# LENGUAJE DE PROGRAMACIÓN PARA GRAFOS COMPACTOS

# (Programming language for compact graphs)

# Simón Oroño1, Gerardo Pirela1

1 Laboratorio de Lenguajes y Modelos Computacionales, Departamento de Computación, Facultad Experimental de Ciencias, Universidad del Zulia, Maracaibo - Venezuela.

simonorono@protonmail.com

# Resumen

En la actualidad, las redes son estudiadas como un medio para analizar complejas estructuras relacionales. Debido a esta tendencia, muchas herramientas se han desarrollado para realizar operaciones sobre redes, sin embargo, estas tienden a consumir de forma desmesurada recursos computacionales. En este trabajo se propone una solución a este problema: un lenguaje de programación que soporte en su sistema de tipos grafos almacenados de forma que se pueda reducir el uso de recursos computacionales requeridos para su manipulación. Los fragmentos de gramática libre de contexto son presentados en la forma Backus-Naur extendida. Se empleó la metodología de Programación Extrema para el desarrollo de un compilador para este lenguaje.

**Palabras claves:** lenguaje de programación, grafos, compilador

# Abstract

Currently, networks are been studied as a mean to analyze complex relational structures. It is because of this tendency, that many tools have been developed in order to perform operations on networks; however, these tend to consume lots of computational resources. In this work, a solution to this problem in proposed: a programming language that supports in its type system graph stored in a way that will allow reducing the amount of computational resources required to handle them. Context-free grammar fragments are presented in the extended Backus-Naur form. The Extreme Programming methodology was used to develop a compiler for this language.

**Keywords:** programming language,graphs, compiler

# Introducción

El diccionario de la lengua española define una red como un conjunto de elementos organizados para un determinado fin. Dada esta definición general, las redes pueden modelar muchos tipos de relaciones y procesos en distintos campos del conocimiento, tales como biología, economía, epidemiología, sociología, entre otros. Según Barabási (2002), la construcción y estructura de las redes es la clave para entender el complejo mundo a nuestro alrededor.

De acuerdo con Mitchell (2009), hasta hace muy poco, el estudio de las redes no era visto como un campo en sí mismo. Epidemiólogos estudiaban la transmisión de enfermedades a través de redes de personas que interactúan, ecólogos estudiaban redes tróficas, sociólogos estudiaban la estructura de redes sociales humanas, economistas estudiaban el comportamiento de redes económicas. Todos estos campos trabajaban de forma independiente los unos de los otros, generalmente inconscientes de los avances de los demás.

Por su parte, las matemáticas y ciencias de la computación estudian las redes en un campo conocido como la teoría de grafos, que vio sus inicios en 1736, cuando el matemático Leonhard Euler demostró que era imposible hallar una solución para el problema de los siete puentes de Königsberg. Gracias a Euler y a la teoría de grafos, se puede asegurar que las redes y los grafos tienen propiedades que limitan o aumentan nuestras capacidades de hacer cosas con ellos (Barabási, 2002).

Durante las últimas décadas, un creciente grupo de matemáticos y físicos aplicados se han interesado en desarrollar un conjunto de principios unificadores que gobiernen las redes de cualquier tipo en la naturaleza, sociedad y tecnología (Mitchell, 2009), debido a esto, existe un creciente número de herramientas para manipular redes representadas como grafos: librerías para lenguajes de programación que añaden soporte para grafos, aplicaciones para la manipulación de grafos, lenguajes para recorrer y hacer consultas a bases de datos basadas en grafos, lenguajes para describir grafos, entre otras.

Sin embargo, las redes estudiadas por los distintos campos del conocimiento son, por lo general, muy extensas (por ejemplo: la Internet, redes neuronales humanas, redes y tramas tróficas, redes de transmisión de enfermedades, entre otras) y las herramientas actuales consumen muchos recursos computacionales para poder representarlas y manipularlas. Con una representación optimizada de grafos en memoria se podrían almacenar grafos muy grandes en muy poco espacio; los grafos representados de esta forma se denominan grafos compactos.

En la actualidad, no existen herramientas que manipulen de forma nativa y computacionalmente eficiente grafos compactos, es por ello que se propone con esta investigación la creación de un lenguaje de programación que dé soporte para grafos compactos de forma nativa en su sistema de tipos y en su librería estándar.

# Grafo

Un grafo consiste en un conjunto finito no vacío , cuyos elementos reciben el nombre de vértices o nodos y un conjunto de pares de elementos de , cuyos elementos se conocen como aristas (Rosen, 2012) y representan una conexión entre nodos. Si las aristas de un grafo no representan dirección, se dice que el grafo es no dirigido; si estás son aristas dirigidas (o arcos) se dice que el grafo es dirigido y los arcos se representan como pares ordenados de nodos que son elementos de (Rosen, 2012).

# Lista de adyacencia

La representación en forma de lista de adyacencia de un grafo consiste en un arreglo de listas, una por cada vértice en . Para cada , la lista de adyacencia contiene todos los vértices tal que existe una arista (Cormen y col., 2010). Esta es la representación idónea para grafos dispersos (dónde es mucho menor que ).

# Matriz de adyacencia

Para la representación en forma de matriz de adyacencia de un grafo , se asume que los vértices se encuentran numerados de una forma arbitraria ; de esta forma, la matriz de adyacencia consiste en una matriz de elementos, tal que para cada elemento perteneciente a se cumple que:

# Grado de un vértice

El grado de un vértice en un grafo no dirigido es el número de aristas incidentes con este, con la excepción de los bucles que se cuentan dos veces. (Rosen, 2012).

En un grafo dirigido se tienen aristas dirigidas, por lo que cada vértice cuenta con dos grados: el in-grado, que es el número de aristas dirigidas hacía , y el ex-grado, que es el número de aristas dirigidas desde (Rosen, 2012).

# Distribución de grados

Si se hiciera un recuento en un grafo del número de nodos por cada grado se tendría una distribución de grados del grafo, que sería entendido igualmente como la distribución de probabilidad de un grado en dicho grafo (Barabási, 2012).

La distribución de nodos de un grafo está definida como la fracción de nodos en el grafo con grado . Si existen nodos en un grafo y de ellos tienen grado , entonces .

Los grafos se han podido clasificar en función de la distribución de grados en diversas topologías, siendo las más conocidas:

* Topología exponencial: se da en redes evolucionistas y ocurre siempre que cada nuevo nodo que se añade posea la misma probabilidad de ser enlazado con el resto.
* Topología libre de escala: la distribución de grados de estas redes sigue una ley de potencias de la forma . En estas redes se cumple el hecho de que existan pocos nodos con alta conectividad, pero el grado de conexión de casi todos los nodos es bastante bajo.

# Compilador

Según Aho y col. (2006), un compilador es un programa que puede leer un programa escrito en un lenguaje (fuente) y traducirlo al programa equivalente en otro lenguaje (destino), reportando cualquier error que consiga durante este proceso.

# Estructura básica de un compilador

El programa fuente que será traducido por el compilador pasa por varias fases y cada una de estas transforma una representación del programa fuente a otra (Aho y col. 2006).

* Análisis léxico: es la primera fase del proceso de compilación, y en ella se lee el flujo de caracteres del programa fuente y se agrupan en secuencias significativas conocidas como lexemas (Aho y col., 2006). Por cada lexema, el analizador léxico genera un *token* de la forma:

Donde “nombre” esun símbolo abstracto que se utiliza durante el análisis sintáctico y “valor” apunta a una entrada en la tabla de símbolos para este *token*.

* Análisis sintáctico: es la segunda fase del compilador. El analizador sintáctico lee el flujo de *tokens* resultante de la primera fase e intenta crear una representación intermedia en forma de árbol para describir la estructura gramatical del programa fuente; este árbol, conocido como árbol sintáctico, será utilizado por las siguientes fases del proceso de compilación (Aho y col., 2006).
* Análisis semántico: utiliza el árbol generado por el analizador sintáctico y la tabla de símbolos para comprobar que el programa fuente es consistente con lo establecido en la definición del lenguaje (Aho y col., 2006).

Durante esta fase, se realiza el chequeo estático de tipos, donde se puede verificar que cada operador tenga operandos cuyos tipos coincidan, que el tipo de la expresión retornada por una función coincida con su tipo de retorno, que las funciones sean llamadas con el número correcto de argumentos y los tipos de estos coincidan con la declaración de la función, entre otras cosas.

* Generación de código intermedio: durante el proceso de compilación un compilador puede construir una o más representaciones intermedias del programa fuente (Aho y col., 2006). Los árboles sintácticos usados durante el análisis semántico son un ejemplo de estas representaciones.

Luego del análisis semántico, algunos compiladores generan una representación intermedia de bajo nivel —similar al código máquina—, que podemos considerar como un programa para una máquina abstracta y que debe tener dos propiedades importantes: debe ser fácil de producir y fácil de traducir en la máquina destino (Aho y col., 2006).

* Optimización de código: la fase de optimización de código independiente de la máquina trata de optimizar el código intermedio generado en la fase anterior en pro de producir mejor código destino (Aho y col., 2006). En este contexto, mejor código puede significarse código más rápido, más corto o un código con menor consumo de recursos.
* Generación de código: el generador de código recibe la representación intermedia del programa fuente como entrada y produce un programa equivalente en el lenguaje destino (Aho y col., 2006). Si el lenguaje destino es código máquina, durante esta fase se seleccionan los registros que utilizará cada variable del programa durante su ejecución.

# ANTLR

ANTLR es un generador de analizadores léxico-sintáctico, escrito en Java, y ampliamente utilizado en el ámbito académico e industrial. ANTLR recibe como entrada una descripción formal de un lenguaje en forma de gramática libre de contexto y, a partir de esta, produce un analizador léxico-sintáctico para ese lenguaje capaz de construir un árbol sintáctico automáticamente (Parr, 2012).

# Objetivos

* Realizar una revisión de la documentación de las herramientas existentes para la manipulación de grafos y para la construcción de compiladores.
* Especificar las características del lenguaje.
* Seleccionar los algoritmos y estructuras de datos que compondrán la librería estándar del lenguaje a desarrollar.
* Desarrollar un compilador para el lenguaje.

# Metodología utilizada

La metodología escogida es la de Programación Extrema o “Extreme Programming”, en la que se ejecutan todas las etapas de desarrollo de software de manera casi simultánea en forma de pequeñas iteraciones con el fin de mejorar la capacidad de respuesta del mismo, que por lo general duran una semana (Marsh, 2014). Las fases de desarrollo se adecuaron para el desarrollo de este trabajo de la siguiente manera:

* Planificación

Los requerimientos del proyecto son evaluados al inicio de cada iteración. Durante esta fase también se estiman la duración, costo y riesgos de la iteración. Se crea una visión general del producto final.

* Diseño

Las decisiones acerca de cómo se organizará el código y la lógica del sistema se realizan en esta fase. Se define la forma en la que deben hacerse modificaciones al código existente en caso de ser necesarias.

* Implementación

En esta fase es donde se lleva a cabo el desarrollo del sistema. El código escrito debe respetar estándares y al finalizar la iteración se debe contar con un producto parcial o totalmente listo. Las pruebas deben ser escritas antes que el código.

* Pruebas

Todo el código escrito durante la iteración debe ser probado en esta fase. Las pruebas se escriben primero y la escritura del código debe orientarse a que este las pase.

# Grafos compactos

Un grafo compacto es un grafo representado de manera de que ocupe el menor espacio posible en memoria. Para lograr esto se definió una representación basada en una matriz de adyacencia almacenada como un arreglo de bits, donde cada uno representa un valor booleano.

Esta representación añade restricciones sobre los grafos a representar y que operaciones se pueden ejecutar sobre ellos:

* + Solo puede almacenar grafos no pesados.
  + Los vértices se especifican al momento de la creación del grafo; agregar o quitar un vértice supone crear un nuevo grafo.

# El lenguaje de programación Compact Graphs

Durante esta investigación se desarrolló un nuevo lenguaje de programación fuertemente tipado, con chequeo estático de tipos llamado Compact Graphs (CG); también se empezó a implementar un compilador orientado a generar ejecutables para la Java Virtual Machine (JVM).

El lenguaje cuenta con una sintaxis y semántica que intenta emular a la del lenguaje de programación Go y cuenta con tipos de datos comunes (entero, flotante, booleano, cadenas de texto) con la inclusión de dos tipos de datos: *digraph* y *graph* para grafos dirigidos y no dirigidos respectivamente.

Las reglas gramaticales específicas para especificar un grafo son las siguientes:

edge = “[“ entero “,” entero “]” ;

graph literal = (“graph”|”digraph”) “(“ entero “)” “{“ { edge } “}” ;

Los grafos son inicializados con un número entero que determina la cantidad de nodos que contiene y, opcionalmente, pueden especificarse las aristas de este. La cantidad de nodos de un grafo es inmutable (debido a las restricciones de los grafos compactos), por lo que agregar nodos a un grafo significa la creación de un grafo nuevo; las aristas no tienen esta restricción, por lo que eliminar o añadir no crea un grafo nuevo.

Actualmente las operaciones binarias sobre grafos que soporta el lenguaje son: unión (||), intersección (&&) y complemento (-) del conjunto de aristas si y solo si ambos grafos tienen el mismo número de nodos. La única operación unaria que puede ser ejecutada sobre un grafo es la negación (!) de sus aristas.

# El compilador *cgc*

Como parte de la creación del lenguaje CG se creó a la par un compilador para el mismo, escrito en el lenguaje de programación Kotlin y que genera ejecutables para la JVM.

Fuente: Propia, 2016

El compilador recibe la ruta de un archivo con código fuente y el contenido de este pasa por siete etapas:

1. Análisis léxico: el analizador léxico lee el archivo de código fuente y separa el texto en *tokens*, que serán enviados al analizador sintáctico. Este analizador es generado por ANTLR 4.
2. Análisis sintáctico: el analizador sintáctico recibe los *tokens* resultantes del análisis léxico y verifica que el orden de estos cumpla con la estructura gramatical del lenguaje. Este analizador es generado por ANTLR 4.
3. Chequeo de símbolos globales: es la primera etapa del análisis semántico, en ella se introducen a la tabla de símbolos las variables globales y las funciones. También se verifica que exista una función llamada *main* que no recibe argumentos y no tiene valor de retorno, que es el punto de entrada del ejecutable.
4. Chequeo estructural: es la segunda etapa del análisis semántico en esta etapa se verifica que las funciones con o sin tipo de retorno retornen o no valores, respectivamente; se verifica también que las sentencias de control (*continue* y *break*) se encuentren dentro de bucles.
5. Verificación de tipos: es la última etapa del análisis semántico, durante esta etapa se verifica que todas las operaciones, asignaciones, llamadas a funciones y casteos cumplan con las restricciones de tipos impuestas por el lenguaje. Si al finalizar esta etapa no se ha reportado ningún error se pasa a las dos últimas fases.
6. Pre-generación: durante esta etapa se les asigna un índice numérico tanto a los argumentos como a las variables declaradas dentro de una función. Esta información es requerida por la JVM para cada función.
7. Generación: es en esta etapa todo el código final es generado y empaquetado en formato JAR ejecutable.

# Recomendaciones

...

# Bibliografía

Aho, A., Lam, M., Sethi, R. y Ullman, J. (2006): **Compilers: Principles, Techniques, and Tools.** Segunda Edición. Addison Weaslie. 1000pp.

Barabási, A. (2002): **Linked: The New Science of Networks.** Primera edición. Cambridge, Estados Unidos. Perseus Publishing. 280pp.

Cormen, T., Leiserson, C., Rivest, R., y Stein, C. (2010): **Introduction to Algorithms.** Tercera edición. The MIT Press. 1251pp

Mitchell, M. (2009): **Complexity: A Guided Tour.** Primera edición. Nueva York, Estados Unidos. Oxford University Press. 227-288.

Real Academia Española: **Diccionario de la lengua española.** [Disponible en] http://dle.rae.es/. Consultado enero 2016.

Rosen, K. (2012): **Discrete Mathematics and Its Applications.** Séptima edición. McGraw Hill. 1071pp.