Informe Tarea 2

Integrantes:
Leandro Calloguerea
Felipe Soto
Diego Rojas

Asignatura:
Compiladores (INFO165)

Docente:
María Eliana de la Maza Werner

Universidad Austral de Chile
10 de diciembre 2020

Índice

Introducción

En esta segunda entrega se nos propone completar traductor dirigido por sintaxis construido en la primera parte del trabajo, donde el desarrollo del mismo no solo presenta un desafío en cuanto a los lenguajes y/o herramientas elegidas sino que también en la forma en que aprovechamos las experiencias colectivas de quienes componen el grupo de trabajo. El trabajo consiste en extender y agregar las funciones asociadas a cada token antes registrado en nuestra gramática con el objetivo de poder hacer un uso eficiente de la herramienta y poder completar su funcionamiento a cabalidad.

Desarrollo

Para el desarrollo del trabajo contamos con la gramática especificada en [Anexo], la cual nos proporcionó la base para la implementación del algoritmo. Dicha gramática fue implementada en el archivo bison LMA. Este proceso no resultó tan difícil como anticipamos debido a la excelente documentación de bison.

Uno de los primeros problemas que tuvimos fue al conectar flex con bison, ya que el analizador léxico que teníamos de la entrega anterior, a pesar de que funcionaba correctamente no nos servía tal como estaba, porque muchas funcionalidades ahora debían hacerse en el archivo bison, por lo que tuvimos que pasar parte de las funciones hechas en lex a bison. Otro problema con el que nos encontramos y podemos decir que fue uno de los que nos quitó más tiempo, fue pasar el valor de los enteros e identificadores desde lex hacia bison, lo cual logramos hacer gracias a la función %union de bison, que permite asignar un tipo de dato a los terminales que lo requieran, lo que en este caso fue “int” para el token ENTERO y “char *” para el token ID.
Una vez que logramos pasar los enteros e identificadores correctamente, el siguiente desafío fue almacenar los datos ingresados, ya que bison no nos dejaba almacenarlos, creamos un archivo arreglo.h, donde definimos una estructura de datos que almacenaba las variables como se ve en la siguiente imagen.

```
typedef struct arreglo{
    char *id;
    int arr[8];
}arreglo;
```

Esta estructura almacena cada arreglo con su respectivo identificador asociado.

Para poder almacenar varios arreglos, definimos dentro del archivo arreglo.h un arreglo de arreglos, como se muestra a continuación:

```
arreglo *aux[L];
```

Donde L es una constante definida al principio, que dejamos con el valor 15. También definimos otra variable global de tipo entero llamada largo, que inicializamos en 0 y va sumando 1 cada vez que se agrega un nuevo arreglo, esta variable es importante, ya que nos permite iterar entre todos los arreglos inicializados a la vez que nos dice cuántos son.
Una vez que implementamos esto el resto se hizo relativamente simple, ya que solo aplicamos el conocimiento y experiencia que tenemos en C para modificar y recorrer arreglos.
Para encontrar un arreglo por nombre, creamos la función:

```
int buscar(char *id){
    for(int i=0; i<largo;i++){
        if(strcmp(aux[i]->id,id)==0){
            return i;
        }
    }
    return -1;
}
```

que es llamada por las funciones meter, sacar, mirar y dato. Esta función retorna el índice del arreglo aux, que corresponde al arreglo con el nombre id.

También implementamos la función:

```
void ordenar(int index){
    int ceros = 0;
    int arr2[8]={0,0,0,0,0,0,0,0};
    for(int i=0; i<8; i++){
        if(aux[index]->arr[i]==0){
            ceros++;
        }
        else
        {
            arr2[i-ceros] = aux[index]->arr[i];
        }
    }
    for(int i=0; i<8; i++){
        aux[index]->arr[i] = arr2[i];
    }

}
```

para ordenar las listas, dejando todos los ceros a la derecha.

Métodos ocupados:

iniciar(id,e1,e2,e3,e4,e5,e6,e7,e8) = método cuál instrucción a través de la gramática inst_iniciar->iniciar(id,..,entero), con palabra reservada como token INICIAR, donde válida con método buscar() en primera instancia si la lista con "id", se encuentra ingresada, para posteriormente registrar en tabla "hash".

ordenar(index) = Método que permite ordenar la lista de elementos ingresados de manera correlativa, incorporado "0" al final de dichos espacios vacíos que quedan, tras procesos de "iniciar", "meter" o "sacar".

meter(id,val,pos) = Función que ejecuta a través de la gramática inst_meter->meter(id,entero,entero), cual operación es a través de la lista existente "id" en la tabla "hash", con valor "val", posicionarse en "pos" de dicha lista.

mirar(id) = Operación cual permite revisar la lista "id" con sus elementos guardados y dicha lista registrada en la tabla "hash".

sacar(id,pos) = Método que nos permite buscar en la lista "id" registrada en tabla, quitar un elemento "pos" guardado en lista.

Especificación del Programa

En la definición del programa, tenemos los archivos fuentes cuales son Tarea1.l , LMA.y y Makefile, cuales forman parte de la columna vertebral en la compilación.

Requerimientos básicos de ejecución:

Los requerimientos de software son:

- Instalada herramientas de compilación **"c++"**
- Instalado el paquete "**bison++"**
- Sistema Operativo compatible, Unix, Windows, MacOS.

Requerimientos de Hardware:

- 200mhz en procesador
- 64mb. de RAM
- 700mb Almacenamiento Base
- 2 bits Procesamiento gráficos, para leer texto, plano

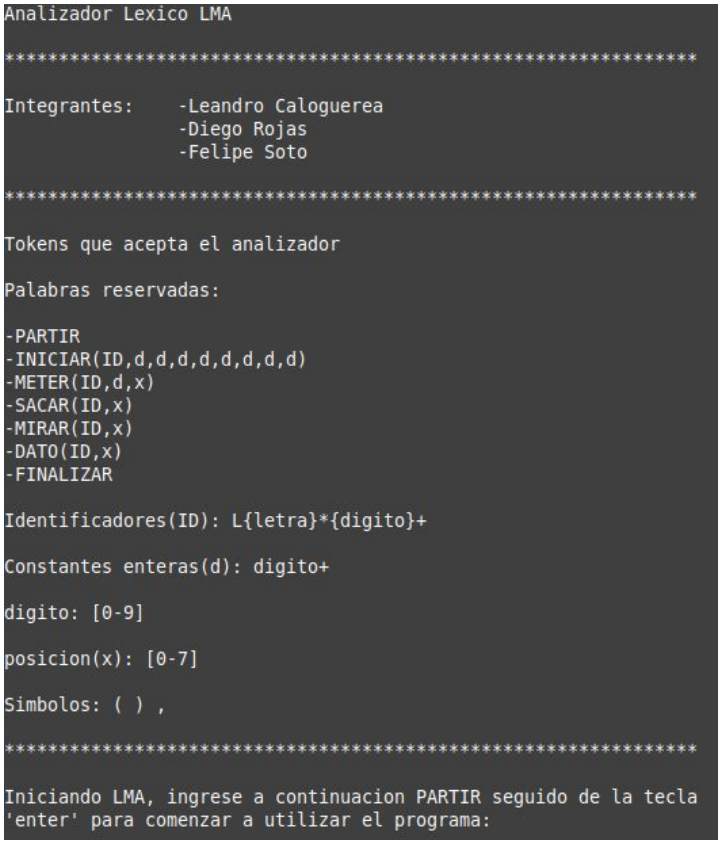
Pruebas Básicas - Compilación

Para el proceso de compilación en Linux/Windows se utilizan los siguientes comandos en la locación fuente de los archivos:

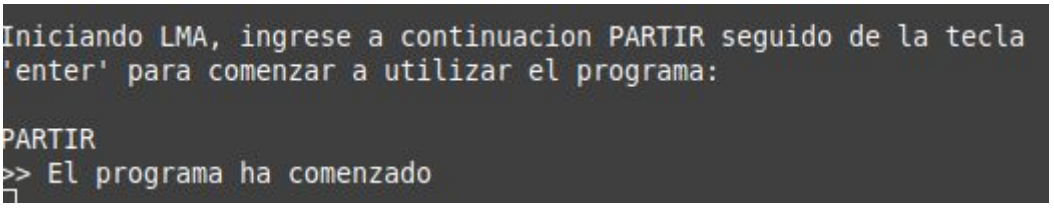
$bison -yd LMA.y para compilar nuestro analizador sintáctico
$flex Tarea1.l para compilar nuestro analizador léxico
$gcc y.tab.c lex.yy.c -lfl -o salida donde finalmente nos entrega el ejecutable "salida.exe"

Pruebas Básicas - Ejecución

En la locación fuente del directorio, posterior a la compilación ejecutamos el archivo "salida.exe", cual pantalla comienza comienza de la siguiente forma:



En lo práctico, mostramos los comandos básicos cuales permiten la interacción inmediata con el usuario con algunos manejos para los errores, para ello es necesario que se inicie con el comando "PARTIR":



Podemos ingresar una lista, con sus respectivos números enteros de registro:

```
INICIAR(Ls2,1,2,3,4,5,6,7,8)
>> Identificador Ls2 ha sido inicializado
```

Podemos mirar que se encuentra guardada la lista:

```
MIRAR(Ls2)
>> 1 2 3 4 5 6 7 8
```

Por último podemos terminar el programa con el comando “FINALIZAR”

Conclusiones

incluyendo mejoras y limitaciones, si existiesen

Consideramos que si bien el proceso para poder construir y completar esta herramienta era implícitamente claro, nos fuimos encontrando en el camino con una serie de obstáculos y desafíos que fueron poniendo a prueba nuestras habilidades y capacidad de documentación, entre ellas volver a conocer el alcance que posee C en cuanto a su estructura y manejo de datos, es aquí donde presentamos la mayor obstrucción ya que la lógica referente a cómo manejar los datos era clara para todos los miembros del equipo. Una vez resuelto este bloqueo el proceso fue bastante expedito donde las labores fueron bien abordadas y distribuidas para así poder maximizar los avances.

Consideramos que este trabajo en particular nos ayuda a interiorizar los conceptos teóricos vistos en clases asociados a la asignatura, donde podemos visualizar con mayor perspectiva el cómo se manejan los tokens, las tablas de símbolos y la utilización del árbol que genera la gramática definida inicialmente de manera implícita.

Mejoras:

1. Hemos dispuesto los respectivos mensajes del sistema para cada acción con el objetivo de mejorar la experiencia de usuario, puesto que para algunas instrucciones el prompt del programa no daba mayor indicio de si este se había ejecutado correctamente o no.
2. Realizamos una serie verificaciones extras para asegurar el correcto funcionamiento del programa, las cuales son:
    a. Límite de intervalo para instrucciones MIRAR, DATO, SACAR y METER donde se requiere un valor x referente a la posición a trabajar. para este caso si el valor no se encuentra entre 1 a 8 retorna un mensaje indicando el problema y solicitando la

corrección.

```
Iniciando LMA, ingrese a continuacion PARTIR seguido de la tecla
'enter' para comenzar a utilizar el programa:

PARTIR
>> El programa ha comenzado
INICIAR(L1,1,2,3,4,5,6,7,8)
>> Identificador L1 ha sido inicializado
DATO(L1,10)
>> La posicion 10 no pertenece al intervalo correcto de indices
del arreglo. Debe ser entre 1 a 8, intente nuevamente...
```

b. Se agrega un validador de si el identificador ya ha sido registrado/iniciado previamente.

```
Iniciando LMA, ingrese a continuacion PARTIR seguido de la tecla
'enter' para comenzar a utilizar el programa:

PARTIR
>> El programa ha comenzado
INICIAR(L1,1,2,3,4,5,6,7,8)
>> Identificador L1 ha sido inicializado
INICIAR(L1,1,2,1,2,1,2,1,2)
>> El identificador L1 ya se encuentra en los registros
INICIAR(L2,1,2,3,1,2,3,1,2)
>> Identificador L2 ha sido inicializado
```

c. Si el identificador usado en alguna instrucción no existe, se despliega el siguiente mensaje.

```
****************************************************************

Iniciando LMA, ingrese a continuacion PARTIR seguido de la tecla
'enter' para comenzar a utilizar el programa:

PARTIR
>> El programa ha comenzado
INICIAR(L1,1,2,3,4,5,6,7,8)
>> Identificador L1 ha sido inicializado
INICIAR(L1,1,2,1,2,1,2,1,2)
>> El identificador L1 ya se encuentra en los registros
INICIAR(L2,1,2,3,1,2,3,1,2)
>> Identificador L2 ha sido inicializado
MIRAR(L3)
>> El identificador L3 no se encuentra en los registros
```

d. Se agregan las palabras reservadas a la interfaz para que el usuario pueda ver el uso correcto de cada una.

Limitaciones:

1.- Debido a conflictos de sistemas decidimos eliminar las puntuaciones de los mensajes del programa para reducir problemas de legibilidad en la interacción con el usuario.

2.- Si bien el programa detecta correctamente cada error de ejecución referente al mal uso de las palabras reservadas, no nos fue posible poder asociar el evento capturado por *yyerror(s)* para poder asociarlo a una función personalizada y así tratar cada incidencia de error de forma apropiada.

Bibliografía

https://crysol.org/recipe/2007-12-09/creacin-de-un-parser-con-flex-y-bison-en-c.html#.X8F61GNR202

https://aquamentus.com/flex_bison.html

https://www.oreilly.com/library/view/flex-bison/9780596805418/ch01.html

https://berthub.eu/lex-yacc/cvs/lex-yacc-howto.html

https://starbeamrainbowlabs.com/blog/article.php?article=posts%2F267-Compilers-101.html

https://www.gnu.org/software/bison/manual/

https://en.wikipedia.org/wiki/GNU_Bison

http://www.admb-project.org/tools/flex/compiler.pdf

http://www.cs.buap.mx/~andrex/parser/Intro_Flex_Bison.pdf

Referencias

http://dinosaur.compilertools.net/bison/bison_5.html

https://www.daniweb.com/programming/software-development/threads/280247/help-with-compiler-design-in-c-using-flex-bison

https://www.cs.uic.edu/~spopuri/cparser.html

https://gnuu.org/2009/09/18/writing-your-own-toy-compiler/

https://docs.google.com/document/d/17i9E_UKLqBrMKuqwqMe-Q5KzUBjttXCt/edit

https://drive.google.com/drive/u/0/folders/1G0rs59vZ06VdI4oCOVNxhsIoep4xq1YZ

Anexo

Gramática:

A -> S
S -> Partir NL inst Finalizar
inst -> inst_inicio inst' | inst_meter inst' | inst_sacar inst' | inst_mirar inst' | inst_dato inst'
inst' -> NL inst inst' | ε
inst_iniciar -> iniciar(id, entero, entero, entero, entero, entero, entero, entero, entero)
inst_meter -> meter(id,entero, entero)
inst_sacar -> sacar(id,entero)
inst_mirar -> mirar(id)
inst_dato -> dato(id,entero)
id -> L digito
entero -> digito | digito digito

Procesamiento de entradas:

Prim(A)= {PARTIR}
Prim(S) = {PARTIR}
Prim(inst) = {iniciar, meter, sacar, mirar, dato}
Prim(inst') = {NL, ε}
Prim(inst_iniciar)= {iniciar}

Prim(inst_meter)= {meter}
Prim(inst_sacar)= {sacar}
Prim(inst_mirar)= {mirar}
Prim(inst_dato)= {dato}
Prim(id)= {L}
Prim(entero) = {entero}

sig(A)={$}
sig(S)=sig(A) = {$}
sig(inst)= (Prim(inst')-{ε}) ∪ {NL} = {NL}
sig(inst')=sig(inst) ∪ sig(inst') = {NL}
sig(inst_iniciar)= (Prim(inst')-{ε}) ∪ sig(inst) = {NL}
sig(inst_meter)= (Prim(inst')-{ε}) ∪ sig(inst) = {NL}
sig(inst_sacar)= (Prim(inst')-{ε}) ∪ sig(inst) = {NL}
sig(inst_sacar)= (Prim(inst')-{ε}) ∪ sig(inst) = {NL}
sig(inst_dato)= (Prim(inst')-{ε}) ∪ sig(inst) = {NL}
sig(id)= {“,” , “)”}
sig(entera)= {“,” , “)”}

Tabla de Análisis Sintáctico

	Partir	iniciar	meter	sacar	mirar	dato	L	NL	dígito	ε
A	A→ S									
S	S -> Partir NL inst Finalizar									
inst		inst -> inst_iniciar inst'	inst -> inst_meter inst'	inst -> inst_sacar inst'	inst -> inst_mirar inst'	inst -> inst_dato inst'				
inst'								inst -> NL inst inst' inst' -> ε		inst' -> ε
inst_iniciar		inst_iniciar -> iniciar(id,....,entero)								
inst_meter			inst_meter(id,entero,entero)							
inst_sacar				inst_sacar(id,entero)						
inst_mirar					inst_mirar -> mirar(id)					
inst_dato						inst_dato -> dato(id,entero)				
id							id->L digito			