Universidad de Santiago de Chile

Facultad de Ingeniería

Departamento de Ingeniería Informática

Paradigmas de Programación

Proyecto semestral de Laboratorio: Paradigma Funcional

Autor: Sebastián Paillao

Profesor: Edmundo Leiva

Fecha de entrega: 09 de Octubre, 2023

Santiago, 2023

Índice

1. Introducción

En el ramo Paradigmas de Programación, correspondiente al nivel 4 de la carrera se estudian distintos paradigmas usados por desarrolladores para resolver las diversas problemáticas que se les plantean. Por lo mencionado anteriormente, en el presente informe, se abordará lo que fue la experiencia y el proceso necesario para la realización de un sistema de chatbots en el lenguaje de programación Scheme, aplicando lo aprendido en clases sobre el paradigma funcional y planteando como objetivo el dejar constancia del avance y del proceso de aprendizaje hecho a lo largo del semestre, además de ofrecer un instructivo al usuario para el uso del programa.

1.1. Descripción del problema

Se solicita el desarrollo de un sistema de chatbots, que debe poseer diversos bots de chat que deben estar conectados entre sí, de forma que una opción lleve al usuario a otro chatbot con otras opciones distintas relacionadas a la elección hecha por el usuario. Dentro del programa debe existir la posibilidad de crear opciones, chatbots, flujos, vincular chatbots con opciones y sus flujos, permitir la interacción entre el usuario y los chatbots, agregar chatbots a un sistema, añadir flujos a un chatbot, guardar un historial para de las interacciones hechas entre cada usuario y hacer una síntesis sobre ellas. Para esto se debe tener en cuenta la existencia de distintos elementos que serán propios del programa y que funcionarán en conjunto para su correcto funcionamiento.

1.2. Descripción del paradigma funcional

En el ámbito de la programación, el paradigma funcional refiere al manejo exclusivo de funciones con un concepto similar al que se tiene de ellas en matemáticas **f(x)**, obteniendo un valor, llevando a cabo el proceso propio de cada función, y retornando el resultado. Estos resultados obtenidos pueden ser usados dentro otras funciones que llevan a cabo otros procesos distintos y retornan también los resultados. El paradigma a emplear se basa en eso, en trabajar con esas funciones y usar correctamente el pensamiento abstracto. Debido a esto se incluyen ciertas restricciones dentro de las instrucciones del laboratorio ya que el lenguaje de programación Scheme no corresponde a un lenguaje completamente funcional. Por esto se restringe el uso de funciones como set! para evitar salirse del paradigma funcional evitando darle valores a las variables dentro del mismo código. En este caso se requiere hacer uso de distintos “TDAs”, que serán las estructuras base para el correcto

funcionamiento del código, que deberán ser trabajados con funciones de orden superior basadas en listas y serán explicadas en la siguiente sección de este informe.

2. Desarrollo

2.1. Análisis del problema

Los requerimientos se dividen en requerimientos no funcionales y funcionales. Los no funcionales son incluir una autoevaluación en la entrega, programar en DrRacket 6.11 o superior, utilizar la librería nativa del lenguaje, comentar todas la funciones, respetar los dominios y recorridos solicitados para cada funcion, separar los TDA en archivos separados y mantener un historial de trabajo en el repositorio de GitHub a lo menos comenzando 14 días antes del día de la entrega.

Requerimientos funcionales:

a) **TDAs**: Según las instrucciones documentadas, las estructuras mínimas a crear serán los TDA system, chatbot, option, user, chatHistory y flow.
b) **TDA Option - constructor:** Función constructora de una opción para flujo de un chatbot. Cada opción se enlaza a un chatbot y flujo especificados por sus respectivos códigos.
c) **TDA Flow - constructor:** Función constructora de un flujo de un chatbot.
d) **TDA Flow - modificador:** Flow - modificador. Función modificadora para añadir opciones a un flujo.
e) **TDA chatbot - constructor:** Función constructora de un chatbot.
f) **TDA chatbot - modificador:** Función modificadora para añadir flujos a un chatbot, debe usar recursión.
g) **TDA system - constructor:** Función constructora de un sistema de chatbots. Deja registro de la fecha de creación.
h) **TDA system - modificador:** Función modificadora para añadir chatbots a un sistema.
i) **TDA system - modificador:** Función modificadora para añadir usuarios a un sistema.
j) **TDA system:** Función que permite iniciar una sesión en el sistema.
k) **TDA system:** Función que permite cerrar una sesión abierta.
l) **system-talk-rec:** Función que permite interactuar con un chatbot
m) **system-talk-norec:**Función que permite interactuar con un chatbot. Mismo propósito de la función anterior pero con una implementación declarativa.
n) **system-synthesis:** Función que ofrece una síntesis del chatbot para un usuario particular a partir de chatHistory contenido dentro del Sistema.

o) **system-simulate:** Permite simular un diálogo entre dos chatbots del sistema.

En resumen, es requerido crear cada TDA y cada una de las funciones especificadas dentro de cada TDA como funciones adicionales, lo que requerirá ser muy organizado con cada archivo y sus respectivas funciones para después poder implementarlas en el main sin problemas. Además, cada funcion debe ejecutarse según la estructura del ejemplo de uso y los scripts de prueba otorgados en el documento.

2.2. Diseño de la solución

2.2.1. TDAs

La solución a la problemática fue siguiendo la estructura requerida, se empezaron a trabajar los TDA y las funciones en el mismo orden solicitado, aquí se muestra cómo se trabajó cada TDA y la funciones que incluyen:

a) **TDA option:** El constructor option toma un id, mensaje, ChatbotCodeLink, InitialFlowCodeLink y los agrupa en una lista. Esta lista representa una opción en un chatbot. Para verificar si una lista es una opción válida, se usa la función option?. El TDA también tiene selectores, funciones que obtienen ciertos valores de la lista, como el código de la opción o el mensaje.

b) **TDA flow:** El constructor flow recibe un ID, un nombre-mensaje y una lista de opciones. Retorna una lista que representa un flujo, eliminando opciones duplicadas con la función auxiliar "*unique-options*".

   (1) **flow-add-option, modificador:** Toma un flujo y una option, verifica si el código de la opción, obtenido a través de la función "*get-optionCode*", ya está en la lista de códigos de las opciones del flujo. Si no está, añade la option al final del flujo. Finalmente, devuelve el flujo, ya sea modificado o no. Esto significa que la función no permite la duplicación de opciones en el mismo flujo.

c) **TDA chatbot:** El constructor chatbot toma un ID, nombre, mensaje de bienvenida, enlace de flujo inicial y flujos. Crea un chatbot como una lista única de estos, eliminando flujos duplicados con la función auxiliar "*unique-flows*".

   (1) **chatbot-add-flow**, **modificador:** Recibe un chatbot y un flujo. Su objetivo es añadir el flujo dado al chatbot, si aún no existe en él. Usa la función auxiliar "*flow-exists?* "para verificar si el flujo ya está presente. Si no está, lo añade al final de la lista de flujos del chatbot y devuelve un nuevo chatbot. Si ya existe, retorna el chatbot sin ese cambio. Usa recursión de cola, cumpliendo con la recursión requerida.

d) **TDA system:** El constructor recibe un nombre, un código de chatbot inicial y una lista de chatbots. Retorna una lista que representa el sistema, eliminando chatbots duplicados usando una función auxiliar. Al crearse deja una lista y una variable vacías, donde se almacenarán los usuarios registrados y el usuario loggeado actual (*currentUser*).

(1) **system-add-chatbot, modificador:** Toma un sistema y un chatbot. Si el chatbot no está presente en el sistema, lo añade a la lista de chatbots y devuelve un sistema modificado. Si ya está presente, retorna el sistema original**.**

(2) **system-add-user, modificador:** Toma un sistema y un nombre de usuario. Si el usuario no está presente en el sistema, crea un nuevo usuario y lo añade a la lista, retornando un sistema modificado. Si el usuario ya está presente, devuelve el sistema sin cambios.

(3) **system-login, modificador:** Toma un sistema y un nombre de usuario. Si el usuario existe y no hay otro usuario loggeado, se inicia sesión con ese usuario, retornando el sistema modificado con el nombre del usuario loggeado actualmente en el parámetro "*currentUser*". Si no se cumple alguna de esas condiciones, retorna el sistema original.

(4) **system-logout, modificador:** Toma un sistema. Si hay un usuario loggeado, cierra la sesión y modifica el sistema para reflejar el cambio, dejando libre "*currentUser*" Si no hay un usuario loggeado, devuelve el sistema sin cambios.

(5) **system-talk-rec:** Recibe un sistema y un mensaje, el mensaje lo intenta transformar a int, en caso de que sí se pueda, consigue el id de las opciones del flujo del chatbot del sistema y compara recursivamente el mensaje transformado con los id de las opciones, al encontrar la opcion correcta, retorna el sistema con el ChatbotCodeLink correspondiente al de la opción seleccionada, después de guardar los datos de la interacción en chatHistory. Mismo caso si no logra transformarlo a int, pero compara recursivamente el string de los mensajes transformado a minúscula con cada keyword de las opciones del flujo del chatbot del sistema transformadas a minúsculas también, en caso de encontrar coincidencia, el mismo procedimiento, si no encuentra nada en ninguno de los dos casos, retorna un error.

(6) **system-talk-norec:** Mismo funcionamiento que la función anterior, pero sin comparar recursivamente las coincidencias.

(7) **system-synthesis:** Imprime bajo un formato estructurado las interacciones entre el usuario y los chatbots guardadas en *"chatHistory"* guardadas en *"user"* guardado en *"system".*

e) TDA user: El constructor del TDA requiere solo un username, y es llamado al momento de añadir un usuario nuevo al sistema con la función "*system-add-user*"*.* Éste al crearse posee una lista vacía donde se guardarán los mensajes del TDA chatHistory.

f) TDA chatHistory: El constructor de chatHistory es una función que se llama dentro de "*system-talk-rec*", y que guarda en su interior sender, message y option, utilizado para guardar en un usuario su historial de interacciones con el sistema de chatbots.

3. Aspectos de implementación

Dados los requerimientos, se registra cada TDA con sus respectivas funciones por separado, al final de cada uno de los archivos de código se hace uso de "*(provide (all-defined-out))",* función de Scheme que exporta todas las funciones creadas para poder importarlas en otro código, así son importadas las funciones de cada uno de los TDA en el archivo "*main.rkt"* junto al script de prueba.

4. Instrucciones de uso

Para poder ejecutar el programa y hacer uso de sus funciones, debe tenerse el entorno **DrRacket,** ejecutarse el archivo "*main.rkt",* y correrlo en el programa. Las distintas funciones y sus usos se encuentran especificadas en el script de prueba.

5. Resultados y autoevaluación

Finalizando ya el tiempo para el proyecto debo hacer una autocrítica más que nada por no haber aprovechado lo suficiente el tiempo disponible. Cumplí con el tiempo para el historial de avances en GitHub pero me confié mucho de mi avance en los últimos días. Los últimos tres días estuve prácticamente sólo desarrollando system-talk-rec, pero no logré hacerla funcionar como se debía, lo mismo con las otras funciones siguientes. Podría decir que de esta experiencia rescato mucho el haber aprendido a pensar de manera abstracta, fue difícil, estresante pero también fue entretenido estudiar este nuevo paradigma presentado. Logré cumplir con la gran mayoría de las funciones, definí buenos TDA, hice buen uso de sus funciones pero el haber confiado tanto en el avance hecho y no preocuparme de lo que me quedaba por hacer terminó jugándome en contra estos últimos días, si hubiera comenzado antes probablemente lo hubiera logrado terminar. Aún así me siento satisfecho ya que el poco tiempo que tuve libre lo dediqué siempre al proyecto y dentro de eso hice mi mayor esfuerzo.

6. Conclusión

Como se evidencia anteriormente, a pesar de no haber cumplido completamente con el proyecto, se rescata el aprendizaje sobre este nuevo paradigma, sobre la gestión de los tiempos y sobre la exigencia de este ramo, aprendizaje que sin duda será aplicado en el desarrollo de los siguientes dos laboratorios de los paradigmas que quedan por aprender.