Instituto Tecnológico de Costa Rica

Semestre I

2021

WazeLog: Documentación técnica

Lenguajes, compiladores e intérpretes

Tarea#1: Programación funcional

Prof. Marco Rivera Meneses

José Alejandro Chavarría Madriz

Natalia González Bermúdez

2019067306

2019165109

Índice

Descripción de hechos y reglas

Hechos

Grafo

En general los hechos se dividen en dos grandes grupos. El primer gran grupo son todos los hechos que definen al grafo. Para describir el grafo se utilizó la siguiente estructura: arco(Origen, Destino, Distancia, Tiempo, Tiempo en presa). Esto quiere decir que existe una arista de origen a destino con una distancia, que toma tal tiempo recorrer y tal otro si hay presa. Tanto origen y destino son expresados mediante strings, mientras que la distancia y los tiempos se expresan mediante números. Es importante recalcar que la relación de arco NO es biyectiva, por lo que en caso de que se pueda regresar del destino al origen se define como un arco por separado, agregando así, también, la libertad de modificar cualquier dato que no sea recíproco respecto un sentido y el otro.

Gramatica

El otro grupo importante de hechos que se define son todos los relacionados al vocabulario. En general se tienen 8 tipos. preposición(), nombre(), conector(), artículo(), lugar(), objeto(), verbo() y verbo_lugar(). Excluyendo aquellos que quedan claros por su nombre, verbo_lugar() se refiere a todos aquellos verbos que comúnmente se utilizan para indicar un lugar de procedencia o destino. lugar() está específicamente ligado con la base de datos del grafo, indica todos los orígenes y destinos del grafo, y nombre() son más que todo expresiones en primera persona utilizadas al referirse a lugares y finalmente objeto() se refiere a locales, que podrían encontrarse en los lugares estipulados en el grafo (Ej: farmacia).

Reglas

Gramática:

Oración:

oracion(X,Y), donde X es la representación de una oración (lista de strings) y L es es lugar que se detecta en la oración. Con esta regla se puede formar una oración con:

- Un nombre y un sintagma_verbal_lugar. Ej: Yo voy a Aserri.
- Un nombre y un sintagma_nominal_lugar. Ej: Yo estoy en Aserri.
- Un sintagma_verbal_lugar. Ej: Estoy en Aserri.
- Un sintagma_nominal. Ej: Al supermercado.
- Un sintagma_nominal_lugar. Ej: A Aserri.
- Un sintagma_verbal y un sintagma_nominal_lugar. Ej: Yo voy a Aserri.
- Un sintagma_verbal y un sintagma_verbal_lugar. Ej: Yo quiero ir a Aserri.
- Un sintagma_nominal y un sintagma_verbal_lugar. Ej: El supermercado está en Aserri.

Sintagma nominal (lugar):

sintagma_nominal_lugar(X,Y), donde X es la representación del sintagma (lista de strings) y Y es el lugar que se detecta. Con esta regla se puede formar un sintagma_nominal_lugar formado por:

- El lugar. Ej: Aserri.
- Una preposición y un lugar. Ej: A Aserri.
- Un sintagma nominal y un lugar. Ej: A la farmacia.

Sintagma nominal:

sintagma_nominal(X,Y), donde X es la representación del sintagma (lista de strings) y Y es el lugar que se detecta. Con esta regla se puede formar un sintagma_nominal formado por:

- Un artículo y un lugar. Ej: La farmacia.
- Un artículo y un lugar, que a su vez es un objeto. Ej: Una tienda.
- Una preposición y un artículo. Ej: En la.

Sintagma verbal (lugar):

sintagma_verbal_lugar(X,Y), donde X es la representación del sintagma (lista de strings) y Y es el lugar que se detecta. Con esta regla se puede formar un sintagma_verbal_lugar formado por:

- Un verbo_lugar. Ej: Ir.
- Un verbo_lugar y un sintagma_nominal de lugar. Ej: Estoy en Aserri.

Sintagma verbal:

sintagma_verbal(X), donde X es la representación del sintagma (lista de strings). Con esta regla se puede formar un sintagma_verbal formado por:

- Un nombre y un verbo. Ej: Yo quiero.
- Un verbo y un conector. Ej: Tengo que.

Grafo:

Camino y viajar

camino(Inicio,Final,Camino,Largo,Tiempo,TiempoPresa), es la regla principal de viajar. Encuentra todas las rutas posibles que hay desde un nodo A a un nodo B.

viajar(Inicio,Final,Visitados,[Final|Visitados],Largo,Tiempo,TiempoPresa). Comprueba ruta directa, agrega Final al camino o comprueba un nodo de conexión e intenta llegar recursivamente al destino desde ese nodo.

Shortest(Inicio,Final,Camino,Largo,Tiempo,TiempoPresa)

Utilizando camino y minimal encuentra la ruta más corta de un nodo A a B.

minimal(Set de rutas completas , ruta completa mínima)

Dónde completo significa una lista con [Caminó,Largo,Tiempo,TiempoPresa]. Determina de un set de caminos cúal es el que tiene menor distancia.

min(Resto,Frente,Minimo)

Dónde mínimo es una ruta completa. Comprueba recursivamente cual ruta tiene la menor distancia que otra.

miRuta(Lista de destinos,Ruta,Largo,Tiempo,TiempoPresa)

Utilizando shortest, recursivamente busca y concatena las rutas más cortas de cada lugar en la lista de destinos en el orden dado Ej: Si Lista de destinos es [A,B,C] recursivamente busca la ruta más corta de A a B, luego de B a C y las concatena sumando sus valores de distancia y tiempos.

wazeLogIn (Destinos)

Dónde destinos es una lista de lugares en el grafo. Imprime las preguntas necesarias para saber si qué tiempo se requiere e imprime la ruta final al usuario utilizando miRuta(). Regla principal de wazeLogOut

wazeLogOut(Destinos,Bool)

Imprime las rutas para cuando hay presa o para cuando no, dependiendo del bool. También imprime mensajes de error para cuando no se entiende la respuesta a si es hora pico o no e imprime si no se logra encontrar una ruta. Sucede cuando no existe la ruta.

Interfaz:

Inicio del programa:

wazeLog: Inicializa el programa. Junta la oración inicial de wazeLog con el usrLog().
usrLog(): Se encarga de tomar el camino construido y enviarlo al grafo para que analice la mejor ruta.
wazeSaludo: Escribe la oración inicial de WazeLog.

Obtener ruta completa:

usrOracion(X): Donde X es el camino que se forma después de tener una conversación con el usuario. Se encarga de llamar a todas las funciones que toman las respuestas del usuario y forma una lista con los lugares de origen, paradas y destino.

Obtener origen:

usrOrigen(X): Donde X es el lugar de origen o el lugar en donde se encuentre el usuario.

- Si la respuesta es una oración válida, encuentra el lugar de origen indicado por el usuario.
- Si la respuesta no es una oración válida le indica al usuario que no entendió y se vuelve a llamar recursivamente.

Obtener paradas:

usrParadas(X): Donde X es una lista con todas las paradas que necesita hacer el usuario antes de llegar a su destino.

- Recibe la respuesta del usuario y llama a todas las reglas auxiliares recursivas (paradas) para definir qué hacer con la respuesta.
- Si la respuesta no es una oración válida le indica al usuario que no entendió y se vuelve a llamar recursivamente.

Regla auxiliar paradas(Input,Lista,Parada,Paradas): Donde Input es la respuesta del usuario en forma de string, Lista es la oración dividida en una lista de palabras, Parada es la nueva parada que indica el usuario y Paradas es una lista con todas las paradas.

- Si la oración es válida, extrae el lugar y lo agrega a la lista de Paradas y vuelve a llamar a usrParadas.
- Si la respuesta del usuario es “no”, es decir no tiene más paradas, termina la recursión de paradas.
- Si la respuesta del usuario es “sí”, es decir tiene más paradas, vuelve a llamar a usrParadas.

Obtener destino:

usrDestino(X): Donde X es el lugar de destino, a donde quiere llegar el usuario

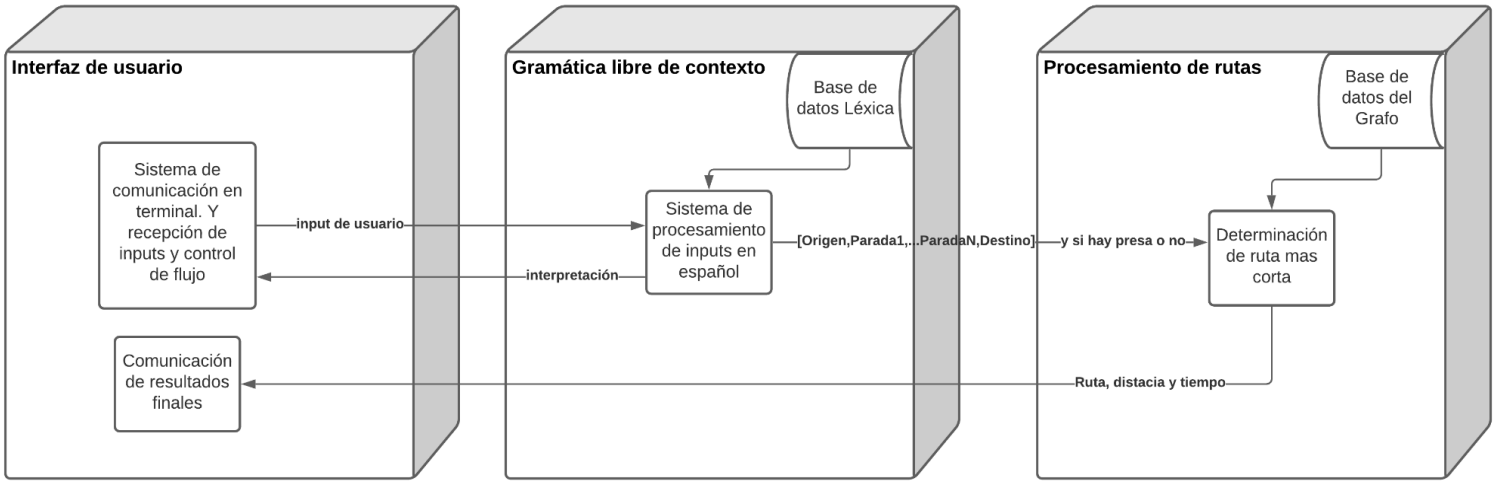
- Si la respuesta es una oración válida, encuentra el lugar de destino indicado por el usuario
- Si la respuesta no es una oración válida le indica al usuario que no entendió y se vuelve a llamar recursivamente.

Verificación de lugar:

usrObjeto(Lugar,X): Donde X es es lugar indicado por el usuario, ya sea un objeto, y X es el lugar una vez que especifique.

- Si el lugar indicado es específico y no un objeto, le asigna el valor de Lugar a X.
- Si el lugar indicado por el usuario no es lo suficientemente específico, se llama recursivamente hasta que indique un lugar que se encuentre en el mapa.
- Si la respuesta no es válida, se le avisa al usuario y se vuelve a llamar usrObjeto.

Diagrama de Arquitectura del programa



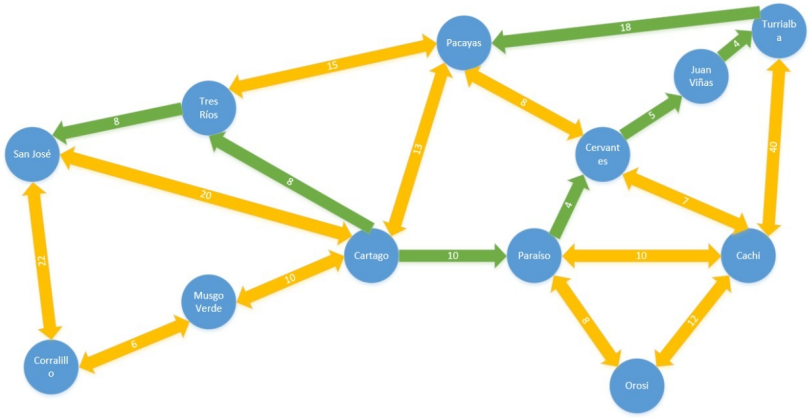
Estructuras de datos desarrolladas

Listas

Para representar el camino que va a recorrer el usuario se utiliza un lista que contiene los lugares que se van obteniendo en la conversación y se guardan en el orden en el que se quieren recorrer, es decir: [Origen, Parada_1, Parada_2, …,Parada_n, Destino] en el que cada elemento de la lista es un string con el nombre de los lugares que hay en la base de datos. El orden de las paradas es relevante, el programa siempre dará prioridad a las primeras paradas aun cuando exista una ruta más corta, pues se asume que el usuario desea realizar las paradas en ese orden específico.

Grafo

Con el fin de representar todas las rutas y caminos posibles en una base de datos se utiliza un grafo mixto. Es decir, todos los potenciales destinos u orígenes están representados por nodos, y sus distancias por aristas que van, teóricamente, en una o dos direcciones, sin embargo al implementarlo mediante el hecho arco(Origen, Destino, Distancia, Tiempo, Tiempo en presa), esto se define como una arista unilateral, por lo que para presentar la bilateralidad es necesario declarar un arco(Destino, Origen, Distancia, Tiempo, Tiempo en presa). Podría también considerarse que algunos de los hechos lugar() corresponden a los nodos del grafo, pero no exclusivamente. El grafo utilizado en el programa es el siguiente (los tiempos se calcularon con un promedio de 60 km/h sin presa y 30 km/h con presa):



Algoritmos desarrollados

Conversación con el usuario

Para lograr formar el camino que va a seguir el usuario se necesita identificar su lugar de origen, destino y todas las paradas que debe hacer. Esto se hace utilizando la gramática libre de contexto para tener una conversación con el usuario y se realiza un algoritmo que forma una lista con todos los lugares en el orden que los debe visitar. Se inicia con wazeLog que junta a wazeSaludo y a usrLog. Con usrLog se llama a la regla usrOracion(Camino) que es la que se encarga de formar el camino. Primero llama a usrOrigen(Origen) y le pregunta al usuario donde se encuentra, de la respuesta obtiene el lugar y si el usuario no escribe un lugar o una oración válida, escribe un error y se vuelve a llamar recursivamente hasta que pueda obtener una localización y asignarla a Origen. Seguidamente se llama a la regla usrDestino(Destino) que funciona igual que la anterior con la diferencia de que cuando obtiene un lugar válido se lo asigna a Destino. Por último, se llama a usrParadas(Paradas) que se llama recursivamente y va guardando en una lista los lugares por los que necesita pasar el usuario antes de llegar a su destino. La lista Paradas se guarda cuando el usuario dice que no necesita hacer más paradas. Después cuando usrOracion obtiene Origen, Destino y Paradas las une en una misma lista que se convierte en Camino y al obtener Camino usrLog() se encarga de enviar esa lista a wazeLogIn(Camino) para que, utilizando el grafo, escriba la ruta con su distancia y tiempo estimado.

Además existe una regla llamada usrObjetos(Lugar,X) que se utiliza en usrOrigen(Origen), usrDestino(Destino) y usrParadas(Paradas) y se encarga de verificar que el lugar proporcionado por el usuario es un destino válido como por ejemplo Cartago y no un lugar más general como un supermercado.

Obtención de la ruta, distancia y tiempos utilizando un grafo

Teniendo como base de datos el grafo, es posible calcular cualquier ruta que tenga un origen, un destino y una n cantidad de paradas. Para ello deben poder utilizar tres procedimientos básicos. Poder encontrar todas las rutas de un nodo A a un nodo B. Poder encontrar la ruta más corta de todas las posibles. Poder encontrar la ruta más corta de A a B pasando por n paradas sin que estas formen necesariamente parte de la ruta más corta de A a B. Estas tres partes son identificables. La primera se encuentra en las reglas viajar() y camino(). Estas dos reglas utilizan los hechos arco() del grafo para encontrar recursivamente una ruta de A a B. Para ello hace dos preguntas, ¿es posible ir de A a B? O ¿Existe un nodo C que me lleva a B? Al hacer esta última pregunta recursivamente puede buscar un nodo D que lleva a B o bien una secuencia de nodos que lleve a B. Para el segundo punto lo que se utilizan son las reglas shortest(), min() y minimal(), que mediante setof() analizan todos los resultados de camino(). Haciendo comparaciones entre las distancias totales de cada ruta encontrada se encuentra la ruta más corta en distancia. Finalmente mediante la regla miRuta() se puede encontrar recursivamente la ruta más corta para una lista de destinos ordenados. Para ello se vale de shortest(), entonces para una lista [A,B,C,D] donde A es el origen, D el destino y B y C son paradas no necesariamente en la ruta más corta de A a D, miRuta() calcula la ruta más corta de A a B y luego le une la ruta más corta de B a C y finalmente la ruta de C a D, luego suma todas las distancias y tiempos y así obtiene la ruta final para el usuario.

Problemas sin solución

Escribir signos de puntuación en las respuestas (salvo tildes)

Los signos de puntuación no son reconocidos por el programa como palabras, por lo que cualquier oración que los contenga será automáticamente descartada como oración y dará el error de no haber comprendido la respuesta a la pregunta realizada.

Nombres compuestos inexistentes reconocidos

Como los nombres compuestos sólo se valida si hay un nombre compuesto A al lado de uno B, no importa la combinación. Por lo que “San Verde” será reconocido como un lugar válido. Sin embargo al calcular la ruta no se logrará encontrar ninguna, puesto que el lugar no existe, por lo que el programa responderá que no hay una ruta disponible, pero nunca avisara que es por haber ingresado un lugar invalido, pues desde su perspectiva, es válido.

Parada en origen

Si se pregunta por una ruta de un origen (A) al mismo origen (B) como parada y luego a un destino el programa buscará la ruta más corta de A a A, probando con las aristas de A y luego de A a B. Entonces a modo de ejemplo: suponiendo que C es un nodo que une a A y B, A y B están unidos, B y C no y el peso de A a C es menor que el de A a B. Si se busca la ruta más corta de A a B pasando por A en vez de solucionarlo como A a B, lo soluciona como A C A B.
Cabe mencionar que si A y B son iguales y las paradas distintas el programa da la salida esperada. No así si todos los parámetros son iguales, en ese caso a la salida sería A C A C A .

Plan de actividades

Descripción de la Tarea	Duración Aproximada	Encargado	Fecha de entrega
Definir la estructura que tendrá el grafo.	1h	Jose	24/03/2021
Crear la base de datos del grafo	1h	Jose	24/03/2021
Crear la base de datos de hechos	1h	Jose	24/03/2021
Definir las reglas identificar oraciones	2h	Nati	02/04/2021
Definir las reglas identificar sintagmas nominales	2h	Nati	02/04/2021
Definir las reglas identificar sintagmas verbales	2h	Nati	02/04/2021
Identificar palabras claves	3h	Nati	02/04/2021
Crear oraciones o respuestas conectoras en caso de que no se entienda lo que el usuario está preguntando	1h	Jose	02/04/2021
Creación de interfaz (unión oración con rutas)	5h	Jose y Nati	02/04/2021

Problemas solucionados

Ruta más corta con paradas

Originalmente solo podía conseguir la ruta más corta de A a B sin evaluar ningún tipo de parada. Para resolverlo se crea una regla que utilizando shortest, recursivamente busca y concatena las rutas más cortas de cada lugar en la lista de destinos según el orden en que se da. Por ejemplo: si la lista de destinos es [A,B,C] recursivamente busca la ruta más corta de A a B, luego de B a C y las concatena sumando sus valores de distancia y tiempos, y así se obtiene la ruta más corta de A a C pasando por B.

Forma amigable con el usuario de escribir el lugar específico

Cuando un lugar tiene un nombre formado por más de dos palabras, como por ejemplo Tres Ríos, ocurre que al tomar el input dado por el usuario se toma como un string y para dividirlo se utiliza split_string y el identificador son los espacios por lo que si el nombre del lugar está separado no se identificaba como un lugar válido. Para corregirlo se crea una regla que define que un lugar puede estar compuesto de dos partes llamadas lugar_compuesto(). Además se necesitó crear una regla para unir dos strings como elementos en una lista a un único string separado por un espacio.

Escribir lugares cuyo nombre lleva “ñ” o tilde

Cuando se usa la codificación UTF-8, al realizar cualquier operación para identificarlo como palabra a un string con una ñ o tilde fallará. Por lo que se cambia a UTF-16LE y funciona apropiadamente. Sin embargo es necesario que ambos archivos estén codificados con el mismo protocolo.

Obtener todas las paradas que va a hacer el usuario.

Inicialmente WazeLog le preguntaba al usuario su lugar de inicio, su destino y si iba a realizar alguna parada y formaba una lista con 3 lugares para calcular la ruta pero no había forma de que el usuario especificara si tenía que hacer más de una parada. Esto se resolvió con una regla

aparte que se llama recursivamente hasta que el usuario indique que no tiene que hacer más paradas.

La interacción del usuario y su forma de responder a las preguntas de WazeLog

Aún con la función write para recibir las respuestas del usuario habían problemas ya que al responder había que ponerlo entre comillas y con un punto al final y esto no es muy amigable para el usuario. Para resolver este problema se utilizó la función read_line_to_string para que el usuario no tuviera que poner las comillas ni el punto y además string_lower para que las mayúsculas no afecten el string.

Obtener la palabra clave de una oración dada por el usuario

Anteriormente solo se verificaba que una oración fuera válida con las reglas de oración pero no había una forma de extraer el lugar de la oración. Para resolverlo se cambió la forma de la gramática libre de contexto para incluir lugares en lugar de objetos y pasó de sólo verificar si una oración O era válida a extraer en lugar L con oracion(O,L).

Crear una respuesta adicional en caso de que el usuario no especifique un lugar geográfico o válido.

Inicialmente el usuario tenía que poner un lugar geográfico específico ya que si no WazeLog le decía que no entendía, este es el caso de que el usuario respondiera con un lugar general como farmacia o supermercado. Para poder resolver esto se incluyó una regla que se llama recursivamente en caso que le pidiera al usuario especificar el lugar en vez de poner solo un objeto.

Conclusiones

- Es posible definir grafos mixtos en prolog utilizando hechos simplemente.
- El backtracking de prolog lo hace ideal para encontrar todas las soluciones posibles a un problema, pero sacrifica eficiencia si no se maneja con cuidado.
- El uso del corte es fundamental para ahorrar tiempo de ejecución y evitar el backtracking incensario de prolog.
- El comando trace se puede utilizar para visualizar el flujo del programa paso por paso.
- setof()es una herramienta sumamente poderosa para aprovechar el backtracking de prolog, permitiendo recolectar todas las soluciones encontradas en una lista.
- El paradigma de programación lógico es muy útil en la creación de sistemas expertos.
- Prolog es muy útil para el manejo de objetos y la forma en que se relacionan.
- Se puede utilizar recursividad para el manejo de listas en Prolog.
- Prolog incluye reglas muy útiles para el manejo de strings como split_string y string_lower.
- La codificación UTF-8 provoca problemas con las ñ y las tildes.
- La regla read_line_to_string lee el input y lo transforma a un string.
- Nombrar los hechos, reglas y variables de manera significativa puede facilitar la comprensión del código.
- Se puede construir una gramática libre de contexto en Prolog porque el análisis de la gramática se traduce muy bien en un programa lógico.

Recomendaciones

- Es buena idea utilizar prolog cuando se desean encontrar todas las soluciones posibles a un problema.
- Utilizar setof() cuando se quieren aprovechar varios resultados del backtrack de prolog.
- Utilizar el comando trace para visualizar el flujo del programa paso por paso.
- Tener una clara idea del flujo del programa y el backtrack para así poder utilizar los cortes donde sea importante.
- Nombrar los hechos, reglas y variables de manera significativa para facilitar la comprensión del código.
- Si se va a trabajar con strings investigar previamente acerca de las reglas especiales para operaciones con strings
- Si se va a trabajar con strings en español asegurarse que la codificación utilizada acepte ñ y tildes.
- Crear reglas simples que pueden ser utilizadas varias veces para evitar la repetición de líneas lo máximo posible y optimizar el programa.
- Si se necesita que el usuario escriba una respuesta, la regla de Prolog read_line_to_string permite que se pueda escribir y que al presionar la tecla intro se toma el input como un string.

Bibliografía

anónimo. (2020). append(?List1, ?List2, ?List3). eclipseclp. http://eclipseclp.org/doc/bips/lib/lists/append-3.html

Anónimo. (s.f). 2.1 Recursion and lists. Users York. https://www-users.york.ac.uk/~sjh1/courses/L334css/complete/complete2su6.html

CodePoc. (2021). prolog program to write the elements of the list line by line. CodePoc. https://www.codepoc.io/blog/prolog/5047/prolog-program-to-write-the-elements-of-the-list-line-by-line?utm_source=22

Segers, S. (2015). Recursively create list in Prolog . StackOverflow. https://stackoverflow.com/questions/33977495/recursively-create-list-in-prolog

SWI-Prolog. (s.f). SWI-Prolog Source Documentation Version 2. SWI-Prolog. https://www.swi-prolog.org/pldoc/doc_for?object=section(%27packages/pldoc.html%27)

Toledo, F., Pacheco, J., & Escrig Teresa. (2001). el lenguaje de programación prolog

user27815. (2015). Creating a list from user input with swi-prolog. StackOverflow. https://stackoverflow.com/questions/31710213/creating-a-list-from-user-input-with-swi-prolog