX/lwaAnwb2owYM8OME9eMEmMQFyoMG0IMG9YwG94wGhY4GyZEG85EGiJIGkqAGAA==
```

5 Organización de los archivos

En el proyecto encontramos 4 carpetas:

- `app` : contiene `Main.hs`, que es el archivo principal (de este se genera el ejecutable).
- `files` : aquí se guardan los archivos que uno luego quiera cargar dentro del intérprete.
- `json` : contiene `cards.collectible.json`, que es el archivo que contiene los datos de todas las cartas coleccionables del juego (actualizado a Abril de 2024).
- `src` : contiene el archivo Parse.y (que contiene la sintaxis para parsear el lenguaje), y los módulos Common (definición de estructuras), Comms (definición de las funciones que utilizan los comandos de uso específico), Deckstring (definición de funciones que trabajan con deckstrings), Monads (definición de mónadas), PrettyPrinter (impresión de elementos) y ReadJSON (funciones para leer el archivo JSON).

6 Lectura del archivo JSON

El archivo JSON e información sobre sus contenidos fueron conseguidos a través del sitio `hearthstonejson.com`[2]. Se puede acceder directamente a la última versión de los datos

disponible en la página a través de la URL: api.hearthstonejson.com/v1/latest/
La librería utilizada para el parseo del archivo es `aeson`[3] [4]. Para poder darle uso a esta librería primero definimos una estructura llamada `HearthstoneCard` (ubicada en el Common), y luego la instanciamos para `FromJSON`. Esto nos permite hacer uso de la función `eitherDecode`, que nos va a devolver un array de `HearthstoneCard`, con lo cual podemos comenzar a trabajar más cómodamente con los datos. Cabe aclarar que tanto cuando definimos `HearthstoneCard`, como cuando la instanciamos, tienen que estar presentes **todos** los campos que pueden llegar a aparecer en un elemento dentro del JSON, aunque luego no vayamos a utilizarlos a todos. La información obtenida la guardamos dentro de la estructura:

```
data CardData = CardData {
  idMap :: Map.Map Int HearthstoneCard,
  nameMap :: Map.Map String HearthstoneCard
} deriving Show
```

Donde básicamente vamos a tener repetidas las cartas: una en cada Map. La diferencia va a ser que en uno las claves son los id y en el otro los nombres. Esto fue hecho de esta manera para poder buscar con la menor complejidad posible las cartas que se definan tanto por su id como por su nombre.

7 Deckstrings

El formato de **deckstring**[5] de Hearthstone es una cadena de bytes codificada en base 64. Este formato permite a los jugadores compartir mazos de Hearthstone de una manera sencilla, importando o exportando los mazos del juego a través de estos códigos. Cada deckstring tiene una cabecera (con datos generales del mazo, que van a ser constantes para los alcances de este proyecto), y un 4 bloques en este orden: los héroes asociados al mazo, las cartas que poseen una sola copia, las cartas con 2 copias y las cartas con n-copias. Cada bloque empieza con un **varint**[6] respresentando el largo del resto del bloque. Luego, para los 3 primeros bloques, el resto consiste en un array de varints. En cambio, para el último es un array de pares de varint. Cada carta está representada por su id única. Una aclaración relevante es que si bien el orden de las cartas no importa al importar o exportar un mazo, solo vamos a obtener exactamente el mismo deckstring si el mazo es igual pero también lo ordenamos de la misma manera.

8 Decisiones de diseño

8.1 Main

El estado de un programa lo guardamos como:

```
data State = S { inter :: Bool, cd :: CardData, ve :: NameEnv }
```

donde sus campos son: un Bool que indica si estamos en modo interactivo, un CardData (la estructura que definimos para guardar la información leída del JSON) y un NameEnv. Esta última es la estructura que representa las expresiones definidas dentro del entorno y sus respectivos nombres. Esta definida como:

```
type NameEnv = [(DeckName, DeckExp)]
```

8.2 Procesamiento de mazos

Dentro del Common tenemos 2 definiciones distintas para un mazo: una que hace referencia directa a la expresión del mazo como se ve en la sintaxis, y otra que trata los mazos de manera más literal. La primera está definida como:

```
data DeckExp = Union DeckExp DeckExp
             | Diff DeckExp DeckExp
             | InCommon DeckExp DeckExp
             | Filter [Filter] DeckExp
             | CardList CardList
             | Deck DeckName
             | Import String
            deriving (Show,Eq)
```

Y la definición más literal hace un mapeo desde el id de cada carta hacia una tupla que representa la cantidad de copias de la carta en el mazo y los datos de la misma:

```
type HearthstoneDeck = Map.Map Int (Int, HearthstoneCard)
```

Al usar un comando que utiliza un mazo, lo primero que se hace es convertir la expresión de mazo dada a un mazo literal. Si llegase a ocurrir algún error durante el pasaje, la operación falla mostrando un mensaje con la razón.

9 Referencias

1 - https://es.wikipedia.org/wiki/Hearthstone
2 - https://hearthstonejson.com/docs/cards.html
3 - https://www.schoolofhaskell.com/school/starting-with-haskell/libraries-and-frameworks/text-manipulation/json
4 - https://hackage.haskell.org/package/aeson-2.2.1.0/docs/Data-Aeson.html#g:2
5 - https://hearthsim.info/docs/deckstrings/
6 - https://protobuf.dev/programming-guides/encoding/