INFORME N° 2

para

Creación de Analizador Sintáctico LMA

INFO 165

Autores:
Ricardo Coronado
Eduardo Hopperdietzel
Víctor Moya

Docente:
María Eliana de la Maza W.

Universidad Austral de Chile
9 de noviembre, 2020.

1. Introducción

El presente documento es un pequeño ejemplo de un traductor capaz de ejecutar programas escritos en LMA (Lenguaje Manejador de Arreglos). El traductor acepta una secuencia de instrucciones del lenguaje LMA y efectúa algunas acciones a medida que se realiza el análisis sintáctico.

Para su implementación se utilizaron las herramientas Flex y GNU Bison.
Flex es un generador de analizadores léxicos (o Scanner), mientras que GNU Bison es un generador de analizadores sintácticos (o Parser) que convierte una gramática dada en un programa en C que analiza dicha gramática. Finalmente, su implementación conjunta permite construir compiladores de lenguajes.

El lenguaje manejador de arreglos (o LMA) es infinito, esto porque puede generar una infinidad de palabras. Está compuesto por una secuencia de 8 elementos, donde cada elemento puede ser un número entero mayor a cero. También, puesto que existe una gramática de libre contexto capaz de generar al lenguaje, decimos que el lenguaje es libre de contexto.

2. Desarrollo

Para la construcción de un traductor dirigido por sintaxis que permite ejecutar programas escritos en el lenguaje LMA se utilizó Flex y GNU Bison, con estos generamos dos archivos, uno de nombre *ALM.l,* escrito en Flex y encargado del análisis léxico, y otro de nombre *ALM.y*, escrito en Bison y encargado del análisis sintáctico del lenguaje.

2.1. Formato de Palabras Reservadas

A diferencia de nuestra gramática inicial, cambiamos el formato de las palabras reservadas a letras minúsculas sin acentuación.

2.2. Sintaxis Nombre de Variable de Arreglos

Las variables deben comenzar con la letra L (mayúscula), seguido de una combinación de cero o más letras (A -Z) mayúsculas o minúsculas sin acentuación, finalizadas por un número entero positivo. El largo total no debe superar los 63 caracteres.

2.3. Gramática

En primer lugar, fue necesario crear una gramática capaz de permitirnos modelar lógicamente el lenguaje. La gramática creada es la siguiente:

PROG → PARTIR INSTS FINALIZAR

INSTS → INST_INICIAR INSTS |
INST_METER INSTS |
INST_MIRAR INSTS |
INST_SACAR INSTS |
INST_DATO INSTS | ε

INST_INICIAR → INICIAR(VAR, NUM, NUM , NUM, NUM, NUM, NUM, NUM, NUM)
INST_METER → METER(VAR, NUM, INDEX)
INST_MIRAR → MIRAR(VAR)
INST_SACAR → SACAR(VAR, INDEX)
INST_DATO → DATO(VAR, INDEX)

Donde:
NUM: Número entero positivo
VAR: L seguido de una o más letras y finalizado con un número entero positivo.
INDEX: Número natural en el intervalo [1,8].

Sin embargo al implementarla en Bison, nos dimos cuenta de que su estructura no nos permitía analizar las entradas inmediatamente, se debía esperar a que el usuario ingrese “finalizar” para recién mostrar los resultados. Esto se debía a problemas de recursividad.

Por lo tanto con unas cuantas modificaciones obtuvimos la siguiente gramática:

programa → T_PARTIR T_SALTO instrucciones T_FINALIZAR T_SALTO
instrucciones → instruccion instrucciones | ε
instruccion → T_INICIAR T_L T_VAR T_COMA entero T_COMA entero … T_COMA entero T_R T_SALTO
| T_METER T_L T_VAR T_COMA entero T_COMA T_INDICE T_R T_SALTO
| T_MIRAR T_L T_VAR T_R T_SALTO
| T_SACAR T_L T_VAR T_COMA T_INDICE T_R T_SALTO
| T_DATO T_L T_VAR T_COMA T_INDICE T_R T_SALTO

entero → T_INT | T_INDICE

2.4. Estructura de Datos

Definimos un struct para almacenar los arreglos del usuario como una lista enlazada. Sus parámetros son:

nombre: Arreglo de tamaño 64 de tipo char para almacenar el nombre de variable de un arreglo.
números: Arreglo de tamaño 8 de tipo int para almacenar los datos del arreglo.
next: Puntero de tipo struct Arreglo * para almacenar el arreglo siguiente.

2.5. Métodos

Implementamos una serie de métodos para simplificar el análisis y el manejo de datos.

int arregloEstaLleno(int *arr)
Este método retorna 1 si el arreglo del argumento está lleno y 0 en caso contrario.

struct Arreglo *buscaArreglo(char*nombre)
Este método busca un arreglo en la lista enlazada que tenga el mismo nombre del argumento. Si lo encuentra, retorna el puntero a este, en caso contrario retorna NULL.

void dato(char *nombre, int pos)
Este método, busca en la lista enlazada el arreglo con el nombre del argumento, si lo encuentra, imprime el valor ubicado en el índice “pos” de su arreglo de números.

void imprimirArreglo(char *nombre)
Este método, busca en la lista enlazada el arreglo con el nombre del argumento, si lo encuentra imprime los valores distintos de 0 de su arreglo de números.

void comprimeArreglo(int *arr)
Este método mueve los elementos distintos de 0 a la izquierda del arreglo de números del argumento.

void sacarElemento(char *nombre,int pos)
Este método, busca en la lista enlazada el arreglo con el nombre del argumento, si lo encuentra asigna 0 al valor en el índice “pos” de su arreglo de números. Luego llama al método **comprimeArreglo()**.

void meterElemento(char*nombre,int valor,int pos)
Este método, busca en la lista enlazada el arreglo con el nombre del argumento, si lo encuentra, inserta el valor en la posición “pos” de su arreglo de números sin eliminar el valor actual.

void iniciarArreglo(char *nombre,int n1,int n2,int n3,int n4,int n5,int n6,int n7,int n8)
Este método añade un nuevo arreglo a la lista enlazada, asignándole el nombre del argumento, y completando su arreglo de números con las variables n1, n2, …, n8.

3. Especificaciones del programa

3.1. Acciones del traductor

- ➔ partir: Primera instrucción. Inicia el programa.
- ➔ iniciar(A, e1,...,e8): Crea un arreglo de nombre A con los elementos e1,...,e8 (números enteros positivos). En caso de agregar menos de ocho números, los restantes serán completados con ceros.
- ➔ meter(A, x, y): Inserta el elemento x en la posición y del arreglo A.
- ➔ sacar(A, y): Elimina el elemento en la posición y del arreglo A.
- ➔ mirar(A): Despliega los elementos del arreglo A.
- ➔ dato(A, y): despliega el elemento que se encuentra en la posición y del arreglo A.
- ➔ finalizar: Última instrucción. Finaliza el programa.

3.2. Compilación

Para compilar el programa usando `Makefile`

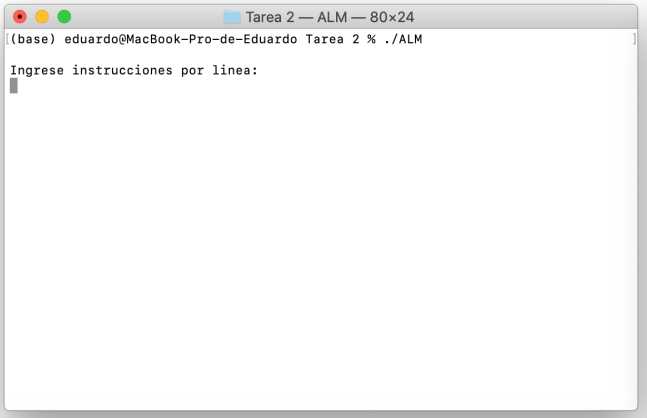
```
$ make
```

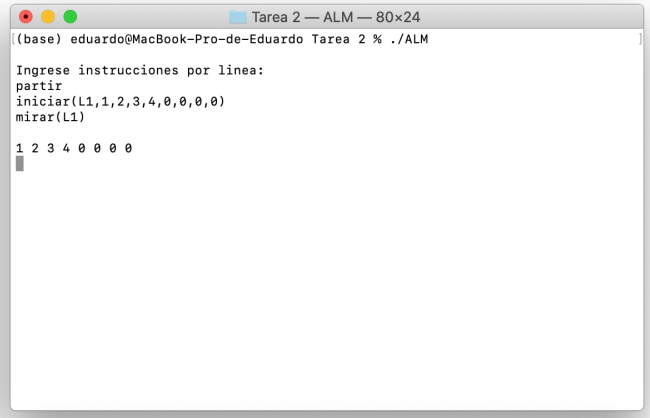
O manualmente en Unix, siguiendo los siguientes pasos:

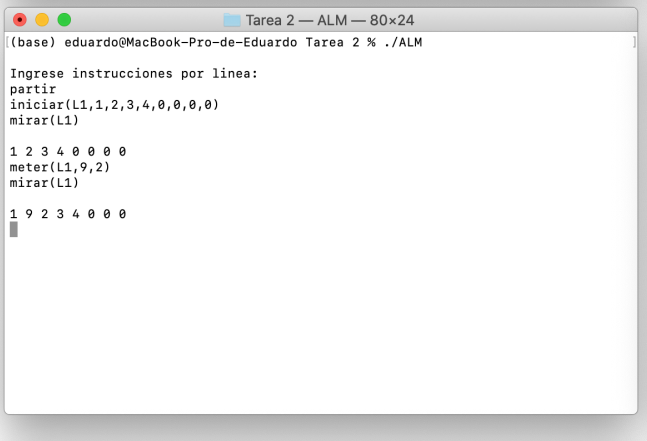
```
$ bison -d ALM.y
$ flex ALM.l
$ gcc ALM.tab.c lex.yy.c -o ALM -lm
$ ./ALM
```

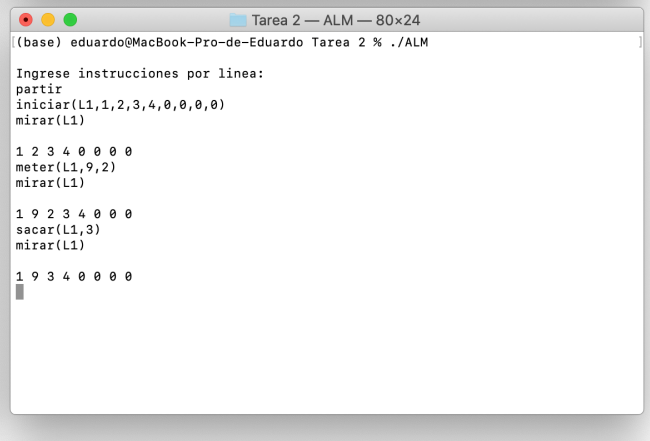
Una vez compilado el traductor, además del archivo ejecutable ALM, se creará el archivo ALM.output, este contiene la tabla de análisis sintáctico del programa y sus respectivos estados.

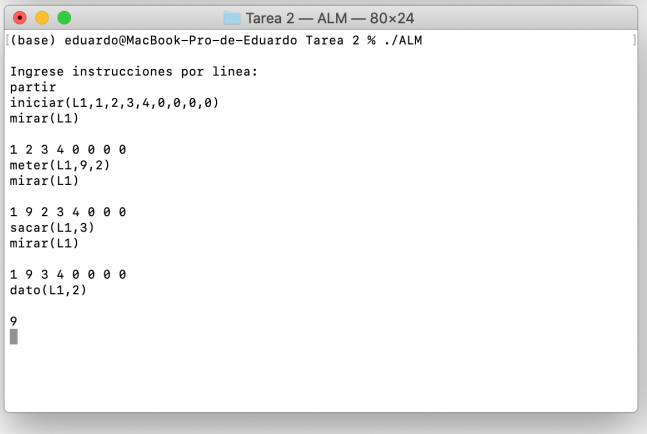
3.3. Ejemplo de uso

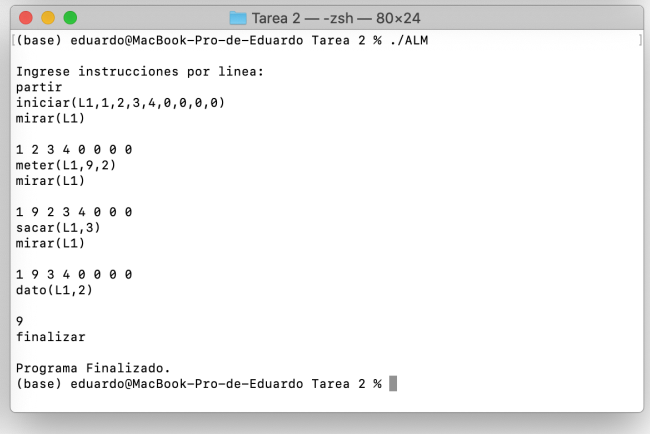












4. Conclusiones

La construcción del traductor dirigido por sintaxis fue una experiencia muy provechosa para nosotros, pues pudimos visualizar de forma directa las etapas y procesos que realiza un computador para interpretar un lenguaje de programación, principalmente en la construcción de un programa.

El mayor problema o dificultad que tuvimos fue que al crear la gramática no nos permitía analizar las entradas inmediatamente, dado a un problema de recursividad.

Afortunadamente después de estudiar y buscar soluciones a estos problemas logramos construir el analizador dirigido por sintaxis.

Una de las posibles mejoras que harían de nuestro traductor un producto de software más óptimo e interesante sería, en primera instancia, que permita el ingreso al usuario de números tanto positivos como negativos, o punto flotantes. También que presente acciones que ejecuten algoritmos de ordenamiento y búsqueda, y que una vez realizada la tarea informe sobre el tiempo de ejecución.

5. Bibliografía

- Ejemplo Flex & Bison. (s. f.). Github/meyerd. Recuperado 9 de diciembre de 2020, de http://github.com/meyerd/flex-bison-example

- Obtención de tablas de análisis sintáctico, GNU manual Bison. Recuperado 9 de diciembre de 2020, de https://www.gnu.org/software/bison/manual/html_node/LR-Table-Construction.html

- Dpto. Informática e Ingeniería de Sistemas Universidad de Zaragoza, & Béjar Hernández, R. (2011). Introducción a Flex y Bison. http://webdiis.unizar.es/asignaturas/TC/wp/wp-content/uploads/2011/09/Intro_Flex_Bison.pdf