Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Licenciatura en Ciencias de la Computación

Estructura de datos y algoritmos 1

Intérprete de Lista

Juan Cruz Olivero

1. Diseño del programa:

Para realizar el intérprete de listas se crearon distintos archivos qué contienen funciones qué resuelven distintos problemas y se complementan entre sí.
El archivo encargada de iniciar y leer los comandos del intérprete es shell.c qué mediante un fgets lee lo ingresado por el usuario, luego para manejar mejor ese comando eliminamos el \n, también se encarga de inicializar 2 tablas hash, una para guardas lo ingresado con deff y otro para defl, mediante la función strtok obtenemos la primera palabra no nula, donde vemos si esa primera palabra es de algunos de nuestros comandos, en caso de ser deff o defl pasa por una función "verificacion_De_Comando" q lo q hace es q verifica luego de las palabras de deff o defl qué lo ingresado tenga nombre, un = y una definición apropiada, en caso de que no se cumpla esto retorna 0, caso contrario 1.

Cuando uno ingresa deff,qué tiene un archivo deff.c donde están las funciones qué nombraremos , pasa primero por "verificion_de_Comando" qué revisa lo antes dicho luego mediante la función "funcion_deff" qué toma el resto de la palabra por el strtok llamada de un inicio y la tabla hash de funciones, donde primero evaluamos qué el nombre ingresado no sea al de una primitiva pues puede generar conflicto cuando queramos hacer el apply, una vez revisado eso verifica qué la definición sea válida con el IsValidFun qué toma la definición y la tabla de funciones y revisa que la definición sea una primitiva, una función definida en la tabla o algo de la forma <f>, siendo f una función válida, si es una función válida retorna 1, caso contrario 0, luego si es válida la definición insertamos el nombre de la función con su definición en la tabla hash de funciones y retornamos la tabla hash, si no es válida retorna la misma tabla que se ingresó

El comportamiento de defl es similar al de deff,de igual manera hay un archivo defl.c, pues revisamos qué el nombre no se encuentre ya definido, luego revisamos mediante la función Isvalidlis qué verifica que la lista sea una lista válida, osea rechaza listas cómo [,],[,...] , etc, si la lista es válida la agrega a la tabla de listas, caso contrario no agrega nada a la tabla

Luego sigue el comportamiento al utilizar la función apply, qué tiene su archivo apply.c, todo empieza en funcion_apply, qué primero verifica qué el comando ingresado tenga forma válida osea qué seguido de apply vengan una serie de funciones, para luego haber una lista ha aplicar esas funciones, para eso existe

isValidApply, qué dado el comando (ubicado en el primer ‘ ‘ a la derecha de “apply”,las 2 tablas hash, osea la de funciones y de listas y un int* que eso lo usamos para saber en q parte empieza la lista, entonces una vez ingresada a la función ,primero tenemos una variable tipo int llamada lista_valida, inicializada en 0, si vemos qué la lista es válida cambiará a 1, luego nos vamos al final y retrocedemos hasta encontrar un carácter distinto de ‘ ‘
si el carácter es ‘]‘ puede qué sea una lista de la forma [..]
entonces seguimos retrocediendo hasta encontrar un ‘[‘ o un hasta llegar al inicio si llegamos al inicio es porque nunca se abrió un ‘[‘ entonces no es una lista válida, si cierra, guardamos esa posición y pasamos comando + (posición del ‘[’) a la función qué verifica qué la lista tenga forma correcta, osea con Isvalidlis,si lo es cambiamos lista_valida a 1, caso contrario de sí el carácter es distinto de ‘]’, puede ser el nombre de una lista ya definida, entonces retrocedemos hasta encontrar un ‘ ‘ o hasta qué nuestro índice=0, luego de eso copiamos donde terminó el while y donde se encuentra el último carácter distinto de espacio, para mediante la función la función buscar de hash buscar ese nombre, si retorna NULL, es porque no se encuentra en la tabla, si es distinto de NULL, entonces cambiamos lista_valida a 1, una vez analizada la lista, veremos cómo esta la variable lista_valida, si es 0 entonces es porque la lista no lo es, entonces avisamos mediante un printf y retornamos 0, caso contrario nos queda ver las partes de la función, para eso utilizamos una función qué se encuentra en deff, qué es IsvalidFun, que dado un char* y una tabla hash de las funciones definidas verifica que el char* sea una función válida, su funcionamiento es simple va avanzando y si ve un espacio vacio osea un ‘ ‘, sigue avanzando, sino lo guarda hasta qué termine y revisa si es una primitiva (mediante otra función qué dado un char* retorna 1 si es una función primitiva, 0 caso contrario, se llama Isprimitiva)sino la busca en la tabla de funciones y si no está ahí , ve si empieza con ‘<’ si es asi ve si encuentra un ‘>’ qué lo cierre, donde llamamos otra vez a isValidFun pero ahora con lo qué está dentro de < >.
Una vez pasado por IsvalidaApply si está devuelve 0 avisamos qué no es válida y termina ahí, caso contrario procede a crear una lista con un puntero al inicio y otro al final de la lista ingresada por el apply, cómo ya sabemos su ubicación nos desplazamos hacía esa parte, la función se llama pasar_a_lista, su funcionamiento es qué primero buscamos la lista en la tabla, si no se encuentra es porque es de la forma [...], si se encuentra trabajamos con su definición([..]) luego hacemos una copia y trabajamos sobre la copia, utilizamos el strtok para ir obteniendo los números que están en char* para usar el atoi y hacer una copia de estos números e ir poniendo en la última posición de la lista hasta que no haya más números, una vez finalizado retornamos la lista.

Luego aplicamos las funciones a esa lista, usamos aplicar_funcion_a_lista, donde avanzamos sobre estas funciones y vemos qué si es una primitiva aplicamos la función correspondiente , qué se encuentran en el archivo primitivas.c, si está definida obtenemos su definición y llamamos a aplicar_funcion_a_lista con su definición y si es repetición usamos otra función para ir aplicando lo qué se encuentra dentro de los <> hasta qué los extremos sean iguales.
Una vez finalizado esto imprimimos la lista con imprimir_Lista, qué recorre la lista e imprime los números. Ya una vez imprimido liberamos la lista

Por último tenemos el comando search. para este comando se implementaron 2 estructuras, Estado y Arreglo_Dlist, la segunda cómo bien dice su nombre es una estructura donde guardamos array de estructuras de Dlist, con la cantidad de la misma y el tamaño qué se usó en el malloc, se creo está estructura porque me parece más sencillo utilizar esto para representar un array de Dlist, pues cómo hay funciones qué sin está función tomaron cómo argumento un tipo DList***, entonces pensé qué iba a ser más cómodo y sencillo trabajar asi.
Luego Estado, es una estructura qué usaremos para buscar las listas Li2, pues se implementó un algoritmo BFS, y cómo tenia qué tener guardadas tanto las Dlist cómo la cadena que iban formando se creo está estructura para tener los datos más juntos.
Ya habiendo explicado para qué sirven estas 2 estructuras hablaremos de la funcion_search. Primero, creamos 2 estructuras una de tipo Estado* y otra Arreglo_Dlist*, luego pasamos a la función verificar_search, qué verifica que lo ingresado sea válido, osea, que empiece y termine con { }, los pares de listas estén separados por ‘;’ y las listas por ‘,’, mientras vamos revisando estas listas si las listas Li1 son válidas las vamos guardando en la estructura Estado, si las listas Li2 son válidas las guardamos en la Estructura Arreglo_Dlist, una vez finalizado el recorrido, ya vamos a tener verificado los comandos y guardados las listas en sus respectivas estructuras, si resulta no válido, entonces liberamos estas estructuras con sus respectivas funciones , si es válido pasamos a la función buscar_solucion, buscar_solcion toma una estructura de Estado*,un Arreglo_Dlist* y la tabla hash de funciones, cómo bien se especificó sigue un algoritmo BFS, ósea en el Estado* se van ha encontrar las Li1 y un char* qué representa las funciones a las qué fueron aplicadas para llegar a ese estado, y mediante el archivo cola.c, donde se halla toda la implementación de está, primeros encolamos el Estado* pasado cómo argumento, para luego tener la variable int solucion_encontrada=0, luego creamos otra tabla hash, qué es para las listas visitadas, esto es para no repetir estados, por ejemplo:

>>search {[],[1]}; vamos a crear un estado por cada función primitiva y definida, pero observemos qué al aplicar 0i,0d es lo mismo, lo mismo para Sí,Sd,Dd y Di, en esos casos, una vez observado qué ya pasamos por ese estado lo liberamos, pues al haber ya pasado gastamos iteraciones, memoria y tiempo innecesariamente, una vez ya aclarado esto, también declaramos int numero_de_iteraciones=0, qué es qué en caso de no poder encontrar una solución en un rango de iteraciones se detiene y dice qué no encontró soluciones, luego de declarar esa variable tenemos un bucle while qué revisa qué mientras la cola no sea vacia, no se haya encontrado solución (con la variable solucion_encontrada) y el numero_de_iteraciones sea menor a la cantidad máxima, haremos, primero descolamos la cola, a ese dato qué descolamos se llamará estado_actual, donde veremos qué si en el estado actual su Dlist* es igual al de Arreglo_Dlist pasado cómo argumento, esto se verifica mediante la función lista_de_lista_iguales, qué dada 2 DList** , uno siendo las Li1 y la otra las Li2, y el la cantidad de pares de lista ,osea el i, verá qué todos sus elementos son iguales, qué se realiza mediante la funcionlista_iguales, qué verifica qué dadas 2 DList* retorna 1 si son iguales 0 caso contrario. Retomando Si resultan iguales esas 2 DList** significa q encontramos una serie de funciones qué de Li1 pudimos llegar a Li2, entonces imprimimos esa serie, qué se encuentra guardada en estado_actual, para luego cambiar la variable solucion_encontrada a 1, para salir del ciclo, caso contrario, Si las 2 Dlist** no son iguales entonces vamos a pasar a nuevos estados, estos nuevos estados van a ser nuestro estado_actual habiendo aplicado una función definida o una primitiva, para eso hacemos una copia del estado, y primero recorremos la tabla hash de las funciones definidas, pues quizá haya funciones qué nos acerquen más a Li2, una vez aplicada la funcion sobre el estado_actual buscaremos en la tabla hash de Listas_visitadas, para ver si ya pasamos por esa lista, pues la clave y definicion de la misma es la secuencia de lista en string,(ej:"[0,1][1,4]") entonces pasamos las listas Li1 ha string y vemos si se encuentran en la tabla, si se encuentran destruyo ese estado, pues cómo ya pase no quiero pasar por lo mismo, si no está en la tabla, agrego la secuencia de lista en string ha la tabla de listas_visitadas para luego encololar el nuevo estado qué tenemos, una vez aplicado el estado actual sobre las funciones definidas nos queda aplicar las primitivas, donde se sigue el mismo procedimiento, una vez ya echo esto destruye el estado actual y sumo una a numero_de_iteraciones, una vez qué salgamos del bucle, nos queda verificar que la cola qué usamos estuvo vacia, si lo está liberamos la cola misma, sino mientras no este vacia desacolamos y destruimos los estados, ya liberada la cola liberamos la estructura Arreglo_Dlist y la tabla hash utilizada para ver las listas pasadas.

2. Estructuras de datos utilizadas:

Cómo se podrá notar, utilizamos diversas estructuras de datos mediante el transcurso del intérprete, y está sección está dedicada para la justificación de su uso:

Estructura DList: Es una estructura de lista con un puntero al inicio y al final, está estructura la utilizamos principalmente en la seccion de apply.c, esto debido a qué cuando pasamos la lista en string a Dlist, cómo las funciones qué hay para aplicar se realizan en los extremos Izquierdo y derecho, con u puntero al inicio y al final podemos acceder rápido a los datos para aplicar estas funciones y evitar recorrer datos innecesarios.

Estructura Cola: Cómo bien dice su nombre es la estructura de la cola, se utiliza principalmente en la sección de search más específico a la hora de tener qué buscar una función para llegar de Li1 a Li2. Elegimos cola pues el algoritmo BFS osea búsqueda por niveles requiere de cola, y se optó por este algoritmo debido a qué busca el camino más corto para el destino y su implementación no es tan compleja y es sencilla de comprender.

Estructura Tabla Hash: Se utiliza en diversas partes, la utilizamos para guardar las funciones y listas definidas con deff y defl, también en la sección del search la utilizamos para evitar pasar por mismos estados,evitando así consumir memoria y recurso de manera innecesaria. Es de gran utilidad pues cómo las funciones y listas definidas tienen qué tener nombre, solamente tendremos qué implementar una buena función de hasheo para así poder guardar y acceder a los elementos de manera veloz.

Estructura Arreglo_Dlist: Surge en la sección de search y debido a qué al querer verificar e ir guardando información de este para así evitar recorrer nuevamente lo ingresado por usuario, para no tener tantos punteros pensé en tener directamente una estructura qué almacene estos datos qué quería ir guardando sumado a su longitud.

Estructura Estado: De manera similar a Arreglo_Dlist, Estado surge de la misma manera, ir guardando información para no recorrer muchas veces lo ingresado, también al ser estructura brinda más comodidad a la hora de formar la serie de funciones.