TPE TLAC

Trabajo Práctico Especial

Primer Cuatrimestre 2021

Grupo: “C-Graph”

Integrantes

Francisco Quesada - Legajo 60524

Octavio Serpe - Legajo 60076

Segundo Espina - Legajo 60150

Gastón Gregorio Donikian - Legajo 60067

Índice

Idea subyacente y objetivo del lenguaje **3**

Consideraciones realizadas (no previstas en el enunciado) **3**

Descripción del Desarrollo del TP **3**

Descripción de la gramática **4**

Palabras reservadas del lenguaje: 4

Snippet de código en lenguaje C-Graph: 4

Dificultades encontradas **5**

Futuras extensiones **6**

Referencias **6**

Anexo **6**

Gramática en formato BNF 6

Idea subyacente y objetivo del lenguaje

El lenguaje desarrollado se llama CGraph (compila archivos nombrearchivo.cg, es decir, con extensión .cg). Este busca simplificar el manejo de grafos proponiendo un lenguaje orientado a grafos. Pudiendo agregar nodos, aristas, recorrer estos grafos de diferentes maneras en tan solo una línea, y eliminar tanto nodos como aristas de una manera sencilla y amigable. De esta manera el usuario no debe saber cómo implementar un grafo en C ni tampoco los algoritmos que recorren a este, para poder trabajar con ellos. Por otro lado, se reduce significativamente la cantidad de líneas escritas y ahorra el manejo de heap al usuario.

Consideraciones realizadas (no previstas en el enunciado)

El compilador implementado compila el archivo ingresado y lo traduce al lenguaje C. Algunas consideraciones para el usuario son que con la función output puede imprimir strings, ints, y constantes. Mientras tanto, la función input solo trabaja con strings, y tampoco hay una función que sea intToString, por ende un programa .cg no puede recibir por input un valor numérico e intentar utilizarlo como integer para utilizar luego.

Además de lo pedido en el enunciado se creó una librería de grafos para poder realizar las funciones más propias del lenguaje.

Descripción del Desarrollo del TP

El desarrollo del TP se podría dividir en tres partes. Una primera parte donde definimos el lenguaje que se quería implementar (la idea) y se creó la librería de grafos por separado para luego unir con nuestro lenguaje. Una segunda parte donde se crea toda la lógica por atras del compilador, entre ellas, la gramática el árbol de código (AST), el scope de variables entre otras cosas. Finalmente, se une ese código con el de la librería de grafos.

Se utiliza lex para poder identificar los tokens de nuestro lenguaje definido en la entrada al compilador. Se utiliza yacc para establecer las reglas de la gramática y parsear los tokens en función de éstas. El yacc también está conectado a otro archivo llamado AST.c y translateAST.c, donde el primero crea el árbol de compilación para luego ser recorrido para genera la salida del compilador, y el segundo se encarga de “traducir” cada nodo del AST en un código válido en C.

Oportunamente, incluimos 5 programas para que se pueda verificar el correcto funcionamiento del compilador:

1. **Hello World.cg:** Imprime hello world recorriendo un grafo.
2. **factorial.cg**: Calcula el factorial de un número natural.
3. **fibonacci.cg:** Calcula un valor de la serie fibonacci.
4. **ifElseIfElse.cg**: Un testeo intenso del funcionamiento del if, else-if y else.
5. **strings.cg**: Utiliza input/output intensamente.
6. **echo.cg:** Recibe por entrada estándar y devuelve por salida estándar.
7. **traversingGraph.cg**: Itera por el grafo de muchas maneras distintas, con comentarios donde expresa lo que se realiza paso a paso y “predice” el output. Es el ejemplo más completo de nuestro lenguaje.

Descripción de la gramática

La gramática acepta tipos de datos de entero (int), string y nodo. Cada instrucción se separa por el delimitador ‘;, y se la trata individualmente como un “statement”’. El archivo a compilar debe comenzar con una función de nombre graph que actua como entrypoint, sin parámetro alguno. Un ejemplo sería:

```
graph() {código a compilar}
```

Soporta asignación, operaciones aritméticas como suma, resta, multiplicación y división. Las operaciones booleanas funcionan con 1 como true y 0 como false, también tienen conectores and, or, not (&&, ||, !, respectivamente), mayor, menor, mayor o igual, menor o igual, igual y distinto (>, <, >=, <=, ==, !=) . Soporta bloques if, else-if y else. También soporta for, y foreach. Tiene mecanismo de input y output que utilizan fgets y printf respectivamente.

Para los grafos se pueden agregar y quitar nodos y aristas entre estos. También se pueden recorrer con el for y el foreach. Además se pueden recorrer con algoritmos específicos de los grafos como BFS y DFS.
La definición de la gramática se encuentra en el anexo.

Palabras reservadas del lenguaje:

int, for, input, output, else-if, if, else, string, node, edge, foreach, Node, in, traverse, starting,with, dfs, bfs.

Snippet de código en lenguaje C-Graph:

graph(){

@ Esto es un comentario. Comienza con @ y termina con \n
@ declaración de los nodos del grafo con su respectivo valor
Node node1 = “nodo1”, node2=“nodo2”, node3=“nodo3”, node4=“nodo4”, node5=“nodo5”;
@ node1 va a node2 con peso 'h'
node1-”h”->node2;
@ node2 va a node3 con peso 'o'
node2--”o”->node3;
node4--”a”->node5;
node5--” ”->node1;

@ se elimina la arista que va de node4 a node5
node4--/->node5;
@ destrucción del node5
/> node5;
@ Iteración por de los nodos del grafo con orden arbitrario

```
        foreach(node in graph){
                @ code
                @Iteración por las aristas del iterador
                foreach(edge in node){
                        @ obtiene el peso de la arista actual, podría accederse con edge
                        @ a secas
                        edge.weight;
                }
        }



        @Foreach de los nodos del grafo con orden dfs arrancando de node0
        traverse(graph with dfs starting node0);

        @Foreach de los nodos del grafo con orden bfs arrancando de node0
        traverse(graph with bfs starting node0);
}
```

Dificultades encontradas

La primer dificultad fue armar la gramática. Todas las ideas que surgieron de posibles gramáticas caían en una de dos posibilidades; o eran más similares a una librería que a un propio lenguaje o eran muy ambiciosas y no estaba dentro del scope de la materia. Por otro lado, la misma se modificó hasta el último momento previo a la entrega, dado que había algo por mejorar, o se proponía otro enfoque.

La segunda dificultad fue en el armado de la librería de grafos. Gracias a que se cursó previamente EDA, se conocía la implementación de esto en Java, pero al pasar la implementación a C ocurrieron dificultades menores, debido a que se carecía de la abstracción a nivel objeto, por lo tanto, se recurrió al uso de punteros y estructuras, realizando un manejo de TAD implícito.

La tercer dificultad encontrada fue en el liberado de memoria. El único momento en el cual se utiliza el heap y no es por la librería de grafos, es cuando se lee de stdin, pero no llegamos a liberar el mismo por cuestiones de tiempo, aunque llegamos a una solución eficaz: luego de popear una scope table, guardar la tabla de símbolos en el nodo AST de bloque de código y en el momento de la traducción del nodo en cuestión, previo a la finalización del bloque con el delimitador '}', imprimir todos los free de cada una de las variables que hayan sido alocadas en el heap, donde las mismas se distinguirían del resto por el tipo de dato en cuestión que se les asignaría.

Otra dificultad que encontramos fue con el abstract syntax tree, o en realidad decidir si usarlo o no. Antes de decidir implementar el AST estuvimos algunos días llevándolo directamente a funciones sin pasar por el AST, obviamente esto era demasiado difícil e imposible de mantener por lo que decidimos volver a empezar esa parte del proyecto con AST.

Finalmente, el scope de las variables era algo que nos interesaba hacer pero no entendíamos bien como hacerlo. Investigando bastante y descubrimos que se podía hacer

con un stack de variables que se crean/destruyen a medida que entran en un bloque de código, mediante el uso de mid-rule-action en bison que nos facilitó la utilización del stack de scopes.

Futuras extensiones

Si vemos la definición de grafos, y los posibles algoritmos interesantes encontramos que no hay muchos algoritmos (aparte de los que implementamos) para todo el universo de los grafos. Consideramos que sería interesante proponer un método iterativo para DFS y BFS que permita trabajar en tiempo de ejecución con el nodo en cuestión donde se encuentre el iterador. Además sería de utilidad realizar el traverse a partir de un nodo en particular y no desde el primer nodo insertado. Por otro lado, si restringimos un poco a los grafos, como por ejemplo en árboles, grafos dirigidos, grafos estructurales, grafos de control de flujo, grafos de coloreo, máquinas de estados, podemos encontrar y hacer un lenguaje mucho más extenso. Es por esto que la primer extensión a futuro sería hacer el lenguaje orientado a objetos para permitir distintos tipos de grafos, es decir, como si fuera una clase de herencia.

Se podrían agregar tipos de datos que sean más interesantes a la hora de hacer un programa ya que C provee varios tipos. A su vez sería conveniente que un programador pueda hacer sus funciones fuera del bloque principal, dando más libertad y la posibilidad de reutilizar código.

Finalmente, respecto a la iteración de las sentencia foreach, sería interesante poder iterar desde cualquier nodo en cuestión.

Referencias

Libro Lex & Yacc - Levine

Anexo

Gramática en formato BNF

G = <NT,T,program,P>

T = {ID, STRING_VALUE, ELSE_IF, IF, ELSE, FOR,NUMBER, INT, STRING, DFS, BFS,OPERATOR, BINARY_BOOL_OPERATOR, UNARY_BOOL_OPERATOR,INPUT, OUTPUT, NODE, NODE_VALUE,NODE_ID, START_ARROW, END_ARROW, REMOVE_ADJACENCY, REMOVE_NODE, FOREACH, NODE_ITERATOR, IN, EDGE_ITERATOR, EDGE_WEIGHT, TRAVERSE, WITH, TRAVERSE_ROCEDURE, STARTING, {, }, (,) , COMMENT, ,, ;}

NT = {program, blockcode, code,declaration,nodeDeclaration, nodeValue, nextNodeDeclaration, adjacencyDeclaration, removeGraphNode, removeGraphAdjacency, removeAdjacency, edgeValue, foreachNode, nodeBlockCode, nodeCode, graphAction, nodeAction, nodeProperty, forEachEdge, nodeField, edgeBlockCode, edgeCode, edgeAction, definition, exp, term, conditional, conditionalElse, boolExp, output, outputNode, outputEdge, forLoop, forBlockcode, traverseGraphNode, traverseGraph, traverseProcedure, edgeProperty}

P= {
program::= GRAPH () blockcode

blockcode::= {code}

code::= ; | code declaration ; | code conditional | code forLoop | code definition ; | code output ; | code nodeDeclaration ; | code adjacencyDeclaration ; | code removeGraphAdjacency ; | code removeGraphNode ;| code foreachNode | code foreachEdge | code graphAction ; | code traverseGraph ; | code COMMENT | lambda

declaration::= INT ID | INT ID = exp | STRING ID | STRING ID = STRING_VALUE | STRING ID = INPUT()

nodeDeclaration::= NODE ID = nodeValue nextNodeDeclaration

nodeValue::= STRING_VALUE | ID

nextNodeDeclaration::= , ID = nodeValue nextNodeDeclaration |lambda

adjacencyDeclaration::= ID START_ARROW edgeValue END_ARROW ID

removeGraphNode::= REMOVE_NODE ID

removeGraphAdjacency::= ID REMOVE_ADJACENCY ID

removeAdjacency::= nodeField REMOVE_ADJACENCY nodeField

edgeValue::= NUMBER | STRING_VALUE | ID

foreachNode::= FOREACH (NODE_ITERATOR IN GRAPH) nodeBlockcode

nodeBlockcode::= { nodeCode }

nodeCode::= ; |nodeCode declaration ; |nodeCode conditional |nodeCode forLoop |nodeCode definition ; |nodeCode outputNode ; | nodeCode foreachEdge | nodeCode nodeAction ; | nodeCode adjacencyDeclaration ; | nodeCode removeAdjacency ;| nodeCode traverseGraphNode ; | nodeCode COMMENT | lambda

graphAction::= ID nodeProperty

nodeAction::= nodeField nodeProperty

nodeProperty::= NODE_VALUE | NODE_ID

foreachEdge::= FOREACH (EDGE_ITERATOR IN nodeField) edgeBlockcode

nodeField::= ID | NODE_ITERATOR

edgeBlockcode::= { edgeCode }

edgeCode::= ; |edgeCode declaration ; | edgeCode conditional | edgeCode forLoop | edgeCode COMMENT

edgeCode definition ;| edgeCode outputEdge ; | edgeCode nodeAction ; |edgeCode

edgeAction ;|lambda

edgeAction::= EDGE_ITERATOR EDGE_WEIGHT

definition::= ID = exp OPERATOR exp | ID = term |ID = STRING_VALUE |ID = INPUT()

exp::= term | exp OPERATOR exp

term::= NUMBER | ID

conditional::= IF (boolExp) blockcode conditionalElse

conditionalElse::= ELSE_IF (boolExp) blockcode conditionalElse | ELSE blockcode | lambda

boolExp::= term | boolExp BINARY_BOOL_OPERATOR boolExp | UNARY_BOOL_OPERATOR boolExp

output::= OUTPUT (STRING_VALUE) | OUTPUT (ID) | OUTPUT (NUMBER) | OUTPUT (graphAction)

outputNode::= OUTPUT (STRING_VALUE) | OUTPUT (ID)| OUTPUT (NUMBER) | OUTPUT (nodeAction)

outputEdge ::= OUTPUT (STRING_VALUE) | OUTPUT (ID) | OUTPUT (NUMBER) | OUTPUT (nodeAction) | OUTPUT (EDGE_ITERATOR edgeProperty)

edgeProperty ::= EDGE_WEIGHT

forLoop ::= FOR (declaration ; boolExp ; definition) forBlockcode

forBlockcode ::= { code }

traverseGraphNode ::= TRAVERSE (GRAPH WITH traverseProcedure STARTING nodeField)

traverseGraph ::= TRAVERSE (GRAPH WITH traverseProcedure STARTING ID)

traverseProcedure ::= BFS | DFS
}