Proyecto Especial

Diseño e Implementación de un Lenguaje

Grupo:

Nombre	Legajo	E-mail
Nicolás Ezequiel Birsa	61482	nbirsa@itba.edu.ar
Juan Ramiro Catro	62321	juancastro@itba.edu.ar
Gonzalo Elewaut	61212	gelewaut@itba.edu.ar
Hugo Lichtenberger	(Intercambio) 65323	hlichtenberger@itba.edu.ar

Tabla de Contenidos

Introducción

Los grafos son elementos usados cotidianamente para representar relaciones entre distintos objetos, como distancias entre ciudades, adyacencias entre países (en coloreo de mapas) o probabilidades de tomar caminos diferentes.

Con el avance de la tecnología, más herramientas se presentan para representarlos, pero con el limitante de necesitar entender el lenguaje subyacente que las implementa, además de, en muchos casos, necesitar un "*overhead*" de instrucciones como importación de librerías, o estructuras y tipos en funciones, sólo para poder representarlo en una imagen.

El principal objetivo del lenguaje diseñado e implementado es simplificar y facilitar dicho proceso, pudiendo definir un grafo y sus operaciones o algoritmos a aplicar de forma precisa y exacta con una cantidad mínima de instrucciones, además de obtener una imagen del mismo a la salida del compilador que lo acompaña.

Este informe detalla el desarrollo de dicho lenguaje, desde la semántica inicial hasta la implementación del compilador que traduce sus instrucciones en un ejecutable ***.py***, que a su vez genera la imagen pedida, haciendo uso de las herramientas ***Yacc*** y ***Bison*** para su compilación, además de ***NetworkX*** y ***Matplotlib*** para la generación de dicha imagen.

Consideraciones

La mayor consideración a tener en cuenta es del universo de grafos que el lenguaje acepta, tratándose actualmente de **grafos simples**. Esta limitación se hizo a fines de poder simplificar el código a realizar y enfocarnos en los algoritmos a aplicar.

Librerías

Para poder ejecutar el compilador, además de las herramientas antes mencionadas, se debe tener instalado:

- ***Matplotlib***, que es una librería usada internamente por NetworkX para la generación de imágenes.
- ***Pipenv***, para el manejo del entorno virtual del programa.
- ***Python***, para la ejecución de las librerías ya mencionadas, además del ejecutable de salida.

Tanto el proceso de instalación como las versiones requeridas se detallan en el archivo README.md del proyecto.

Palabras reservadas

En el lenguaje implementado existen una serie de palabras reservadas que no pueden ser usadas para, por ejemplo, nombres de grafos o nodos. Algunas son:

- ***colors***
- ***mst***
- ***to***

No se detallarán los múltiples casos en que un programa puede fallar debido a esto, pero se debe considerar que, en caso de usar alguna palabra requerida por un comando del lenguaje, la falla probablemente se deba a dicho uso.

Desarrollo del Proyecto y Fases del Compilador

El proyecto se dividió en 3 grandes partes. Primero, de **diseño** del lenguaje a implementar, segundo del **frontend** o análisis sintáctico del compilador, y por último del **backend** o análisis semántico y generación de código.

Diseño

Como ya se mencionó, el objetivo principal del lenguaje es poder simplificar y facilitar las operaciones con grafos sin perder “poder de expresión”. Es por ello que se decidió dividir el conjunto de instrucciones en 3 grupos de bloques:

Declaración

Para reducir la cantidad de texto necesaria para declarar un grafo, el usuario tiene la posibilidad de usar un tipo de grafo predefinido como base. Estos tipos de grafos pueden ser:

- Simple: Sin nodos o aristas de base
- Ciclo (**Cn**)
- Rueda (**Rn**)
- Estrella
- Completo (**Kn**)
- Bipartito Completo (**Kn,m**)

Cabe destacar que dichos grafos son sólo una base, por lo que el usuario puede luego elegir remover nodos y aristas.

```
Graph A

Cycle B:
    nodes a, b, c, d, e

Wheel Cgraph:
    center a
    nodes b, c, d

Star anotherGraph:
    center a
    nodes b, c, d, e

Complete allLettersWithoutNumbers:
    nodes a, b, c, d, e

BipartiteComplete LastOne:
    group a, b, c, d
    group e, f
```

Figura 1: Ejemplo declaración de grafos.

Operaciones: Add, Remove y Apply

Los bloques de operaciones son los principales del programa, ya que permiten aplicarle *operaciones básicas*, además una serie de algoritmos. Para lo primero encontramos los bloques **Add** y **Remove**, mientras que lo segundo se realiza en el bloque **Apply**.

Para los bloques **Add** y **Remove** las operaciones posibles son las *básicas* de agregar o quitar nodos y aristas. En cada bloque se puede realizar alguna de estas operaciones, teniendo en cuenta que el grafo sobre el que operan debe haber sido previamente declarado.

```
add to A:
    nodes a, b, c, d, e, f, g, h
    edges a-b, a-d, b-c, b-d, b-f, c-e, c-f, d-e, e-g

remove from A:
    nodes h
    edges c-f
```

Figura 2: Ejemplo add y remove.

En cuanto al bloque **Apply**, éste permite aplicarle una serie de algoritmos al grafo, que son:

- BFS
- DFS
- Coloreo de nodos
- Encontrar nodos de corte
- Borrar nodos de corte (y, por ende, sus aristas)
- Encontrar el árbol recubridor mínimo de cada componente

Es importante recalcar que éstas operaciones se realizan en **orden sucesivo**, por lo que si se desea realizar BFS entre un par de nodos luego de haber borrado los nodos de corte del grafo, el camino encontrado puede resultar distinto. Otro caso sería el de colorear y posteriormente borrar un nodo de corte, donde el coloreo se anula.

Dichas operaciones tienen la posibilidad de ser visualizadas en un archivo de salida agregando un “>” más el nombre del archivo de salida a generar luego de su instrucción correspondiente. Esto es opcional, y solo se puede realizar en las llamadas *operaciones finales*, que en la versión actual del compilador resultan todas las del bloque **Apply**, menos el coloreo, que puede ser visualizado usando el bloque siguiente.

```
apply to A:
    bfs from a to g > bfsAtoG
    dfs from b to e > dfsBtoE
    colors:
        #ff0000 e, d
        #00ff00 c, f
    find cut nodes
    delete cut nodes
    mst > mstA
```

Figura 3: Ejemplo apply.

Generación de imagen

Este bloque consta de una única operación, que es la de generar una imagen de salida del grafo, y tiene la posibilidad de ser llamada **entre cualquier par de bloques**. Es así que se puede visualizar el estado del grafo luego de declararlo, aplicarle algún bloque Add o Remove o hasta luego de un bloque Apply con un simple coloreo para visualizar un coloreo de los nodos.

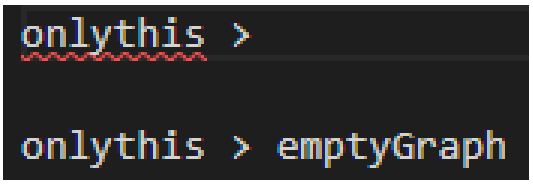

Figura 4: Ejemplo de imprimir un archivo.

Existe la posibilidad de detallar el nombre de los archivos de salida, aunque es algo opcional, ya que de lo contrario dicho nombre será el del grafo. A modo de facilitar este uso, se utiliza una variable file counter global para que cada vez que se imprimen imágenes de esta forma no se sobreescriban.

Frontend

La segunda parte del proyecto fue el analizador sintáctico del lenguaje. Para esto utilizamos tanto **Flex** como **Bison**. Usando **Flex** con su archivo **.l** pudimos distinguir los distintos caracteres especiales que exigen nuestro lenguaje. Cada bloque tiene sus palabras reservadas que nos permiten distinguir el arranque de un nuevo bloque, así como también patrones dentro de los bloques para declarar las cosas.

A su vez también atrapamos patrones genéricos que pueden ser utilizados para distintos nombres de variables (nodos, aristas, colores, nombres de archivos, etc) o parámetros de los algoritmos.

```
{color}                                 { return ColorPatternAction(yytext, yyleng); }
{digit}+                                { return DigitsPatternAction(yytext, yyleng); }
{whitespace}                            { IgnoredPatternAction(yytext, yyleng); }
{letter}+                               { return StringPatternAction(yytext, yyleng); }
```

Figura 5: Ejemplo de flex patterns.

En estos patrones, el usado para nombres de variables es el de **{letter}+** que puede estar compuesto por una serie de letras y números . Esto limita a que los nombres de variables y archivos de salida están compuestos **sólo por letras**.

Este análisis léxico era luego enviado a **Bison** para poder analizarlo semánticamente. Bison con su archivo **.y** era el encargado de fijarse el correcto orden de las palabras, donde todos las estructuras de palabras sean coherentes con la definición del lenguaje. Para este paso decidimos utilizar un enfoque recursivo. Esto se debía a que teníamos varias listas, ya sea de nodos como aristas.

```
program: declaration block                                  { $$ = AppendProgramGrammarAction($1, $2); }
    | declaration                                           { $$ = ProgramGrammarAction($1); }
    ;

block: instruction block                                    { $$ = AppendBlockListGrammarAction($1, $2); }
    | instruction                                           { $$ = BlockListGrammarAction($1); }
    ;

instruction: outputGraph                                    { $$ = BlockListGrammarAction($1); }
    | declaration                                           { $$ = BlockListGrammarAction($1); }
    | addBlockBegin                                         { $$ = BlockListGrammarAction($1); }
    | removeBlockBegin                                      { $$ = BlockListGrammarAction($1); }
    | applyBlockBegin                                       { $$ = BlockListGrammarAction($1); }
    ;
```

Figura 6: Ejemplo de bison actions.

En este paso se aplican algunas limitaciones sintácticas, como que un programa debe siempre iniciar por una **declaración** de algún grafo, o que una declaración debe tener un tipo de grafo seguido de un nombre de variable, que se limitan en este paso para poder realizar una validación temprana de los posibles errores.

A su vez para continuar el enfoque recursivo agrupamos todas las instrucciones que tendrían cada tipo de bloque como un conjunto. De esta manera el bloque de **Apply** tenía una lista de instrucciones y por ende lo que había que chequear es que cada instrucción esté bien hecha.

Para cerrar con el análisis semántico, vemos la posibilidad de intercalar y combinar operaciones de agregar, remover y aplicar los algoritmos. De esta manera el usuario podía utilizar algoritmos en grafos antes y después de modificarlos. Por este motivo utilizamos la idea de bloques, y esto se vio reflejado en **Bison**.

Backend

En esta última etapa encontramos 3 componentes importantes, que son el árbol de sintaxis abstracta (**AST**), la tabla de símbolos, y la generación final de código.

Árbol de sintaxis abstracta (AST)

En este caso se emplea el enfoque recursivo anteriormente planteado en Bison para generar el árbol. Para implementarlo se debieron utilizar **enums** y **uniones**, ya que cada “hoja” del árbol debía retornar un mismo tipo de dato, lo que llevó a utilizar varias estructuras auxiliares, como la de **declaration**, usada para las distintas formas de declarar un grafo en el lenguaje.

```
declaration: GRAPH_TYPE STRING                                          { $$ = CreateSimpleGraphGrammarAction($2); }
    | CYCLE_TYPE STRING BEGIN_BLOCK cycleBlock                          { $$ = CreateCycleGraphGrammarAction($2, $4); }
    | WHEEL_TYPE STRING BEGIN_BLOCK wheelBlock                          { $$ = CreateWheelGraphGrammarAction($2, $4); }
    | STAR_TYPE STRING BEGIN_BLOCK starBlock                            { $$ = CreateStarGraphGrammarAction($2, $4); }
    | COMPLETE_TYPE STRING BEGIN_BLOCK completeBlock                    { $$ = CreateCompleteGraphGrammarAction($2, $4); }
    | BIPARTITE_COMPLETE_TYPE STRING BEGIN_BLOCK bipartiteCompleteBlock { $$ = CreateBipartiteCompleteGraphGrammarAction($2, $4); }
    ;
```

Figura 7: Ejemplo de bison actions.

Tabla de Símbolos

Este componente es la “base de datos” del compilador, y se encarga de validar el programa semánticamente, donde se limitan cosas como el uso de bloques en grafos inexistentes o querer aplicar BFS desde o hacia nodos inexistentes.

Las variables de dicha tabla son los **Grafos**, que se almacenan en una tabla de hash usando su nombre como índice, e internamente cada uno posee una lista recursiva para sus nodos y otra para sus aristas, reutilizando las ya mencionadas en la sección del AST.

Generación de Código

Luego de verificar que el árbol haya sido correctamente formado y la tabla de símbolos no posea conflictos en sus datos, se procede a generar un ejecutable **.py** como código intermedio, que a su vez debe ser ejecutado para poder obtener las imágenes finales detalladas en el programa.

Para poder ejecutarlo se debe previamente configurar un entorno con las dependencias necesarias y tener instalada una versión de python compatible con las versiones usadas. Este proceso se detalla a fondo en el archivo **README.md** presente en el repositorio.

Dificultades

Al principio resultó difícil definir la sintaxis a usar para nuestro lenguaje, pues estábamos acostumbrados a lenguajes de programación generales como C o Java, cuya sintaxis es general y compleja, cuando nosotros solo nos limitamos a trabajar con grafos y las operaciones a hacer sobre ellos. Luego tuvimos que definir que tan simple deberíamos hacerlo, pues de nada sirve hacer un lenguaje específico difícil de aprender y utilizar, y tampoco tenía sentido hacerlo tan simple que no podría cumplir con las funcionalidades que planeabamos o ser extendido a futuro.

Después encontramos dificultades desarrollando el backend: para simplificar las cosas en principio decidimos utilizar la libreria Cytoscape de JavaScript, que ofrecía todo lo que necesitábamos en un lenguaje fácil de escribir a archivo, pues Javascript es muy permisivo en términos de sintaxis. Pero luego descubrimos que no podíamos imprimir las imágenes, lo que era una idea central de nuestro proyecto, por lo que tuvimos que reescribir todo el código generado en Python utilizando NetworkX para el manejo de grafos y Matplotlib para graficarlos.

Futuras extensiones y modificaciones

A futuro se podrían realizarle varias mejoras y optimizaciones del compilador, siendo las más notorias:

- Inclusión de más algoritmos en el bloque apply, como *Dijkstra* para encontrar el camino más corto entre un par de nodos o de Flujo para intentar alcanzar un nodo final, partiendo de un inicial con una “capacidad”. Estos podrían tener una sintaxis del tipo:
    - *dijkstra from a to b*
    - *flow from a to b capacity 10*

- Mayor uso de tablas de hash para guardar los datos en la tabla de símbolos, ya que actualmente sólo los grafos como variables se guardan de esta manera; los nodos y aristas se almacenan como linked lists, reutilizando estructuras del *AST*.

- Implementación de elementos que sean limitaciones del compilador y no de las librerías usadas, como el uso de dígrafos, aristas con peso decimal o nombres de archivos más flexibles.

Referencias

- Documentación de Cytoscape.js
- Documentación de NetworkX
- Documentación de Matplotlib
- Código base usado para el proyecto

Bibliografía

- Levine, John. *Flex & Bison*. O'Reilly Media, 2009.